SUPERVISOR: AHMED HANIF

NARWAL AUTH API DELIVERABLE 11

MUHAMMAD NASEEM
SAMI NAEEM
KYNAT MANSHA

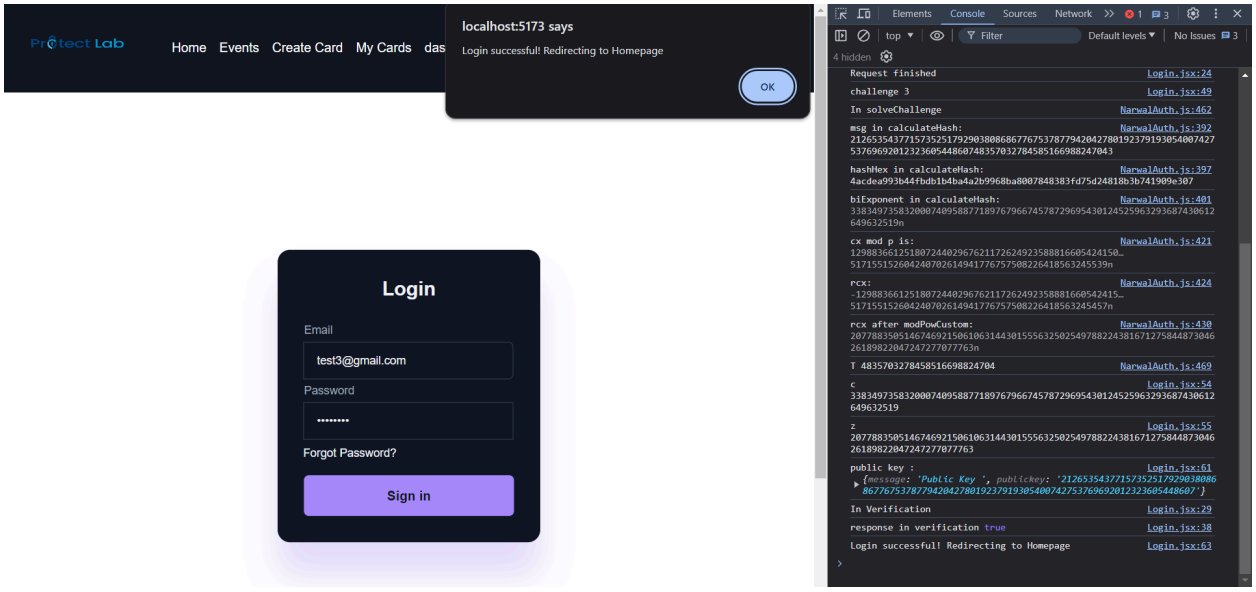
API INTEGRATION

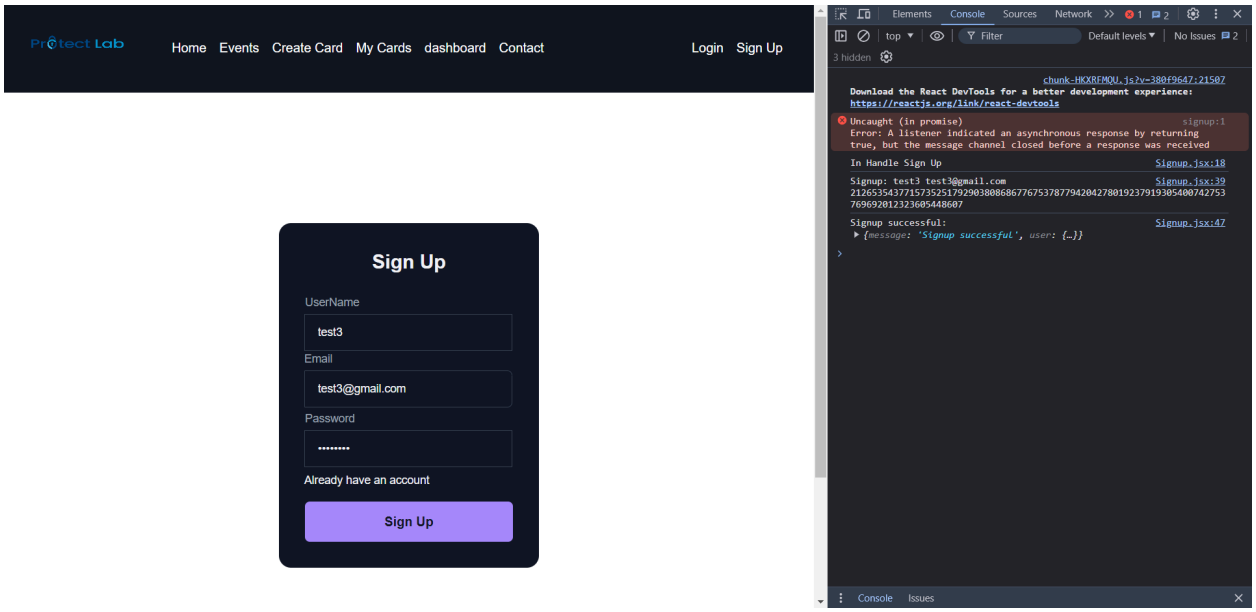
Resolved all previous issues with api integration.

