Application Security

Secure Webpage Implementation

by

Victor Jay Sagrado Guarino
Nina Raganová
Ferran Palmada Méndez
Aleix David Martín Álvarez

Professor: Silvia Llorente Viejo
Barcelona, December 12, 2023

Contents

1 Project Overview

Our company operates a dedicated webpage providing a platform for applicants to submit their applications for various job offerings. This online interface serves as a central hub for individuals searching for employment opportunities within our organization. Through this webpage, candidates can access the available job listings and apply for positions that align with their skills and interests.

2 Technical Explanation

2.1 Frontend

The planning stage requires:

- Main Page
- Job Offers Page
- Apply Page
- Review page for applicant
- Review page for recruiter
- About us page

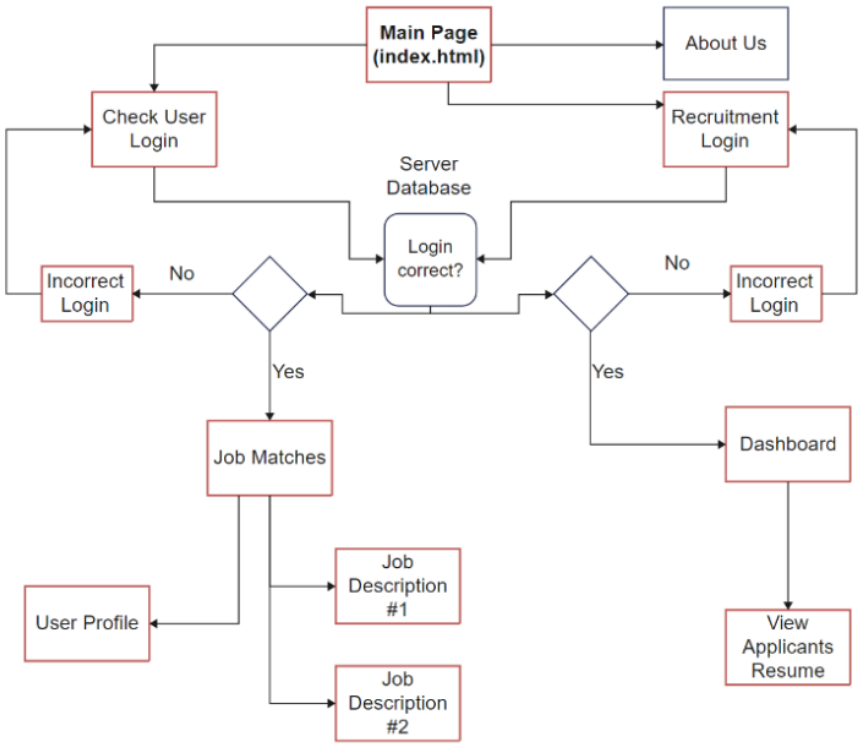

Figure 1: Webpage initial diagram.

Front-end development is responsible for the visual aesthetic, functionality and usability part of the project. It is the user's first contact with the IT Recruitment company online. Additionally, security during web development is integral. TLS/SSL Certificate is also generated in this part of the project. In the initial development of the website, it has a dedicated page for recruitmer's login that is separate from the main landing page designed for users (applicants) to login. It also forwards user profiles to additional pages to view job offers

Further adjustments were made after a series of consultations with the team for the website's seamless integration and security. Ending up with an agreement that it can be made more user-friendly by removing unnecessary pages, particularly the recruitment login page. This can be integrated on the main page, cutting redundancies. Another redundancy that was cut back are the job description pages in the user profile. Final decision was made to make the entire job description/matches be viewed directly on the user profile page, reducing additional clicks and enhancing user experience. Finally, login functions will automatically identify if the user logging in is a recruiter or an applicant through access control, assuming successful verification, and will simply forward them to their corresponding pages.

