

Criterion C

Table of Contents

1. Used Framework.....	2
1.1 Using the Flask Framework	2
2. File Management:.....	3
2.1 File Storage.....	3
2.2 File preview and removal	5
3. Usage of database	6
3.1 usage of SQLite database.....	6
3.1.1 using flask_migrate library to generate the database.....	6
3.1.2 using flask_flask_sqlalchemy library to manipulate the database	6
3.2 the implementation of database model for each entity	7
4. User account management	8
4.1 User sign-up	8
4.2 User log-in	9
4.3 "Forget password" and E-mail verification process.....	10
4.4 using python decorator and flask session to implement login authentication	11
5. Implementation of GUI.....	13
5.1 Usage of existing tools.....	13
5.1.1 Vue.js 3	13
5.1.2 Element-UI Plus.....	13
5.2 Calendar.....	14
6. System Security Insurance.....	14
6.1 Database encryption	14
6.2 Data Verification—Password Double-entry	15
6.3 Encrypted display when keying in a password	15

1. Used Framework

1.1 Using the Flask Framework

It is wise to use this existing tool because Flask offers a highly practical method for creating an HTTP server with robust functionality. A Flask object is created in the program's entry module and used as the server object. Before the server begins, blueprints and a set of routes are registered to it. "app" is an instance of Flask that accepts the name of a package or module as an argument. Using the `app.route` decorator saves the relationship between the URL and the executed view function to the `app.url_map` property. The service is started by executing `app.run()`. By default Flask only monitors to the local 127.0.0.1 address of the virtual machine on port 5000.

```
from flask import Flask, session
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
from sqlalchemy.event import listen
from sqlalchemy.pool import Pool
import config
from flask_mail import Mail, Message

app = Flask(__name__)
# auto refresh
app.debug = True

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///app.db'
```

Figure 1.1 Using the Flask framework

```
from app import app
# from api import account
from views.views import user, templates, employee, manager

# if not os.path.exists('./upload'):
#     os.mkdir('./upload')

if __name__ == "__main__":
    app.run(port=5000)
```

Figure 1.2 app.run() execution

2. File Management:

2.1 File Storage

The process that requires file storage is for the employee to submit a request for reimbursement. Users could attach files to their applications. There are mainly two kinds of files: receipts and other supporting materials. If the file is of image format (ex. .png, .jpg, .jfif), then a thumbnail would be represented on the application filling window as a user-friendly function. These files would be stored after the employee submitted the application. Later, when the manager processes these applications, those files would be reloaded and presented.

Since this reimbursement system is an internal system and there is no network transmission complexity involved, for file storage, my solution is to store them directly on the disk, instead of turning them into other forms like base64 or byte string, then stored in the database. One advantage is that this could promote the query efficiency of database, while maintaining the tidiness of database scheme. (Saito, Bershad and Levy 1999, p.124-139) So I adapt `FileStorage.save()` for file storage.

```
# randomly generate a file name
filename = ''.join(random.choices(string.ascii_letters + string.digits, k=8)) + \
    os.path.splitext(file.filename)[1]
file_path = os.path.join('static', 'upload', filename)

# save files into static/upload path
file.save(file_path)
img1_list.append('/{}/{}/{}'.format('static', 'upload', filename))
```

Figure 2.1.1 Generating file name and file storage

One difficult point is that we cannot ensure the filename in each application would be unique, and if repeated filename exists, we could not separate them apart. So, its essential to add prefix or additional information to uniquely identify each file. My solution is to use a randomly generated string of 8 length, consists of char and number. This encoding method could support up to $(26 * 2 + 10)^8 = 2.1 * 10^{14}$ types of filename. So it could promise there's no repeat filename.

