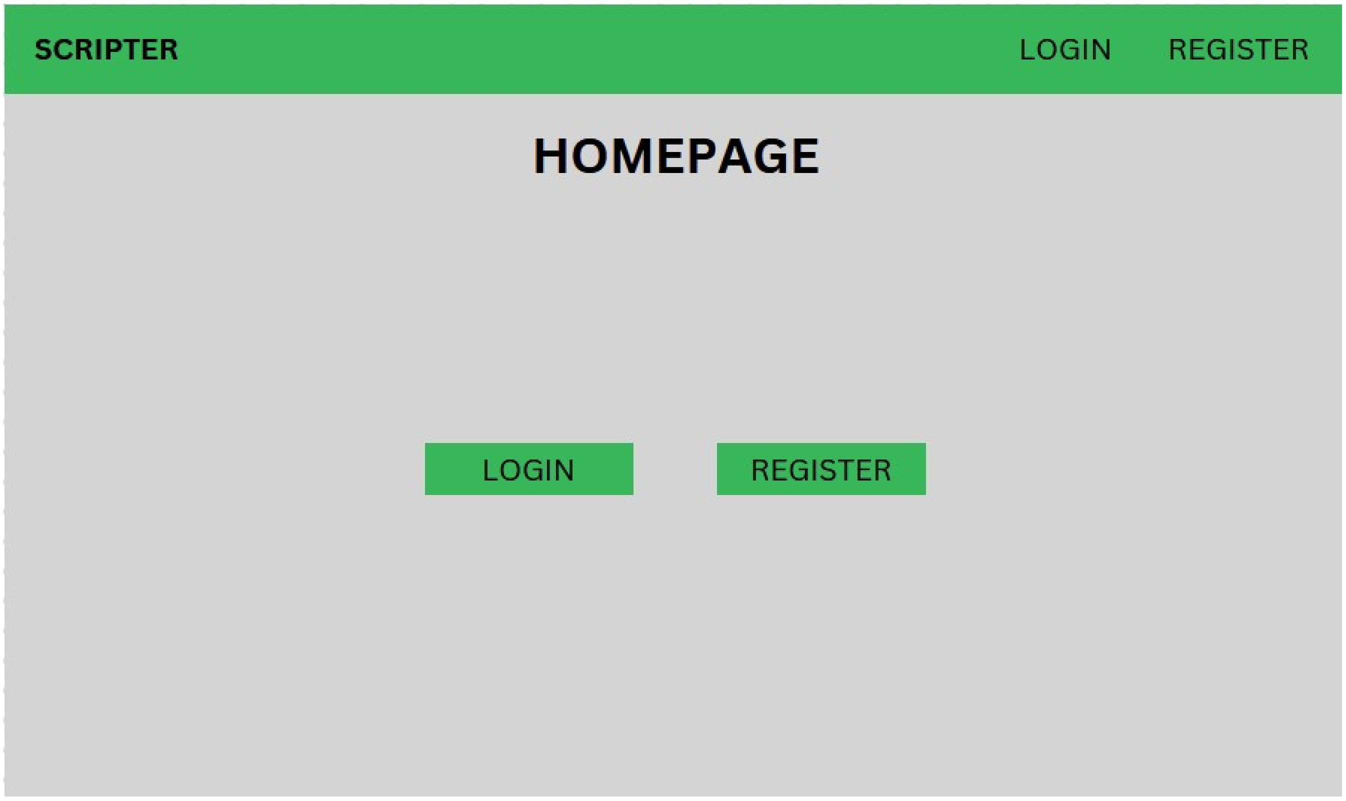