- issues with Login verification

Code update:

- Used ‘Spawn’ library that helps integrate python development into a javascript file.
- Used python to integrate the intensive calculations functionality.
- ‘BigInt’, which was causing issues with our javascript file, worked flawlessly with python.





Express and Middleware Setup:

- **Express**: Initializes an Express application.
- **cors**: Middleware to enable CORS (Cross-Origin Resource Sharing) for handling requests from different origins.
- **express.json()**: Middleware to parse incoming JSON payloads.
- **express.urlencoded({ extended: true })**: Middleware to parse incoming URL-encoded payloads.

```
const express = require('express');
const cors = require('cors');
const { spawn } = require('child_process');
const app = express();

app.use(cors());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

Helper function for running python scripts:

- runPythonScript: Executes a Python script asynchronously using **spawn** from **child_process**.
- Parameters: **scriptPath** (path to Python script), **args** (array of arguments for the Python script), **callback** (function called after script execution).
- Functionality: Captures script output (**stdout**), logs errors (**stderr**), and calls the provided callback with data or error messages.

```
// Function to run Python script asynchronously
function runPythonScript(scriptPath, args, callback) {
  const pythonProcess = spawn('python', [scriptPath].concat(args));

  let data = '';
  pythonProcess.stdout.on('data', (chunk) => {
    data += chunk.toString(); // Collect data from Python script
  });

  pythonProcess.stderr.on('data', (error) => {
    console.error(`stderr: ${error}`);
  });

  pythonProcess.on('close', (code) => {
```

```
    if (code !== 0) {
      console.log(`Python script exited with code ${code}`);
      callback(`Error: Script exited with code ${code}`, null);
    } else {
      console.log('Python script executed successfully');
      callback(null, data);
    }
  });
}
```

Initialization and Challenge Storage:

- challenges: **Map** to store generated challenges associated with email addresses.
- Console log: Indicates server start for monitoring purposes.

```
const challenges = new Map();
console.log("narwalauth api.js is starting...");
```

Endpoint: Generate Challenge:

- Endpoint **/generate-challenge**: Handles POST requests to generate and send a challenge
- Request Body: Expects **email** field.
- Validation: Checks if **email** is provided; returns error if not.
- Python Script Invocation: Calls runPythonScript to execute **zkp_operations.py** with **generate_challenge** argument.
- Response: Stores generated challenge in **challenges** map and sends it as JSON response

```
app.post('/generate-challenge', (req, res) => {
  console.log("In generate-challenge api");
  const { email } = req.body;
  if (!email) {
    return res.status(400).json({ error: 'Email is required' });
  }

  // Call Python script to generate challenge
  runPythonScript('zkp_operations.py', ['generate_challenge'], (err,
results) => {
    if (err) throw err;
    const challenge = results[0]; // Assuming results is an array
```

```
containing challenge
    challenges.set(email, challenge);
    res.json({ challenge: challenge.toString() });
  });
});
```

Endpoint: Verify Authentication:

- Endpoint **/verify**: Handles POST requests to verify authentication data (**publicKey**, **c**, **z**)
- Validation: Ensures required fields (**email**, **publicKey**, **c**, **z**) are present; returns error if any are missing.
- Python Script Invocation: Calls **runPythonScript** to execute **zkp_operations.py** with **verify** argument and authentication data.
- Response Handling: Parses output from Python script to determine authentication success or failure based on the first element of returned data.

