

Talent Verify

Fincheck Software Engineering solution by Dean Tinotenda Maponga
(deanmaponga@gmail.com)

1. Version control

A public github repository was created to host the system. The repository can be accessed by following this link <https://github.com/DeanMaponga/fincheck> , the running website can be access at this link <http://18.191.1.123>

2. Django & React + Cloud

A client server architecture was used for the project. The Django system was treated as the server (backend) which stored all data in a sqlite database which was accessed using django models. The end users interacted with the system using the frontend, I am familiar with angular so I created an angular frontend instead of react. Both the backend and froment were hosted on a AWS virtual machine with the same ip address but different ports. In the coming weeks and months I hope to learn react and implement the frontend using react.

3. Database tables

There database tables were created:

- a. The company table contained the details about each company and it has the following columns:
 - i. Name - The name of the company
 - ii. date_of_registration - a DateField which showed when company was registered
 - iii. Registration_number - A character field showing company registration number
 - iv. Address - A TextField which stored the address of the company
 - v. Contact_person - a character field which showed the name of the person to contact at the company
 - vi. departments - a TextField which showed the departments the company had.
 - vii. Number_of_employees - An integer which showed the number of employees present at the company
 - viii. contact_phone - a character field which showed the contact details of the company, regex was used to validate phone numbers.

- ix. Email - an email field which showed the email of the company
- b. The Employee Table contained details about each employee and it had the following columns:
 - i. Name - a character field showing name of employee
 - ii. Employee_id - the employee id, was a character field which was encrypted
 - iii. Company - this was a foreignKey which linked the employee to the company table
 - iv. Department - This was a charfield which stored the details of the department the employee worked for
- c. The Role Table contained the details of the role that each employee had, this table had the following columns:
 - i. Employee - A foreign key that linked role to an employee
 - ii. Role - A character field that stored the role of the employee in the company.
 - iii. Start_date - a date field that stored the date an employee started working in the role.
 - iv. end_date - a date field that stored the date the employee stopped working in the role
 - v. Duties - the duties the employee had to do when they performed the role.