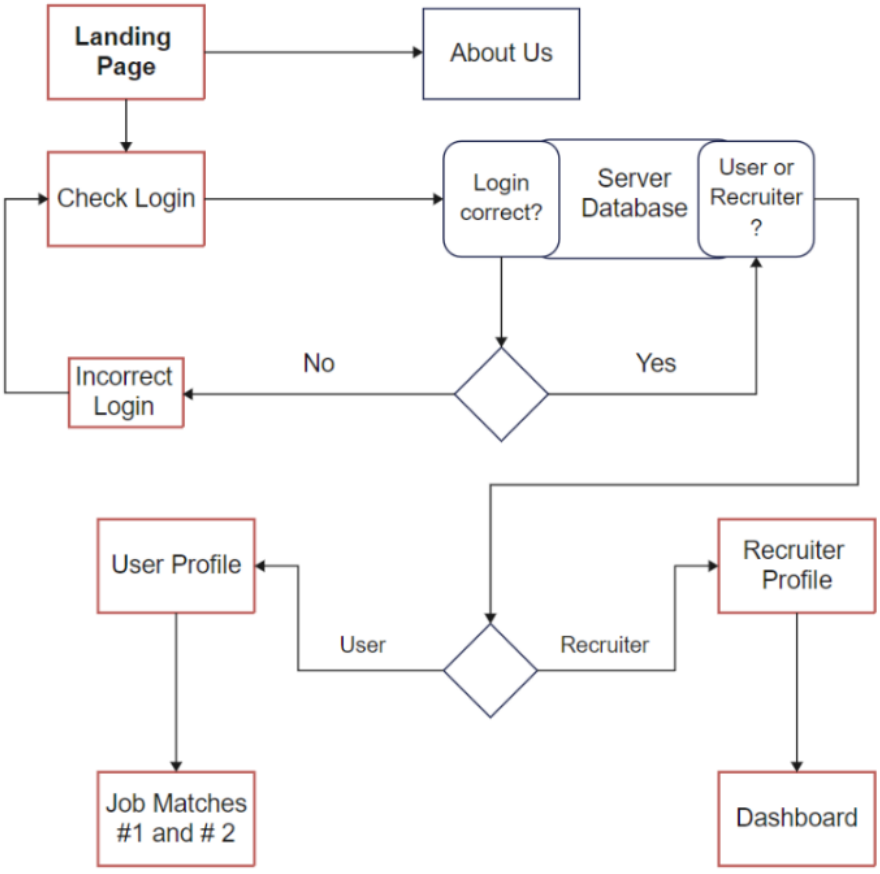

Figure 2: Webpage final diagram.

2.1.1 HTML and CSS

Creating an interactive and visually appealing user interface for the front-end development, the goal is to ensure seamless user interactions, a polished professional aesthetic and a solid security architecture.

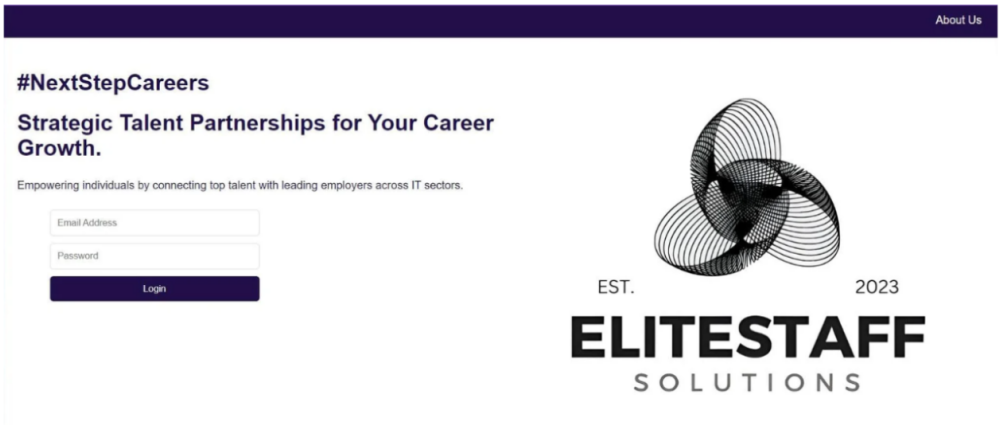

Figure 3: Webpage main HTML.

Users first land on the main page showing the brand and login function. They can then input their login details directly on the main page. After clicking 'Login', the login information is then transmitted to the back-end (database server) in a secure fashion to verify and confirm their identity. If the login details do not match the information from the database, the server will forward them back to the main page with a notification that the username and password is incorrect. If the login is successful, it will then lead them to their respective profiles. Depending on whether the user is an applicant or a recruiter, they can either view the recruitment dashboards of applicants who applied for the job offers matched (recruiter profile), or they will be able to view the available job positions that are recommended by the recruiters (applicant profile).

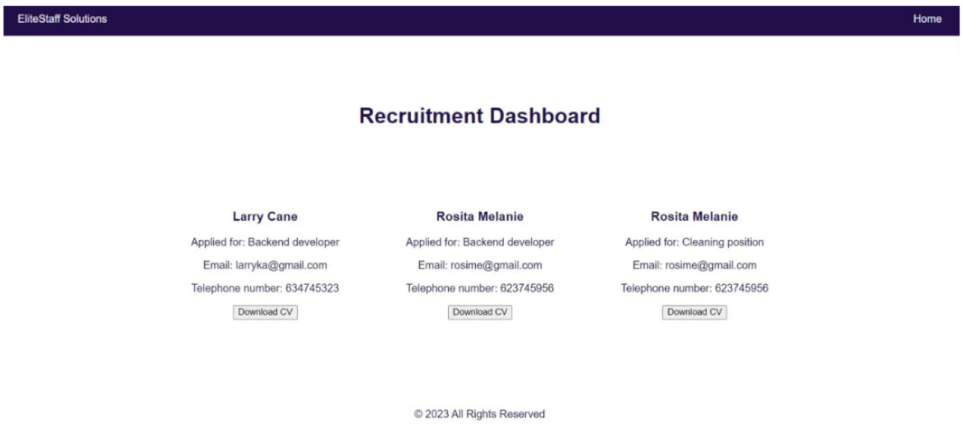

Figure 4: Webpage recruiter's HTML.