```
app.post('/verify', (req, res) => {
  const { email, publicKey, c, z } = req.body;

  const challenge = challenges.get(email);
  if (!challenge || !email || !publicKey || !c || !z) {
    return res.status(400).json({ error: 'Limited Information' });
  }

  console.log(req.body);
  console.log("challenge", challenge.toString());

  // Call Python script to verify authentication
  runPythonScript('zkp_operations.py', ['verify', publicKey, c, z,
challenge.toString()], (err, results) => {
    if (err) throw err;

    // Process output from Python script
    const rawOutput = results.split('python verification
script')[1].trim();
    console.log('Processed output from Python script:', rawOutput);

    try {
      // Assuming Python script returns JSON-like output
      const resultsArray = rawOutput.slice(1, -1).split(',').map(item =>
item.trim());
      const firstElement = parseInt(resultsArray[0], 10);
```

```
      // Determine success based on Python script output
      if (firstElement === 1) {
        console.log('Authentication success');
        res.json({ success: true });
      } else {
        console.log('Authentication failed');
        res.json({ success: false });
      }
    } catch (parseError) {
      console.error('Error parsing JSON:', parseError);
      res.status(500).json({ error: 'Invalid response from verification
process' });
    }
  });
});
```

Server Initialization:

- Server Initialization: Starts the Express server on specified port (**3001** by default).
- Console log: Logs server start-up message for monitoring.

```
const port = process.env.PORT || 3001;
app.listen(port, () => {
  console.log(`API running at http://localhost:${port}`);
});
```

Our ZKP_Operations.py file

generate_challenge()

- Function: Generates a random challenge used for authentication.
- Returns: Random integer challenge between 100 and 999.

```
def generate_challenge():
    """
    Generates a random challenge integer between 100 and 999.
    Returns:
        int: Randomly generated challenge.
    """
    return randint(100, 999)
```

compute_hash(data)

- Function: Calculates SHA-256 hash of input string data.
- Args: data (str) - Data to be hashed.
- Returns: Hashed value as an integer.

```
def compute_hash(data):
    """
    Computes SHA-256 hash of input data.

    Args:
        data (str): Data to be hashed.

    Returns:
        int: Hashed value as an integer.
    """
    hash_object = hashlib.sha256(data.encode())
    hash_hex = hash_object.hexdigest()
    hashed_password_decimal = int(hash_hex, 16)
    return hashed_password_decimal
```

verify(public_key, c, z, challenge)

- Function: Verifies authentication data against a challenge using zero-knowledge proof.
- Args:
  - **public_key** (str): Public key for verification.
  - **c** (str): Hash value received from client.
  - **z** (str): Exponent value from client.
  - **challenge** (str): Challenge value received from client.
- Returns: Tuple indicating verification success (1) or failure (0), computed **Tnot** value, and computed hash value.

```
def verify(public_key, c, z, challenge):
    """
    Verifies the authentication data against a challenge.

    Args:
        public_key (str): Public key for verification.
        c (str): Hash value received from client.
        z (str): Exponent value from client.
        challenge (str): Challenge value received from client.

    Returns:
        tuple: First element is 1 for success, 0 for failure.
               Second element is Tnot value computed during verification.
```

```
                Third element is computed hash value.
    """
    Y = int(public_key)
    c = int(c)
    z = int(z)
    g = 2  # generator
    p =
407407195266897217253689137681875632210293678733187250127228089870876259952
6673412366794779  # Prime number

    Yc = pow(Y, c, p)
    Tnot = pow((Yc * pow(2, z, p)), 1, p)

    hash_input = f"{Y}{Tnot}{challenge}"
    computed_hash = compute_hash(hash_input)

    if c == computed_hash:
        return 1, Tnot, computed_hash
    else:
        return 0, Tnot, computed_hash
```

Main Execution (if __name__ == '__main__')

- Main Execution: Checks command-line arguments to decide which operation to perform (generate_challenge or verify).
- Usage
    - ❖ For generate_challenge: Prints a randomly generated challenge.
    - ❖ For verify: Prints the result of verification (success or failure), Tnot value, and computed hash value.

```
if __name__ == '__main__':
    operation = sys.argv[1]

    if operation == "generate_challenge":
        print(generate_challenge())
        sys.stdout.flush()
    elif operation == "verify":
        public_key, c, z, challenge = sys.argv[2:]
        print(verify(public_key, c, z, challenge))
        sys.stdout.flush()
```