4. Encryption

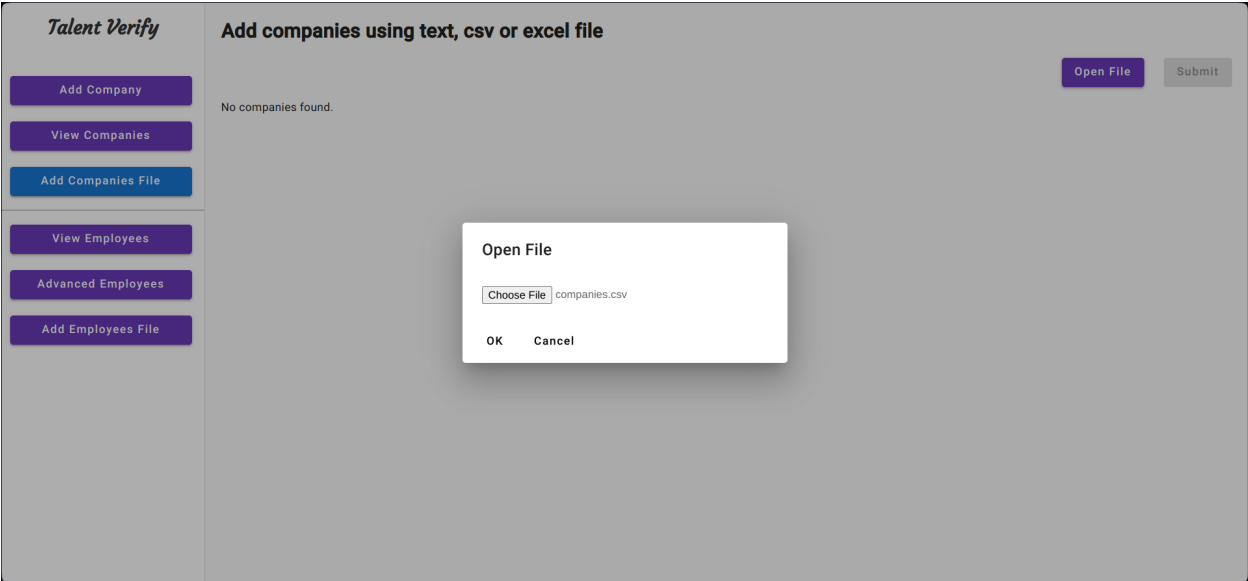
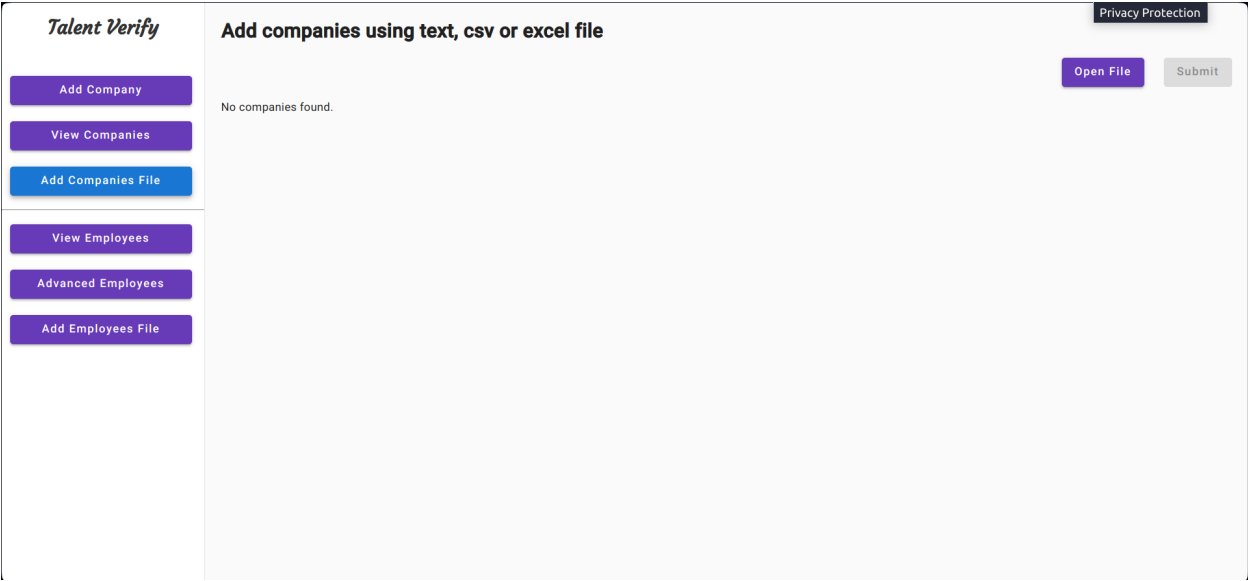
Some fields were encrypted to ensure data security incase of database breaches. The following columns were encrypted:

- a. Company registration number
- b. Employee phone numbers
- c. Company phone numbers
- d. Employee Ids

Encryption was achieved using `django_cryptography` modules in `models.py`

5. Company Bulk update interface

An interface was made to allow companies and employees to be added in bulk by a company using a .txt, .csv or .xlsx file as shown in the screenshots below:



Talent Verify

Add Company

View Companies

Add Companies File

View Employees

Advanced Employees

Add Employees File

Add companies using text, csv or excel file

Open File

Submit

company x

2023-02-02

Registration Number: 123

Address: 12 harare Street Harare

Contact Person: Mr John

Departments: Technology Administration

Number of Employees: 5

Contact Phone: 0772123456

Email: x@email.com

school c

2001-07-02

Registration Number: 1234

Address: 12 harare Street Harare

Talent Verify

Add Company

View Companies

Add Companies File

View Employees

Advanced Employees

Add Employees File

Add companies using text, csv or excel file

Open File

Submit

✓

Company added successfully!