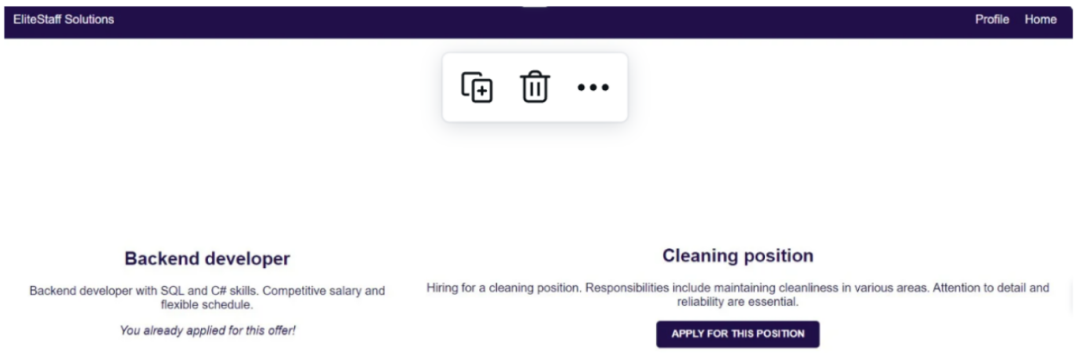

Figure 5: Webpage applicant's HTML.

2.1.2 JavaScript

JavaScript has been used to execute multiple tasks, including requesting information from the database through the backend or undertaking actions in response to button presses.

For instance, the container that encapsulates boxes containing user information in the recruiter's view is dynamically generated. Starting from an empty container, the JavaScript initiates requests to the database to retrieve information and dynamically populate the previously empty container by appending new boxes for each user:

```
1  document.addEventListener('DOMContentLoaded', function () {
2      fetch('/get_recruiter_applicants')
3      .then(response => response.json())
4      .then(data => {
5          const jobApplicantsContainer = document.
               getElementById('jobApplicantsContainer');
6
7
8          // Iterate through job offers and create HTML
               elements
9          data.forEach(jobApplicant => {
10             const jobApplicantDiv = document.
                   createElement('div');
11             jobApplicantDiv.innerHTML = `
12             <div style="margin: 50px;">
13                 <h3>${jobApplicant.Name}</h3>
14                 <p>Applied for: ${jobApplicant.Applied}</
                       p>
15                 <p>Email: ${jobApplicant.Email}</p>
16                 <p>Telephone number: ${jobApplicant.
                       TelephoneNumber}</p>
```

```
17                      <button onclick="DownloadCV('${
                           jobApplicant.Email}', '${jobApplicant.
                           Name}')">Download CV</button>
18                  </div>`;
19                  jobApplicantsContainer.appendChild(
                       jobApplicantDiv);
20              });
21          })
22          .catch(error => console.error('Error fetching job
               offers:', error));
23      });
```

Another example involves the *"Apply for this position"* button, where users call a javascript function to apply for a job. This button establishes a connection to the database, updating the user information:

```
1   function applyToOffer(title) {
2       // Use fetch to send the title to the backend
3       fetch('/apply_to_offer', {
4           method: 'POST',
5           headers: {
6               'Content-Type': 'application/json',
7           },
8           body: JSON.stringify({ title: title }),
9       })
10      .then(response => response.json())
11      .then(data => {
12          // Handle the response from the backend
13          console.log(data);
14
15
16          // Check for a success message or condition
17          if (data && data.redirect) {
18              // Redirect the client-side browser to the
                   specified URL
19              window.location.replace(data.redirect);
20          }
21      })
22      .catch(error => console.error('Error applying to
           offer:', error));
23  }
```

2.2 Backend

The backend consists of a Python program running on our server computer. This Python program provides services to the backend application and serves as an intermediary linking the frontend and the backend with the database.

2.2.1 Flask Implementation

Flask is a Python framework that easily enables communication between the frontend and backend. It allows calling Python functions and routing new pages and HTML content based on user interactions with the frontend.