The attached file will be stored using the randomly generated fine name:



Figure 2.1.2 Randomly generated file names preview

Then we could just store the filename into the database, which costs much lower space than storing entire file into the database. This means Database is the index where the table corresponds to the file name.

img1_list	img2_list	status
[]	[]	no
[]	["/static/upload/SBVgBRd5.	yes
["/static/upload/Rx35ZZeG.	["/static/upload/mtHh4Vgj.	waiting
["/static/upload/1Gaw6mAV.	["/static/upload/oJ8qil95.	yes

Figure 2.1.3 File name indexes in the database

```

img2_list = []
for file in files:
    # generate a random filename
    filename = ''.join(random.choices(string.ascii_letters + string.digits, k=8)) + \
        os.path.splitext(file.filename)[1]
    file_path = os.path.join('static', 'upload', filename)
    # save files into static/upload path
    file.save(file_path)
    img2_list.append('{}/{}/{}'.format('static', 'upload', filename))

# store in database
apply = Application(applicant_id=applicant, doc_type=form_type, amount=price,
                    title=title, department=department,
                    project=project,
                    date=datetime.datetime.strptime(application_date, '%Y-%m-%d').date(),
                    desc=apply_content,
                    expense_list=expense_list, status='waiting',
                    img1_list=img1_list, img2_list=img2_list)

db.session.add(apply)
db.session.commit()

```

Figure 2.1.4 Storage of file name in database

2.2 File preview and removal

To make the file thumbnails appear when the user uploads a file, I first created a URL that can be accessed to the file. When `dialogVisible.value` is true, the file thumbnails are displayed and vice versa. To achieve the functionality that a user can delete the uploaded file, I used the `console.log` method to achieve file removal.

```

const associatedFileList = ref([])

const associatedDialogImageUrl = ref('')
const associatedDialogVisible = ref(false)

const associatedHandleRemove = (uploadFile, uploadFiles) => {
    console.log(uploadFile, uploadFiles)
}

const associatedHandlePictureCardPreview = (uploadFile) => {
    dialogImageUrl.value = uploadFile.url
    dialogVisible.value = true
}

```

Figure 2.2.1 File Preview and File Removal

3. Usage of database

3.1 usage of SQLite database

3.1.1 using flask_migrate library to generate the database

The database I chose is SQLITE, which is a lightweight database, easy to query and develop. I use **Flask-migrate** extension as the database manager. With **Flask-migration**, I could easily back up database snapshots, roll back to recent editions, and migrate databases between different machines and environments.

```
def run_migrations_online():
    """Run migrations in 'online' mode.
    In this scenario we need to create an Engine
    and associate a connection with the context.
    """

    # this callback is used to prevent an auto-migration from being generated
    # when there are no changes to the schema
    # reference: http://alembic.zzzcomputing.com/en/latest/cookbook.html
    def process_revision_directives(context, revision, directives):
        if getattr(config.cmd_opts, 'autogenerate', False):
            script = directives[0]
            if script.upgrade_ops.is_empty():
                directives[:] = []
                logger.info('No changes in schema detected.')

    connectable = current_app.extensions['migrate'].db.get_engine()

    with connectable.connect() as connection:
        context.configure(
            connection=connection,
            target_metadata=target_metadata,
            process_revision_directives=process_revision_directives,
            **current_app.extensions['migrate'].configure_args
        )

        with context.begin_transaction():
            context.run_migrations()
```

Figure 3.1.1 run_migrate_online mode

3.1.2 using flask_sqlalchemy library to manipulate the database

The connection to database is established by SQLALCHEMY, which is another FLASK extension. To use it, we should first write corresponding configurations:

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///app.db'
```

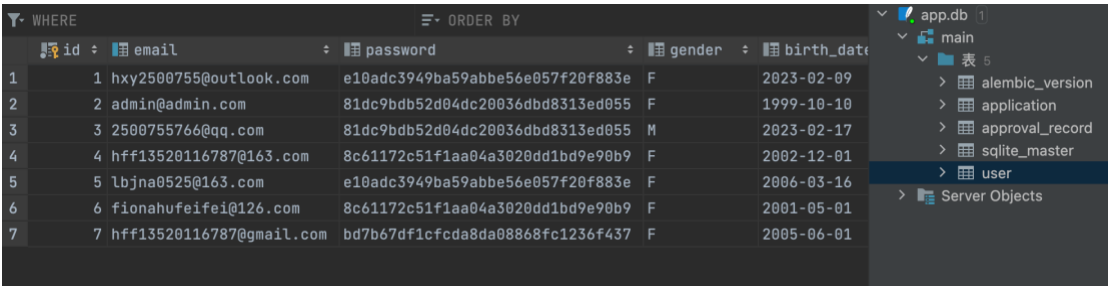
Figure 3.1.2.1 configuration corresponding to SQLALCHEMY

Then we could create an instance on this FLASK application instance and integrate them together. With these operations, we could manage the migration of database through Migrate instance.