OK

The files allows bulk updates of companies

6. Search employee interface

An interface was made to allow employees to be searched using name, employer, position, department, year started, year left as shown in the screenshot below:

Talent Verify

Add Company
View Companies
Add Companies File
View Employees
Search Employees
Add Employees File

Advanced Employee Search

Name
Employee name
Employer
Enter company name
Department
Company department
Position
Employee role in company
Start Year
YYYY
End Year
YYYY
Submit

Talent Verify

Add Company
View Companies
Add Companies File
View Employees
Search Employees
Add Employees File

Search Results

Andrew
Employee ID: 12
Company: first Company
Department: Management
Role: Manager
Start Date: 2020-10-28
End Date:
Duties: Stores manager

The submit button was active when an input was made and a search results page showed the employees that were found

7. TalentVerify bulk update interface

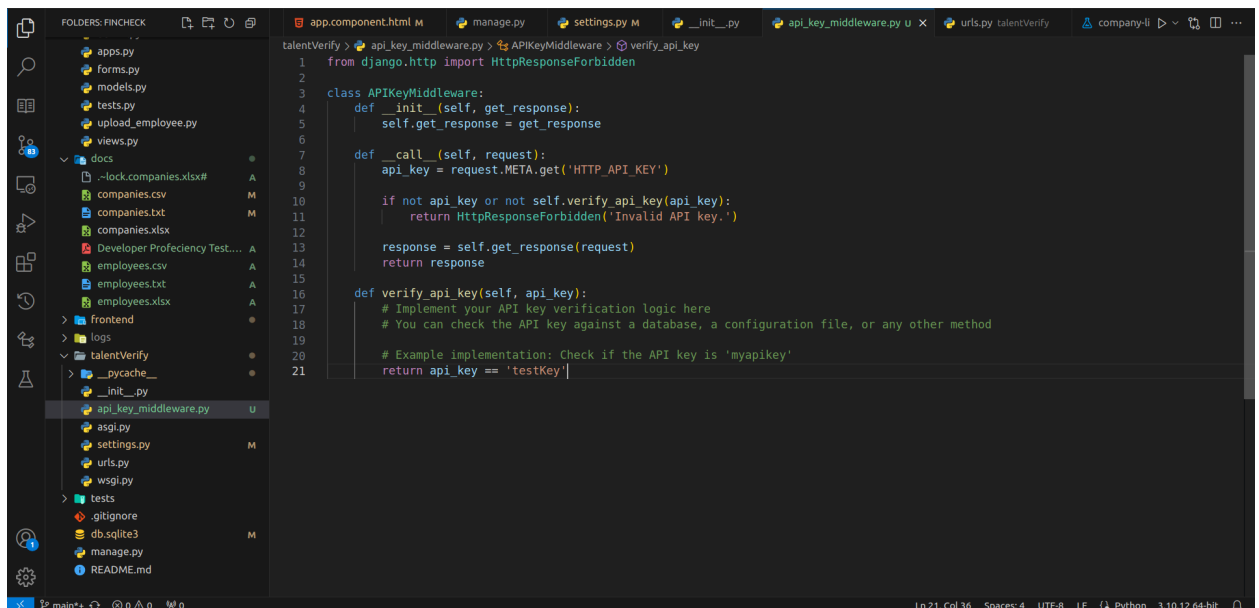
The bulk employee was added to the sidebar for talent verify employees to add employees or companies in bulk or update them