The server is hosted at the IP address 127.0.0.1:5000, accessible through HTTPS, as detailed in the *Section 2.2.6*. Any computer connected to the same network has access to it.

The simplest example of rendering an HTML, the home page in this case, whenever a user connects to the webpage is as follows:

```
1  @app.route('/')
2  def home():
3      return render_template('index.html')
```

Furthermore, it allows the use of conventional methods such as POST and GET for transmitting and retrieving data or files. For instance, the following code shows the transmission of a file from the server to the user:

```
1  @app.route('/send_document_to_user', methods=['POST'])
2  def send_document_to_user():
3      [...]
4      TOKEN VALIDATION
5      DATABASE QUERY
6      [...]
7
8      download_email_pdf_from_database(applicant_email)
9      document_path = './temp_files/temp_CV.pdf'
10     document_name = 'CV.pdf'
11
12     return send_file(document_path, as_attachment=True,
           attachment_filename=document_name)
```

2.2.2 Login System and Sessions

The login system is based on tokens. Each user has its password hashed with a salt stored in the database, and when it tries to login, the HTTPS request contains its email and password. When the backend receives this request, it performs a query to the database to get the password hash, and then, with a function to check if a hash is generated from

plaintext (also provided by *bcrypt*, the same library that is used to perform the hash), the backend checks if the credentials are valid.

If the credentials are not valid, the backend will return an error to the frontend. Otherwise, it will generate a random token that will be stored in the cookies of the user, and each future request that this user sends will contain this token.

The backend will also create an object of the type *"UserInformation"*, which stores:

- User email and user ID to perform database queries
- User type to control access to the URLs
- Information regarding the encryption system (explained in *Section 2.2.5*)

After creating this object, it will store on a map a key-value pair, where the key will be the random token generated and the value will be the "UserInformation" object. So, when the backend receives a request, it can extract the token from it, and check if the *"active_sessions"* map contains that key. If so, it will mean that the user has performed a login before, and additionally, the backend will have direct access to the most relevant information about the user without performing more queries to the database.

In the following function (the function called to check the user's profile at the frontend), we can see how the token is used to check if the user has logged in, and it also checks its type to control its access to the resources:

```
1   @app.route('/user_profile')
2   def user_profile():
3       if 'session_token' in session:
4           session_token = session['session_token']
5
6
7           if session_token in active_sessions:
8               ActiveUser = active_sessions[session_token]
9               print(f"Active user's mail is {ActiveUser.
                    userMail}, with ID = {ActiveUser.userID}")
10          else: return 'ERROR: Invalid session.'
11      else: return 'ERROR: Not logged in.'
12
13
14      if ActiveUser.userType == UserType.APPLICANT: return
           render_template('user-info.html')
15      else: return "ERROR: Logged user is not an applicant"
```

2.2.3 PDF Sanitation

In the context of PDF sanitation, the *PyMuPDF* package, imported as *fitz*, is used. It is a library with no security breaches reported, which performs the validation of a PDF file to ensure its integrity and absence of malicious code.

```
def is_valid_pdf(file_path):
    try:
        with fitz.open(file_path) as pdf_document:
            # Access the first page to check if the
               document is valid
            pdf_page = pdf_document[0]
            return pdf_page is not None
    except Exception:
        # An exception will be raised if the file is not
           a valid PDF
        return False
```

Once a user uploads a PDF, the file is temporarily stored in the server folder. The function checks the first page information and the file header to determine the PDF's validity. Independently of the result, the temporary PDF is deleted from the server folder. However, it is uploaded to the database only if it is a valid PDF file.

2.2.4 Queries to Database

To connect to the database and perform queries, the backend uses the library *pyodbc*.

When the backend starts, it connects to the database using a login that has been previously configured at the database (explained in *Section 2.3*). This connection is kept open during all the executions.

Most of the routed functions that are called with the frontend requests also call a function that performs a query to the database using the previous connection.

All the queries are parameterized. The fact that this library queries with parameters instead of concatenating the user's input prevents the user from performing SQL injection tests, which is one of our security features.

The following function consists of one of the functions that performs queries to the database, and it can be seen how it creates the query and iterates through the obtained rows to create the map that will be returned to the frontend in JSON format.

```
def get_job_offers_from_database_for_applicant(userID):
    global cursor, connection

    print("Getting job offers for applicant...")
    cursor.execute(f"SELECT O.OfferTitle, O.
       OfferDescription, CASE WHEN OA.UserID IS NOT NULL
       THEN 1 ELSE 0 END AS IsRelated FROM Offer O LEFT
       JOIN OfferApplicant OA ON O.OfferID = OA.OfferID
       AND OA.UserID = ?;", userID)

    rows = cursor.fetchall()
```

```
8
9           jobOffers = []
10          for row in rows:
11              dictionary = {}
12              dictionary['title'] = row[0]
13              dictionary['description'] = row[1]
14              dictionary['alreadyApplied'] = row[2]
15              jobOffers.append(dictionary)
16
17          return jobOffers
```

2.2.5 Encryption System

To protect the user data and ensure its confidentiality, the web application incorporates encryption of personal data, as well as encryption of the files stored in the database. As the application was developed with the Security by Design principle, the database was designed to store all the necessary values to provide for a proper run of the encryption system.