```
db = SQLAlchemy(app)
migrate = Migrate(app, db)
```

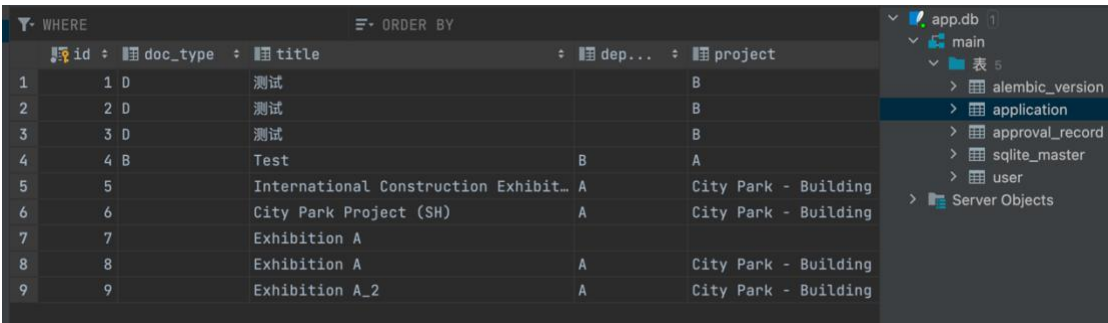
Figure 3.1.2.2 the migration of database through Migrate instance

When initializing the table structure of the database, we use the inherence class of `db.Model`, which indicates this model could be used as an interface for direct query. (Myers and Copeland 2015, p.71-71) See Criterion B 2.2 for the more detailed E-R model.



	id	email	password	gender	birth_date
1	1	hxy2500755@outlook.com	e10adc3949ba59abbe56e057f20f883e	F	2023-02-09
2	2	admin@admin.com	81dc9bdb52d04dc20036dbd8313ed055	F	1999-10-10
3	3	2500755766@qq.com	81dc9bdb52d04dc20036dbd8313ed055	M	2023-02-17
4	4	hff13520116787@163.com	8c61172c51f1aa04a3020dd1bd9e90b9	F	2002-12-01
5	5	lbjna0525@163.com	e10adc3949ba59abbe56e057f20f883e	F	2006-03-16
6	6	fionahufeifei@126.com	8c61172c51f1aa04a3020dd1bd9e90b9	F	2001-05-01
7	7	hff13520116787@gmail.com	bd7b67df1cfcda8da08868fc1236f437	F	2005-06-01

Figure 3.1.2.3 the "user" form in the database



	id	doc_type	title	dep...	project
1	1	D	测试		B
2	2	D	测试		B
3	3	D	测试		B
4	4	B	Test	B	A
5	5		International Construction Exhibit...	A	City Park - Building
6	6		City Park Project (SH)	A	City Park - Building
7	7		Exhibition A		
8	8		Exhibition A	A	City Park - Building
9	9		Exhibition A_2	A	City Park - Building

Figure 3.1.2.4 the "application" form in the database

3.2 the implementation of database model for each entity

Each entity would be realized as a database model, we take User as an example:

```

class User(db.Model):
    id = db.Column(db.Integer, autoincrement=True, primary_key=True)
    email = db.Column(db.String(128), unique=True)
    password = db.Column(db.String(32))
    name = db.Column(db.String(16))
    # gender
    gender = db.Column(db.String(8))
    # date of birth
    birth_date = db.Column(db.String(8))
    # department
    department = db.Column(db.String(16))
    # project list
    project_list = db.Column(db.String(255))
    # profile_path
    profile_path = db.Column(db.String(255))
    # role
    role = db.Column(db.String(16), default='employee')
    # superior_id
    superior_id = db.Column(db.Integer)
    # created_time
    created_time = db.Column(db.DateTime, default=datetime.datetime.now())

```

Figure 3.2.1 Database model – "User"

It realized as an inheritance of database model, which could be easily queried like:

```

users = User.query.filter(User.email == email)

```

Figure 3.2.2 queries the inheritance of database model

Another advantage of this solution is that queries through the API of database model would be automatically checked and extracted, avoiding SQL injection and other SQL attacking methods.

4. User account management

4.1 User sign-up

I use `request.form.get()` method to get input from the users when registering a new account. The inputted email will be search in the database to check whether it has been registered already. If it is an existing email in the database, the `sign_up()` function will return false. After the account is successfully registered, the program will automatically redirect to login page, which provide a convenience for the users.