8. Potential design weakness & mitigations

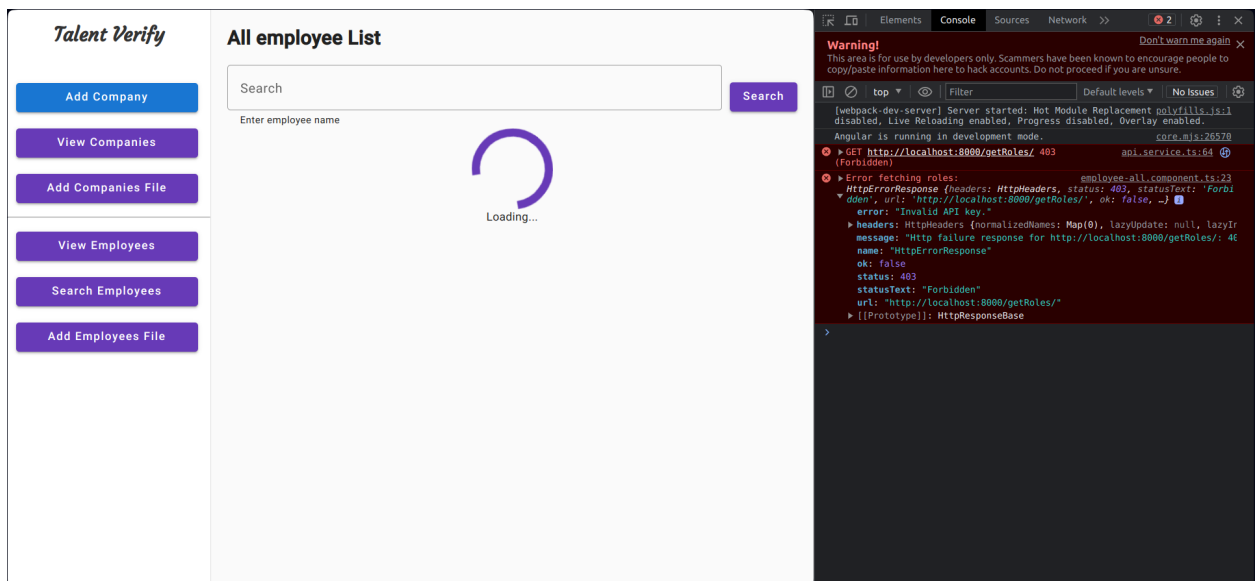
The weakness of the current system is that the system had no account management. These were not implemented due to time restrictions on the scope of the project,

given more time in later versions user accounts should be created. Future versions should include a more robust authentication system, which includes the creation of an client account system where frontend users login to use the system and the actions of each user are logged and restricted based on their privileges as well as api key invalidation after a certain time period.

API key middleware was created to protect api endpoints from unauthorized users, this, the following screenshots the middle ware and a rejected api request when the key was not provided



```
1 from django.http import HttpResponseRedirect
2
3 class APIKeyMiddleware:
4     def __init__(self, get_response):
5         self.get_response = get_response
6
7     def __call__(self, request):
8         api_key = request.META.get('HTTP_API_KEY')
9
10        if not api_key or not self.verify_api_key(api_key):
11            return HttpResponseRedirect('Invalid API key.')
12
13        response = self.get_response(request)
14        return response
15
16    def verify_api_key(self, api_key):
17        # Implement your API key verification logic here
18        # You can check the API key against a database, a configuration file, or any other method
19        # Example implementation: Check if the API key is 'myapikey'
20        return api_key == 'testkey'
```



Talent Verify

All employee List

Search

Enter employee name

Loading...

Warning! Don't warn me again

This area is for use by developers only. Scammers have been known to encourage people to copy/paste information here to hack accounts. Do not proceed if you are unsure.

[webpack-dev-server] Server started: Hot Module Replacement enabled, Live Reloading enabled, Progress disabled, Overlay enabled.

Angular is running in development mode. core.js:26578

GET http://localhost:8000/getRoles/ 403 (Forbidden) api.service.ts:64

Error fetching roles: employee_all.component.ts:23

HttpErrorResponse {headers: HttpHeaders, status: 403, statusText: 'Forbidden', url: 'http://localhost:8000/getRoles/', ok: false, ...}

error: "Invalid API key."