The main objective of the design of the encryption model was to maintain the highest possible level of security, not leaving a space for the attackers to get to the personal information of the users of the application. Therefore, the server does not store any personal information in plaintext, moreover, also no plaintext symmetric encryption keys are stored anyway on the server. *Figure 6* shows the schema of the model and the key derivation phase. Once the user enters their password, it is verified by the comparison with the hashed version (using salted hashing) stored in the database. In case the check is successful, the password is used as an input of the key derivation function (PBKDF2), which produces a value that serves as the seed of a pseudorandom generator. This step guarantees the deterministic behavior of the subsequent asymmetric keypair generation. Using this procedure, the server does not need to store the private key of the user, as it is able to regenerate it at every login.

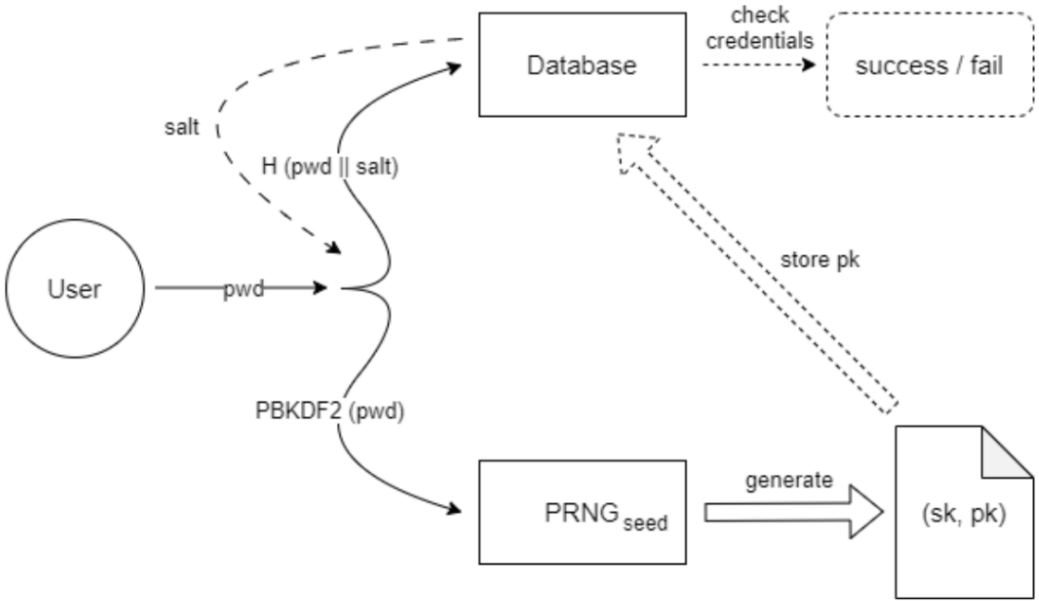

Figure 6: Asymmetric keypair generation.

After the registration of a user, the encryption system generates the asymmetric keypair, stores the public key in the database for later use, generates a symmetric key which is used for the encryption of the personal information about the user, as well as their files (uploaded CVs), and encrypts the user-provided personal data (shown in *Figure 8*). The encryption key is then encrypted with the public key of the user and stored into database. Once the user uploads a CV, it is also encrypted using the same symmetric key. After a successful login, the server regenerates the asymmetric keypair, which means that it can decrypt the symmetric key with the private key of the user, even though it is not stored anywhere.