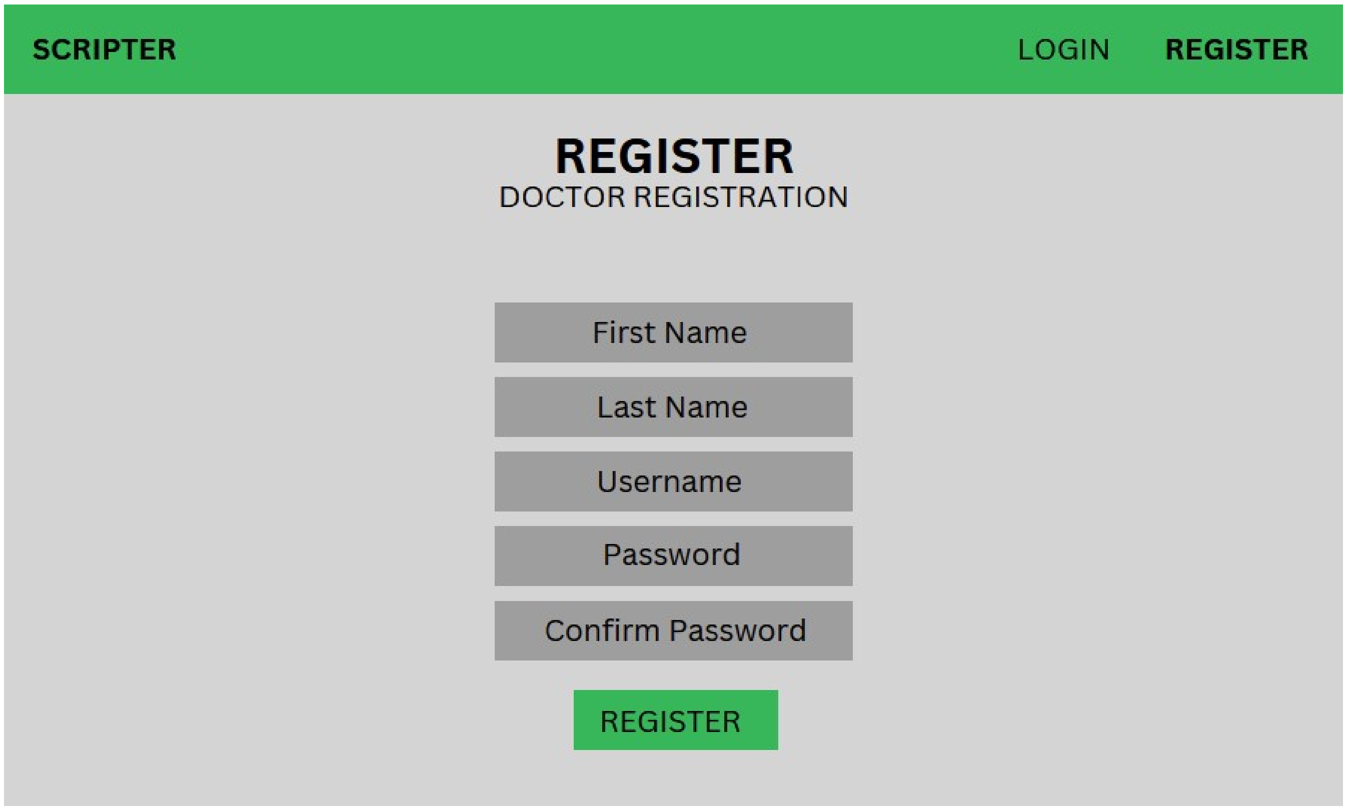
Implementation and Analysis