headers: HttpHeaders {normalizedNames: Map(0), lazyUpdate: null, lazyIr

message: "Http failure response for http://localhost:8000/getRoles/: 40

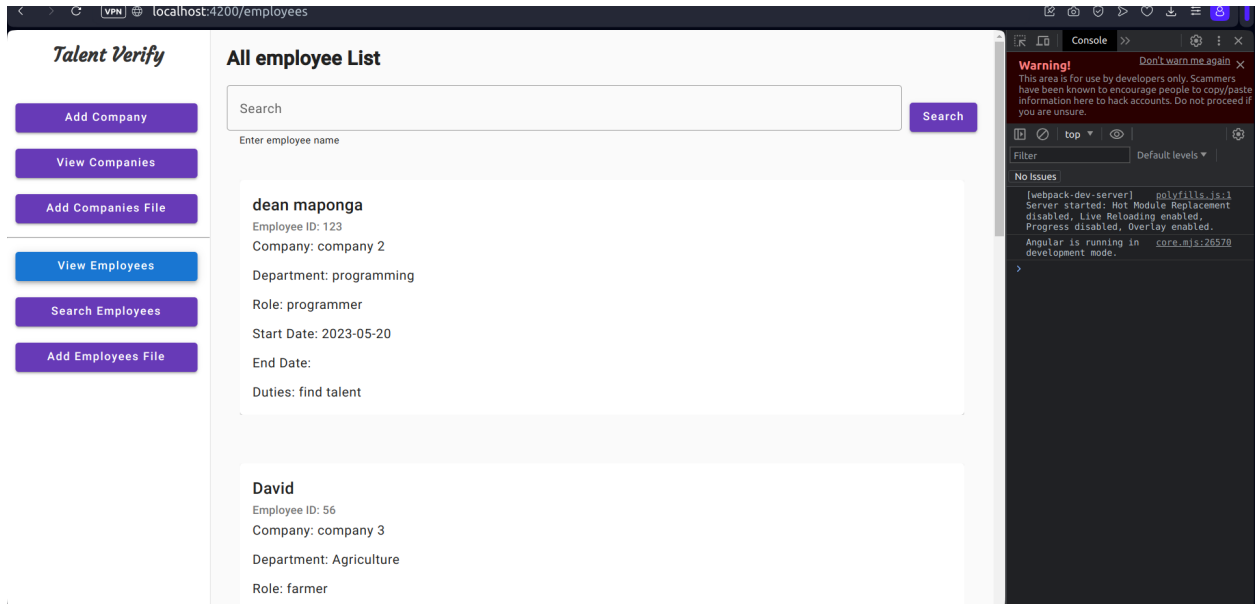
ok: false

status: 403

statusText: "Forbidden"

url: "http://localhost:8000/getRoles/"

[[Prototype]]: HttpResponseBase



The above screenshot shows api successfully worked when apikey was provided

The system was hosted on a single virtual machine this could pose an issue during high load events or during maintenance. Auto Scaling and load balancers should be used to scale the system to handle more traffic.

The system used a sqlite database which was stored on the same virtual machine as the backend, this may cause an issue when the database contents outgrow the storage space of the virtual machine. A standalone database virtual machine should be used instead which will be able to independently scale up or down depending on data usage and traffic. Instead of using sqlite, a mysql or postgresSQL database can be used to store django data.

9. Security tests

Various tests were made to ensure the system worked securely, such as the following:

- a. **Input validation** - A django database model was used where the datatype of each field was specified. The model also used regex for validation, for example the employee phone number. Incoming requests were checked to ensure that they were not malformed, if so an HTTP error 400 bad request was returned before processing the request
- b. **Authentication & Authorization** - API keys were used to access the API endpoints of the django api. The keys were pre shared with the client who used the Angular frontend to access the api

- c. **Cross Site Request Forgery (CSRF)** - This was avoided by using http methods such as POST, PATCH and DELETE for sensitive data instead of GET requests alone. The use of API keys ensured that the authorized user could access endpoints.
- d. **SQL injection** - The use of django models and serializers ensured that the code did not have direct access to SQL. Django automatically escapes parameters passed to SQL queries. This protects against malformed user input, developer errors were avoided through the use of models.
- e. **Serialization issues** - Complex data types cause errors during serialization. The use of django serializers ensured that all complex serialization was handled by the framework and exceptions were raised and handled when serialization could not occur.
- f. **Logging & Monitoring** - log files were recorded in the logs directly and monitored from time to time to ensure that the system was performing optimally, security errors could be detected based on the recorded logs

10. Stress tests

Various tests were done to ensure that the system worked as required:

- a. Load Testing - A simulation was done to check how the system worked under heavy loads
- b. Stress test - Multiple malformed inputs were given to the system to test error handling
- c. Database test - high read/writes were performed to check how the system performed under load.

11. Different solution using different technology

The Django backend could have been replaced with a Express.js server in a node.js environment or ASP.NET in a C# environment, depending on the technology stack used at talent verify. The front end could have been replaced with Angular or Flutter which runs on various platforms.

12. Time constraint Shortcomings

Instead of presharing api keys a more robust authentication solution involving account creation and signin could have been created.

13. Cloud deployment

The Django backend and angular frontend were both deployed on a AWS virtual machine which ran in the cloud. An nginx server served the angular website and CORS was set up to ensure the angular + django requests succeeded.