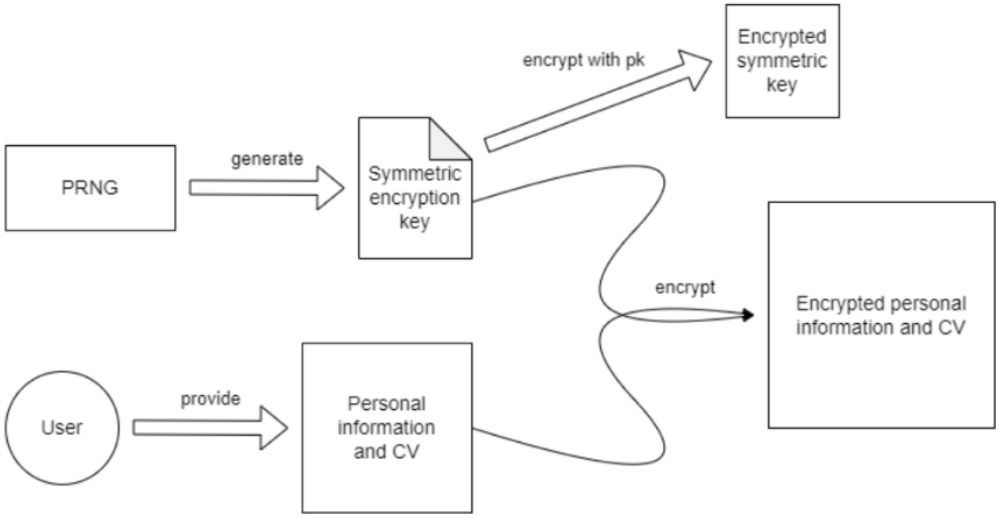

Figure 7: Personal data and symmetric key encryption.

As the main purpose of the application is to provide a platform where the users can apply to job offers, the encrypted information must be available to the recruiters that are in charge of the positions for which the users apply. Therefore, the encryption key has to be shared with the recruiter at the time of the application for the role. As can be seen in *Figure 8*, to share the encryption key with the recruiter, the server has to first decrypt the encrypted symmetric key retrieved from the database using the applicant's private key, then encrypt it with recruiter's public key, and finally, store the value in the database. Once the recruiter logs in, their private key is regenerated, and they are able to decrypt the symmetric key and therefore also applicant's personal data and files.

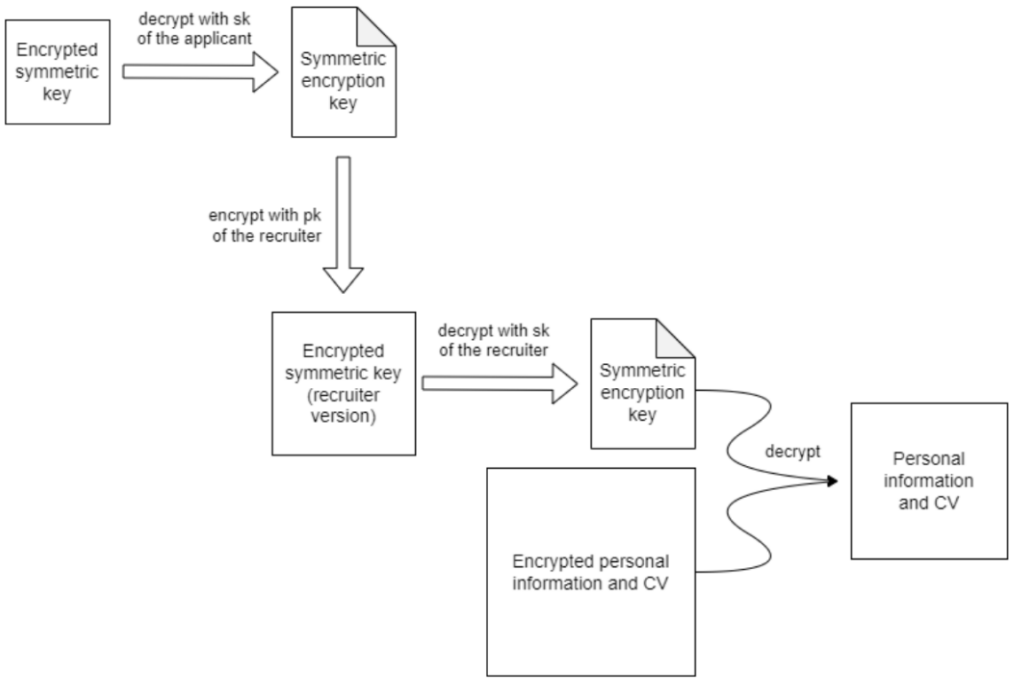

Figure 8: Symmetric encryption key sharing in the application process.

Even though the encryption keys are derived from the user's password, there are still ways how to deal with forgotten passwords, although those are not implemented in the application. For the applicants, it might be best if they receive a message that they can reset their password, but they will lose the access to their information and uploaded files. Nevertheless, as the recruiters have a copy of the symmetric key (encrypted with their public keys), the applications will not be lost, and the application process can continue. However, this approach might be a little unpleasant in case the recruiter forgets their password, because they would lose the access to information about all their applicants. Therefore, they need a different mechanism to deal with forgotten passwords. As the web application is designed for a recruiting company, it is likely that the recruiter needs to access the portal during normal working hours. So, if they have not received any applications yet, it is safe to just regenerate their asymmetric keypair. In case they have already received some applications, the application might use a "backup recruiter mechanism", which basically means that each job offer will have assigned also another recruiter ("backup") and in case the main recruiter forgets their password, the "backup" recruiter receives an alert to

log in to the portal and the server can automatically encrypt the symmetric keys of the applicants corresponding to the roles they have in common. Even though the chance that both recruiters forget their passwords at the same time is quite low, the probability can be reduced even more by having multiple "backup" recruiters instead of a single one.