```

# sign-up
@app.route('/sign_up', methods=['GET', 'POST'])
def sign_up():
    if request.method == 'POST':
        email = request.form.get('email')
        password = request.form.get('password')
        re_password = request.form.get('re_password')
        if not email or not password:
            return render_template('register.html', error='All input fields are required.')
        if password != re_password:
            return render_template('register.html', error='Confirm password must be the same as the password.')
        user = None
        # searching user in the database

        user = User.query.filter(User.email == email).all()

        if user:
            # if the email has been registered already, return false
            return render_template('register.html', error='The email address has been registered.')
        if user and user[0].status == 0:
            user[0].delete()

        # user sign up
        hl = hashlib.md5()
        hl.update(password.encode(encoding='utf-8'))

        db.session.add(User(email=email, password=hl.hexdigest()))
        db.session.commit()
        return redirect('/login')

```

Figure 4.1.1 user sign_up function

4.2 User log-in

The email and password inputted by the user will be accessed within the **login()** function. The validation of password refers to the comparison between the user-entered password and the password, which is stored as ciphertext, corresponding to the email address stored in the database. If the email address and the password are valid and correspond to each other, the user will be redirected to the manager interface or the employee interface depending on their role (**User.role**). On the contrary, if the email account or password is incorrect, the system will prompt "Username or password incorrect ".

```

@app.route('/login', methods=['GET', 'POST'])
def login():
    role = request.form.get('role')
    if request.method == 'POST':
        # access the email and password inputted in the frontend
        email = request.form.get('email')
        password = request.form.get('password')
        if not email or not password:
            return render_template('login.html', error='All input fields are required! ')
        # validate password
        hl = hashlib.md5()
        hl.update(password.encode(encoding='utf-8'))
        print(hl.hexdigest())
        print(email)
        print(role)
        user = None

        user = User.query.filter(User.email == email, User.password == hl.hexdigest(),
                                User.role == role).all()

    if user:
        # login success
        session['user'] = user[0].id
        session['user_role'] = role
        if role == 'manager':
            return redirect('/manager/index')
        else:
            return redirect('/employee/index')
    else:
        return render_template('login.html', error='The username or password is incorrect')

```

Figure 4.2.1 user login system

4.3 "Forget password" and E-mail verification process

The "Forget password" function includes: the user submits a reset password request, the system sends a random verification code, and the user resets the password. Before sending the E-mail to the user, a random verification code of fixed length needs to be generated.

```

# generate verification code
def get_random_code(code_length):
    code_source = '1234567890'
    code = ''
    for i in range(code_length):
        # randomly select a char
        # code +=choice(code_source)
        str = code_source[randrange(0, len(code_source) - 1)]
        code += str
    # return verification code
    return code

```

Figure 4.3.1 generating verification code

An email containing the verification code will be sent to the user. When sending an email, the

program will use the try and except method to return the sending status, which 1 represents sending successfully.

```
# sending email

def send_email_code(email, code):
    message = Message(
        subject='Verification System',
        recipients=[email],
        # sender='zzz2500755@foxmail.com',
        sender='fionahufeifei@126.com',
        # email content
        body='Your verification code is '+code
    )

    # sending email and return sending status
    # 1 represent sending successfully
    try:
        status = mail.send(message)
        print(status)
        return 1
    except Exception as e:
        raise e
        return 0
```

Figure 4.3.2 sending email and return sending status

4.4 using python decorator and flask session to implement login authentication

The first difficult point is the maintenance and authorization of the login state. This is mainly realized by session, a kind of data stored on server end. When a user posts a request, flask could store information from current context in session and combine it with current user.

The logged user's data and role (employee or employer) would be stored in session after they logged in successfully.