Summary

The project is a web application that uses symmetric and asymmetric encryption, along with other industry standard communication and security protocols to transfer prescriptions from doctors to pharmacies. In this submission, the methods used in various facets of the project are discussed. Most of the features are coded in a modular fashion and all elements stand on their own, for now. The final steps of the project development will be polishing the UI/UX and tying all the modules together.

Front-End





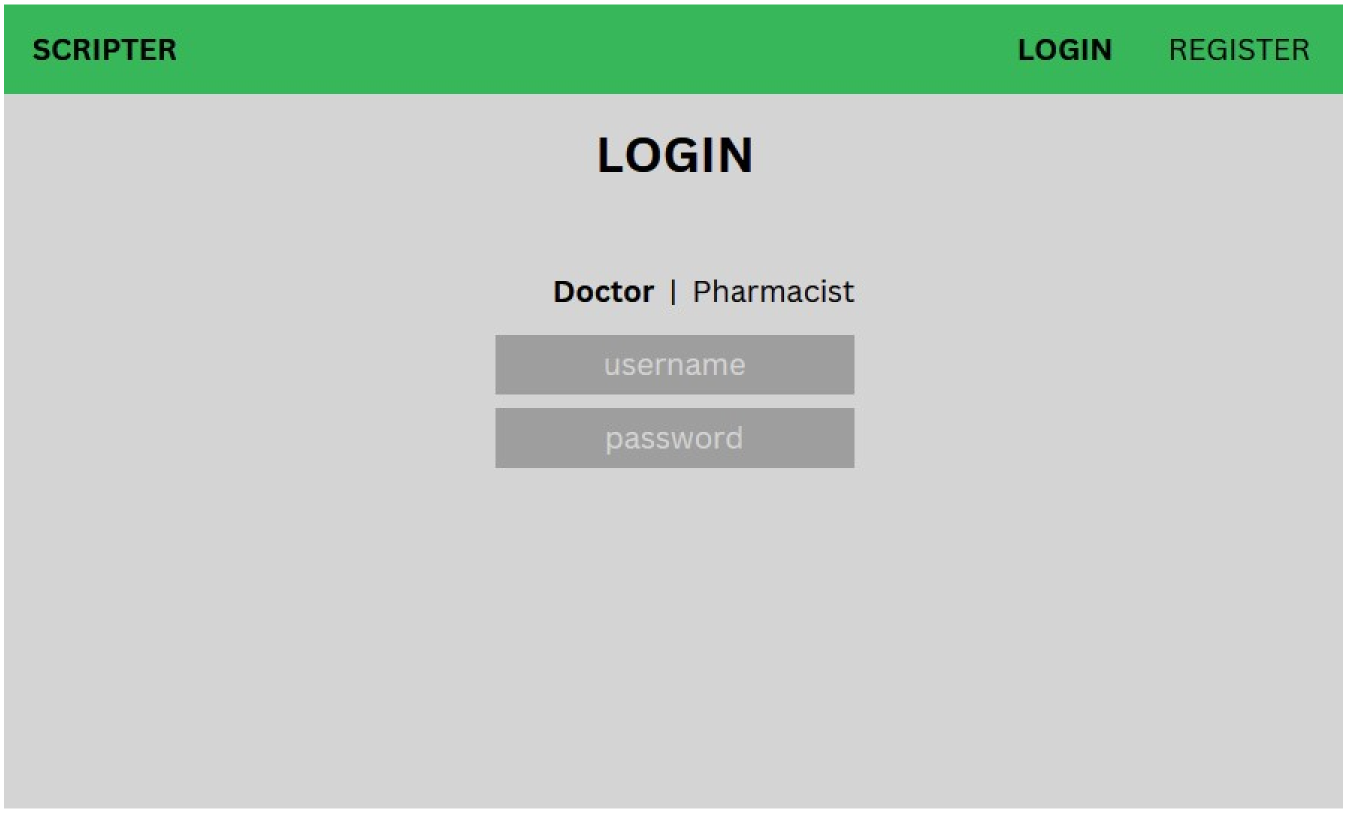
SCRIPTER

LOGIN

REGISTER

REGISTER

DOCTOR REGISTRATION



The web application is built using NodeJS and Express. CSS and some HTML are used. There are two types of users: Doctors and Pharmacists. Depending on the type of user, the pages you can access differ. Here is a breakdown of all pages:
Login (Doctors & Pharmacists)
Register as a new user (Doctors & Pharmacists)
File Upload (Doctors)
Prescription List (Doctors & Pharmacists; varies depending on user)

Login Page

The login page will contain a radio button that lets you choose what type of user you are. This is included as the registration database includes a 'user type' indicator which needs to match with the user input to enable a successful login.

Register

The Register page allows new users to join the service. For now, it simply uses a radio button to denote what type of user you want to register as. Generally, you would need to provide some sort of identification as to whether you're a doctor or a pharmacist, but that is not within the scope of this project.

File Upload

Doctors will be able to upload files and send it to a selected recipient. This page contains the core of the application. When a user selects a file to be sent to a specific recipient, SHA-256 is used to hash the file. This is then signed using the sender's private RSA key. A hash of the file and the original file key are sent to the recipient who will be able to verify the document.

Prescription List

Pharmacists will be able to see the prescriptions that they receive placed here. A button next to the file will allow the user to verify the incoming document. Once the file is verified, they will be able to download it. This page will contain the file name, a checksum to verify file integrity and a verify button that triggers a back-end script that uses DS verification protocols to ensure the document was signed by the intended sender. Once the file is verified, an option to download the file will be accessible.

Middleware
NPM (Node Package Manager) is used to install and manage all the packages used.

Multer: This is a file-handling module that allows the users to upload multiple types of files. For this project, we are using the module's built in features to restrict what types of files can be uploaded. On top of that, client-side input validation methods are used for file naming conventions to ensure both user ease and security concerns can be met. Multer does not do file type approval on its own, but allows the user to create functions that can handle these concerns. This is implemented in the application as follows:

```
app.post('/upload', upload.single('file'), (req, res, next) => {
    // Handle the uploaded file
    const whitelist = 'application/pdf';
    const meta = await FileType.fromFile(req.file.path)

    if (!whitelist.includes(meta.mime)) {
        return next(new Error('file is not allowed'))
    }
    res.json({
        file: req.file,
        body: req.body,
    })
    })
```

Python-Shell: Python is used in the project for RSA key pair creation and digital signature purposes. In order to run python scripts within the application and retrieve results, Python-Shell module is used. In this project, the RSA key pairs are created when a specific type of user registers and their password is used as passphrase for the private key. These are input into the python script as arguments and Python-Shell facilitates such functions.