2.2.6 Certificate Generation

Web pages are developed as a secured website from the beginning, demonstrating the team's dedication to make security integral in every stage of the project development. Keeping that in mind, generating TLS/SSL certificates is focused on making sure that users are not landing on phishing sites.

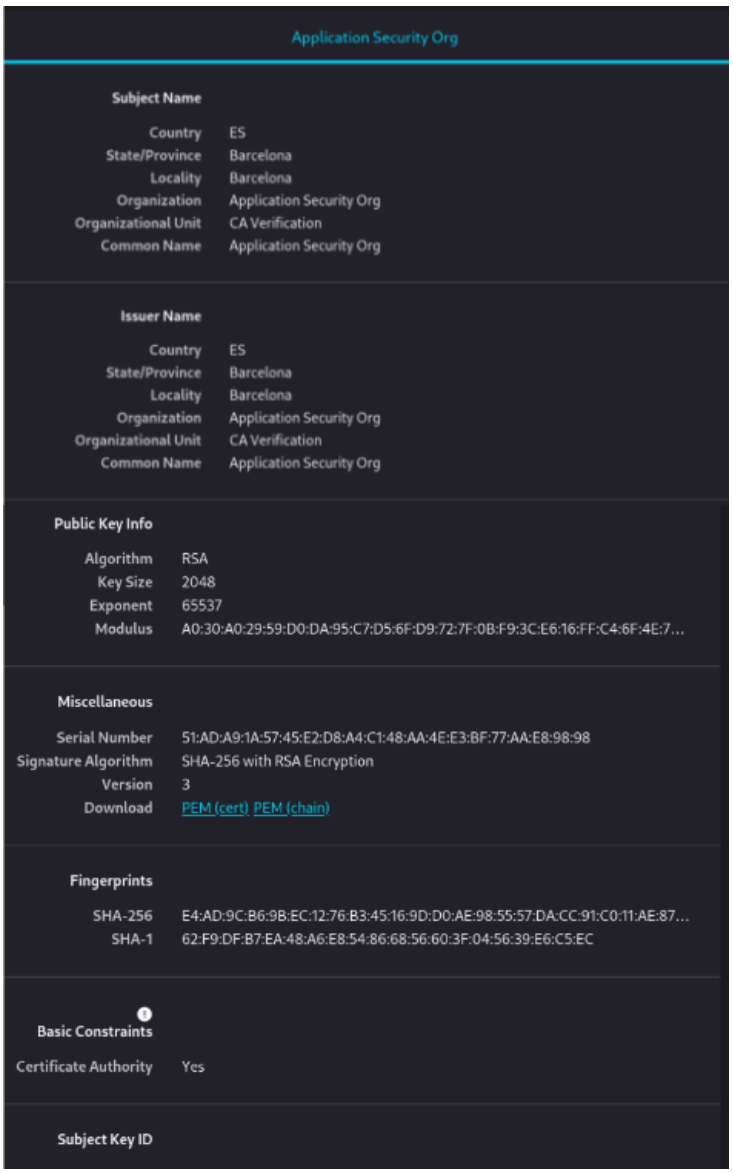

Figure 9: Self-signed certificate.

The self-signed CA certificate is generated with a standard RSA public key and uses

SHA-256 for signing. It is valid for one year and includes basic constraints indicating it is a CA certificate. This CA is later installed on the server.

2.3 Database

For the database, the solution uses "Microsoft SQL Server". The server computer has a SQL Server running on it at the same time that the backend is running (which, in the future, could be moved to another computer if it was necessary), and this server has been configured with SSMS (Sql Server Management Studio).

The data base schema consists of 5 tables, as shown in the *Figure 10*.