```
if user:

    # login success
    session['user'] = user[0].id
    session['user_role'] = role
    if role == 'manager':
        return redirect('/manager/index')
    else:
        return redirect('/employee/index')
else:
    return render_template('login.html', error='The username or password is incorrect')
```

Figure 4.4.1 user-password checking

To maintain the login state, each time the user conducts a request, we need to repeat the whole process of authorization. This would make huge redundant code using function. The solution I use is wrapper, a decorator that could be seen as the extension of other functions. I use it in the `check_login` function:

```
# log-in authentication
def check_login(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        user = session.get('user')
        role = session.get('user_role')
        if not user or not role:
            return redirect('/login') # if logged-in user invalid, jumped to logging page
        else:
            user = User.query.filter(User.id == user).all()
            if user:
                user = user[0].__dict__.copy()
                del user['_sa_instance_state']
                del user['password']
                user['created_time'] = user['created_time'].strftime("%Y-%m-%d %H:%M:%S")
                for attr, value in user.items():
                    if value is None:
                        user[attr] = ''

                request.user = user
            else:
                redirect('/login')

            return func() # if valid user logged-in, continue func()
    return wrapper
```

Figure 4.4.2 the use of wrapper

We could see that an `func()` has passed to the wrapper, it checks whether current user could match the data stored in session. If not matched, it would end the execution of `func()` and jump to logging page. But if matched, then it would just return `func()`, so that the original function would continue running. One advantage of this solution is its convenience. We just need to add a decorator where we want to check the logging state, such like:

```
@app.route('/employee/apply/submit', methods=['POST'])
@check_login
def submit_apply():
```

Figure 4.4.3 check the logging state by adding a decorator

And the session would be cleared when logging out.

```
# logout
@app.route('/logout', methods=['GET'])
def logout():
    # User log-off
    session.clear()
    return redirect('/login')
```

Figure 4.4.4 the `logout()` function

5. Implementation of GUI

5.1 Usage of existing tools

5.1.1 Vue.js 3

The frontend is implemented by Vue.js 3. Vue.js 3 offers a powerful and intuitive approach to building user interfaces. With its component-based architecture, I can create modular and reusable UI elements, enhancing code organization and maintainability. Vue.js 3's reactivity system enables me to efficiently handle data binding (Macrae 2006, p.26), ensuring seamless synchronization between the UI and underlying data.

```
function normalizeStyle(value) {
  if (isArray(value)) {
    const res = {};
    for (let i = 0; i < value.length; i++) {
      const item = value[i];
      const normalized = isString(item)
        ? parseStringStyle(item)
        : normalizeStyle(item);
      if (normalized) {
        for (const key in normalized) {
          res[key] = normalized[key];
        }
      }
    }
    return res;
  }
  else if (isString(value)) {
    return value;
  }
  else if (isObject(value)) {
    return value;
  }
}
```

Figure 5.1.1 function normalizeStyle() in Vue.js3

5.1.2 Element-UI Plus

To create an appealing and user-friendly interface for my project, I have chosen to integrate Element-UI Plus as the UI framework. Element-UI Plus is a powerful and versatile library that offers a wide range of pre-designed components, ensuring a consistent and professional look across the

application. With its responsive layout and extensive collection of UI elements such as buttons, forms, tables, and models, Element-UI Plus has enabled me to create interactive and visually appealing user interface.

```

/! Element Plus v2.2.17 */

(function (global, factory) {
  typeof exports === 'object' && typeof module !== 'undefined' ? factory(exports, require('vue')) :
  typeof define === 'function' && define.amd ? define(['exports', 'vue'], factory) :
  (global = typeof globalThis !== 'undefined' ? globalThis : global || self, factory(global.ElementPlus = {}, global.Vue));
})(this, (function (exports, vue) { 'use strict';

  var freeGlobal = typeof global == "object" && global && global.Object === Object && global;

  var freeSelf = typeof self == "object" && self && self.Object === Object && self;

  var root = freeGlobal || freeSelf || Function( args: "return this" )();

```

Figure 5.1.2 Element-UI Plus components

5.2 Calendar

The "Date of Expense(s)" entry that employees fill in when completing the reimbursement request form will be done through an interactive calendar. Employees can fill in the time by clicking on the corresponding date on the calendar.