BcryptJS: Passwords are stored as hashes in the database. When an user wants to log into the service, the hash of the entered password is compared to the hash stored in the database. Bcrypt enables the hashing of a registered password and the subsequent verification.

Registration:

```
app.post('/registered', function (req,res) {
        const saltRounds = 10;
        const plainPassword = req.body.password;
        // function that initiates creation of RSA key pair
        bcrypt.hash(plainPassword, saltRounds, function(err,
hashedPassword) {
        // Store hashed password in your database.
        ...
         // SQL query setup that takes input from user, verifies that
the query matches the structure of the database, then submits query
        ...
    });
```

Subsequent Login:

```
app.post('/loggedin', function (req,res) {

        let hashedPassword = "SELECT password FROM authentication WHERE
username = (?)";
        let username = req.body.username;
        db.query(hashedPassword, username, (err, result1) => {
            if (err) {
                return console.error(err.message);
            }
```

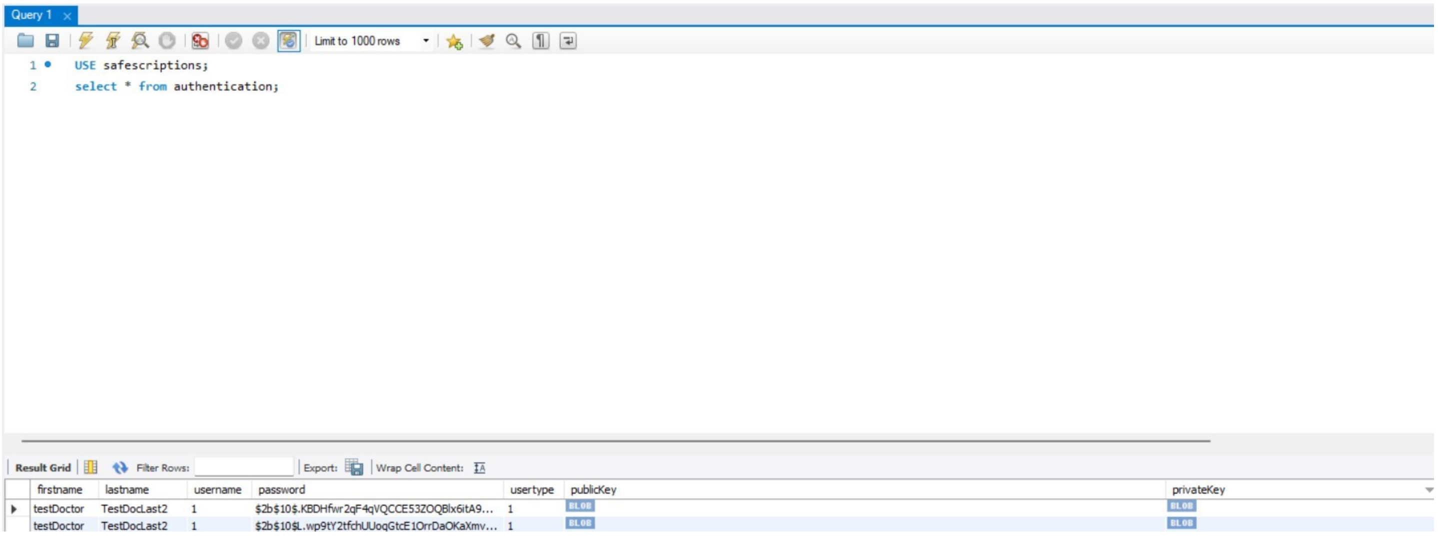
```
                else {
                    var finalpwd = result1[0].password;
                    bcrypt.compare(req.body.password, finalpwd,
function(err,result2){
                        if (err) {
                            console.error(err.message)
                        }
                        else if (result2 == true) {
                            req.session.userId = req.body.username;
                            res.redirect('./');
                            console.log("Logged In")
                        }
                        else {
                            console.log("Your password is wrong");
                        }
                    })
                    }
            });
        })
```

Express-Validator: This module deals with input validation concerns. Certain input pathways from the user can be exploited if not monitored properly. Concerns such as Cross Site Scripting (XSS) and SQL injection exist, and since the project deals with a database it is important to address these concerns. Input validation is the first step in this.

PyCryptodome: This is a package for Python's Crypto library. This package allows us to create RSA key pairs and also use PKCS#1 PSS, a probabilistic digital signature scheme based on RSA. A working example of the core of this project can be found below.