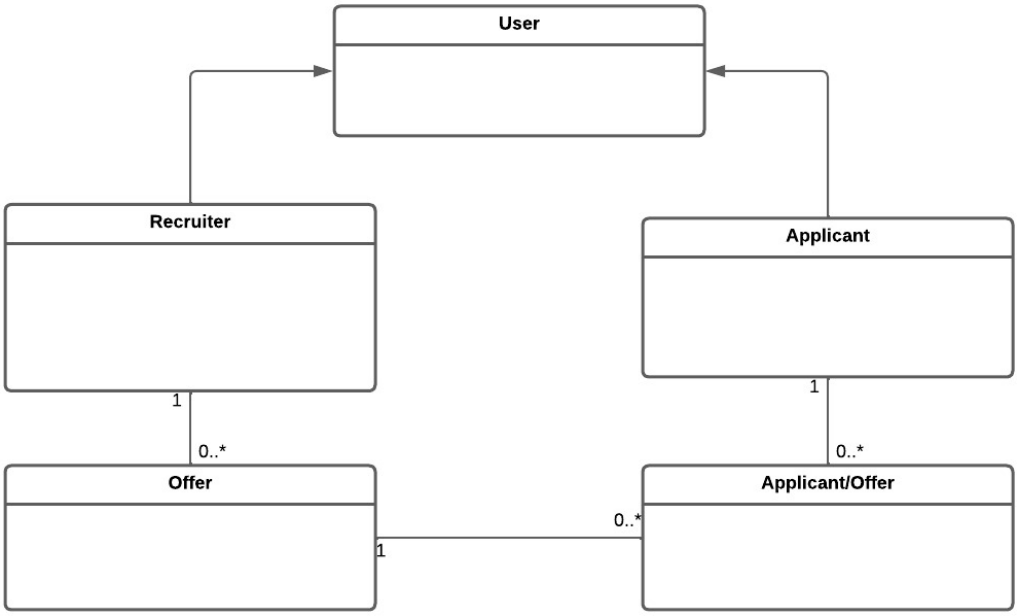

Figure 10: Database diagram.

As can be seen, the database has a *"User"* table that saves all the information common to each user type. From this table, derive two other tables, Recruiter and Applicant, so their unique information (the applicant CV, for example) can be stored without having empty fields or repeated values. There is also an *"Offer"* table, which stores the information of the current job offers related to the recruiter table, so each offer has a responsible recruiter, and a single recruiter can handle multiple offers. Finally, there is a table to store the relationships between the applicant and the offers (which applicants applied to each offer), as the relationship is multiple on both sides.

This database uses the *FILESTREAM* functionality of SQL Server, which allows to store and manage large binary data (such as documents) outside of the database files while still providing seamless integration with the database engine. This functionality is used to store the applicant's CVs (which are pdfs) and the user's private keys (pem files).

In order to allow the backend to connect to the database, the database has *"SQL Authentication Mode"* and has a user enabled with the necessary permissions to perform the desired queries.

References

[1] **Author(s):** Mihirs16. **Title:** *Project Darwin*. **Retrieved from:** *https://github.com/mihirs16/Project-Darwin/blob/master/templates/*.

[2] **Author(s):** Shubham1710. **Title:** *Fenice-Network*. **Retrieved from:** *https://github.com/shubham1710/Fenice-Network/tree/master/fenice/candidates/templates*.

[3] **Author(s):** Emmet. **Title:** *Emmet Documentation*. **Retrieved from:** *https://docs.emmet.io/cheat-sheet/*.

[4] **Author(s):** MDN Wb Docs. **Title:** *Learn to style HTML using CSS*. **Retrieved from:** *https://developer.mozilla.org/en-US/docs/Learn/CSS*.

[5] **Author(s):** Juan Bautista. **Title:** *Implementing a root Certification Authority with OpenSSL*. **Retrieved from:** *https://atenea.upc.edu/course/view.php?id=86013*.

[6] **Author(s):** PyPDF. **Title:** *PyPDF Documentation*. **Retrieved from:** *https://pypdf.readthedocs.io/en/stable/*.

[7] **Author(s):** PyCryptodome. **Title:** *PyCryptodome Documentation*. **Retrieved from:** *https://pycryptodome.readthedocs.io/en/latest/index.html*.

[8] **Author(s):** PyCryptodome. **Title:** *PyCryptodome Documentation*. **Retrieved from:** *https://pycryptodome.readthedocs.io/en/latest/index.html*.

[9] **Author(s):** PyMuPDF. **Title:** *PyMuPDF Documentation*. **Retrieved from:** *https://pymupdf.readthedocs.io/en/latest/*.

[10] **Author(s):** Flask. **Title:** *Flask Documentation*. **Retrieved from:** *https://flask.palletsprojects.com/en/3.0.x/*.

[11] **Author(s):** Bcrypt. **Title:** *Bcrypt Documentation*. **Retrieved from:** *https://github.com/pyca/bcrypt/*.

[12] **Author(s):** Secrets. **Title:** *Secrets Documentation*. **Retrieved from:** *https://docs.python.org/3/library/secrets.html*.

[13] **Author(s):** BinASCII. **Title:** *BinASCII Documentation*. **Retrieved from:** *https://docs.python.org/3/library/binascii.html*.

[14] **Author(s):** BinASCII. **Title:** *BinASCII Documentation*. **Retrieved from:** *https://docs.python.org/3/library/binascii.html*.