```

</el-form-item>
<el-form-item
  label="Date of Expense(s)"
  :label-width="formLabelWidth"
  style="..."
>
  <el-col :span="11">
    <el-date-picker
      v-model="applyContent.applicationDate"
      type="date"
      format="YYYY-MM-DD"
      value-format="YYYY-MM-DD"
      placeholder="Pick a date"
    ></el-date-picker>
  </el-col>
</el-form-item>

```

Figure 5.2.1 the implementation of the interactive calendar

6. System Security Insurance

6.1 Database encryption

I use the `hashlib.md5()` encryption method to encrypt user passwords and store them in the database. (Zaki and Al-Humadi, 2020) md5 cannot be decrypted, which ensures the security of user passwords to a certain extent. I initialize an instance of `md5()`, then pass in the content to be encrypted via the `update()` method, and finally return the md5 value via the `hexdigest()` method of the md5 object.

```
if code == code_form:
    # password encryption
    hl = hashlib.md5()
    hl.update(password_form.encode(encoding='utf-8'))
    users.update({
        'password': hl.hexdigest()
    })
    db.session.commit()
```

Figure 6.1.1 using `hashlib.md5()` for password encryption when reset the password

6.2 Data Verification—Password Double-entry

When a user registers a new account or resets the password, the system requires the password to be entered twice. If the user does not enter exactly the same content twice, the system will pop "Confirm password consistency". This ensures a certain degree of data security caused by users entering their passwords incorrectly.

```
code_form = request.form.get('code')
email_form = request.form.get('email')
password_form = request.form.get('password')
re_password = request.form.get('re_password')
if not code_form or not email_form or not password_form or not re_password:
    return render_template('forget_pwd.html', error='All input fields are required.')
if password_form != re_password:
    return render_template('forget_pwd.html', error='Confirm password consistency')
users = User.query.filter(User.email == email)
```

Figure 6.2.1 the new password needs to be entered twice when reset the password

6.3 Encrypted display when keying in a password

The password is displayed encrypted when typed by the user. The `sr-only` method will hide the text so that the inputted password will be invisible. This keeps the user's information secure.

```
<!-- 'sr-only' will hide the text : 'Password'. So, 'Password' will be invisible -->
<label for="inputPassword" class="sr-only"></label>
<input
    type="password"
    id="inputPassword"
    class="form-control mb-2"
    placeholder="Password"
    name="password"
    required
>
<div class="checkbox mb-3">
```

Figure 6.3.1 using sr-only to let the inputted password invisible

Word Count: 1093

Reference list

- Degott, C., Borges, N.P. and Zeller, A. (2019). Learning user interface element interactions. *International Symposium on Software Testing and Analysis*. doi:<https://doi.org/10.1145/3293882.3330569>.
- eForms (2022). *Free Employee Reimbursement Form - PDF / Word / eForms – Free Fillable Forms*. [online] eforms.com. Available at: <https://eforms.com/employee/reimbursement/>.
- Gaffney, K.P., Prammer, M., Brasfield, L., Hipp, D.R., Kennedy, D. and Patel, J.M. (2022). SQLite: past, present, and future. *Proceedings of the VLDB Endowment*, 15(12), pp.3535–3547. doi:<https://doi.org/10.14778/3554821.3554842>.
- Gaspar, D. and Stouffer, J. (2018). *Mastering Flask Web Development*. Packt Publishing Ltd.
- Grinberg, M. (2018). *Flask Web Development*. O’reilly Media, Incorporated.
- Kunal Relan and Springerlink (Online Service (2019). *Building REST APIs with Flask : Create Python Web Services with MySQL*. Berkeley, Ca: Apress.
- Macrae, C. (2018). *Vue.js: Up and Running*. ‘O’Reilly Media, Inc.’
- Myers, J. and Copeland, R. (2015). *Essential SQLAlchemy*. ‘O’Reilly Media, Inc.’
- Saito, Y., Bershad, B.N. and Levy, H.M. (1999). Manageability, availability and performance in Porcupine. *Symposium on Operating Systems Principles*. doi:<https://doi.org/10.1145/319151.319152>.
- Thodge, S. (2018). *Cloud Analytics with Google Cloud Platform : an end-to-end guide to processing and analyzing big data using Google Cloud Platform*. Birmingham: Packt Publishing.
- Zaki, W. and Al-Humadi, M. (2020). *CRYPTOGRAPHY IN CLOUD COMPUTING FOR DATA SECURITY AND NETWORK SECURITY*. [online] Available at: <https://faculty.uobasrah.edu.iq/uploads/publications/1611727426.pdf> [Accessed 27 Jun. 2023].