Back End

The database is managed by MySQL. The configuration for the database is as follows:
1. First Name. 2. Last Name. 3. Username. 4. Password. 5. User Type. 6. Public Key. 7. Private Key.



```sql
USE safescriptions;
select * from authentication;
```

firstname	lastname	username	password	usertype	publicKey	privateKey
testDoctor	TestDocLast2	1	$2b$10$.KBDHfwr2qF4qVQCCE53ZOQBlx6itA9...	1	BLOB	BLOB
testDoctor	TestDocLast2	1	$2b$10$L.wp9tY2tfchUUoqGtcE1OrrDaOKaXmv...	1	BLOB	BLOB

All elements of the database are NOT NULL and username needs to be unique. Public and Private keys are hashed and stored as blobs. Private Key is stored in an encrypted form and requires the user to enter their password once more to be decrypted and used for signing. There are two types of users: Doctors (1) and Pharmacists (0). Only Doctors will be able to create key pairs. Pharmacists will be able to lookup the public key of all doctors through a separate linked database. This is not a front-end feature. The secondary database will only contain the name of the user, their ID and their public key. This is in order to protect the private key from being accessed by unauthorised users.

Overview of the core of the project
Workflow
Sender side signing:

1. User selects file to be uploaded to server.
2. User then selects specific file to be sent to specific recipient
3. Once selected, user signs file using their private key
4. Once signed, user sends file to recipient
5. In the backend, the application hashes the file using SHA-256.
6. The SHA-256 hash is signed using the sender's private key
7. This signed hash and the original file are sent to the recipient

```python
# sign.py
from Crypto.Signature import pss
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
```

```
file_location = 'prescription.pdf'
file_in = open(file_location, 'rb')
message = file_in.read()
file_in.close()

key = RSA.import_key(open('myprivatekey1.der', 'rb').read())
h = SHA256.new(message)
signature = pss.new(key).sign(h)

file_name = 'signature.txt'
file_out = open(file_name, 'wb')
file_out.write(signature)
file_out.close()
```

The code above shows steps 5, 6 and 7. Once a PDF is set up for signing, it is first read as bytes and hashed using SHA256. Then the signature is stored in a txt file. This is for illustration purposes. When the program is run, this will be stored as a blob in the database. One other feature that is considered for implementation is the checksum that is output as a hash here. This can be used to verify if the file has been corrupted during storage or transmission. Although it may be unnecessary if the file is verified successfully, it will be useful to determine whether the validation failed due to wrong signature or if the file was corrupted during storage.

Recipient side verification

1. Recipient gets original file and signed hash
2. Recipient hashes the original file using SHA-256
3. They then verify this hash using the sender's public key.
4. The signed hash is then compared to the recipient's hash and verified.
5. If verification succeeds the user is able to download the file.

```
# verify.py
from Crypto.Signature import pss
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA

file_location = 'prescription.pdf'
file_in = open(file_location, 'rb')
message = file_in.read()
```

```
file_in.close()

sign_file = 'signature.txt'
file_in = open(sign_file, 'rb')
signature = file_in.read()
file_in.close()

key = RSA.import_key(open('mypublickey1.der', 'rb').read())
h = SHA256.new(message)
verifier = pss.new(key)

try:
    verifier.verify(h, signature)
    print("The signature is authentic.")

except (ValueError):
    print("The signature is not authentic.")
```

The code excerpt above shows the core workflow on the recipient's side. Here the filenames are hard-coded into the script for illustration purposes, but during production these will be embedded as a JSON message along with the file transfer.

Terminal:

```
PS D:\CSLevel6\FP-Supplementary> python verify.py
The signature is authentic.
```

Hosting and Storage

For now, the project is hosted on a local computer. For testing purposes this is sufficient. But we run into problems when the application goes to production. For one, the storage for these files is simply a partition in the hard drive of my laptop. Although this works for now, the storage server needs to be live at all times and configuring it every time the device is restarted isn't feasible. Virtual Machines can be used as servers by using VPNs to connect servers and hosts. But this is nothing more than a sandbox. Another option is to use the Goldsmiths' Igor network to host the application. Although there are some limitations in regards to speed and efficiency, this might be a feasible option. Microcomputers such as Raspberry Pi's were considered as storage solution but we run into the same problem as VMs. Moreover, a Static IP is pertinent for servers and that is not something I can acquire now. For now, the most feasible options are a makeshift network in the Goldsmiths Igor platform or paying for hosting services such as IONOS or GoDaddy.