The Expose Part of the Project

1. Databases

Setting up the databases is a prerequisite for the rest of the steps.
Below are the databases used and ways to set them up:

1. Neo4j
   - ➔ Create an account on the Neo4j Platform
   - ➔ Create a database
   - ➔ Get the connection string
2. MongoDB
   - ➔ Create an account in MongoDB Atlas
   - ➔ Create a database and a collection inside of it
   - ➔ Get the connection string
3. Redis
   - ➔ Create an account in Redis Labs
   - ➔ Create a database (chose Redis Stack option to get all the additional features)
   - ➔ Get the connection string

2. Email Server

- Prerequisite:
  - Have an Sendgrid account and API key
  - Have npm and node installed
  - Webhook container is hosted
  - Create a Neo4j DB
  - Host the Neo4j DB
- Steps:

Ran “npm init” and created the package.json. Added the dependencies for express, nodemailer, swagger, dotenv, jsonwebtoken and azure. Created the app.js and added a route that has to send an email. Made it so the route accepts 2 emails(sender and receiver emails). Setup nodemailer(from their documentation) with smtp. Signed up for an online service that sends these emails out(Sendgrid), since gmail now blocks third-party apps to connect directly to emails. Replace auth details with the Sendgrid auth details. Added neo4j-driver and a neo4j-node-ogm to package.json as to create the invite and the user profile’s. First I check if they exist within our neo4j db. If not I add them(failsafe if the contact between registration and my service is impossible), if they do exist I add them to constants. Then I check if an invite already exists, if it does then I just send a message that the user was already invited. If it doesn’t I add them to the database(2 users and invite) and then I send the email with a link in it. This link is to our route for confirmation. There I create a

webtoken with the 2 user emails and invite. I connect to the container for our webhook and get the url of the integrating group. Then I redirect with the token in the url.

3. Update Server

- Prerequisite:
    - Have an Azure account
    - Have npm and node installed
    - Create an Azure storage container
    - Create the MongoDB database
    - Hosted MongoDB
- Steps:

Ran “npm init” and created the package.json. Added the dependencies for express, multer, twilio, mongoose, swagger, dotenv, jsonwebtoken and azure. Sign up for twilio to send sms messages. Created the app.js and added a route that updates profile information for a user. Added the mongoose models and connection to MongoDb. In the route I find the user and then update his profile information and save it to the database I also send a sms message to the old phone. If the user doesn’t exists I send an error response. I create a middleware which takes the file that is being uploaded and saves it to Azure storage. As a successful response I send the updated user info with a link to the newly uploaded picture. Added service bus function/listener which waits to receive messages from auth to create new users in the database.

4. RSS server

- Prerequisite:
    - Have Azure Extension for Vs code installed
    - Have an Azure account
    - Have npm and node installed
    - Create an Azure storage container
- Steps:

Used Azure extension for VS Code to create a local function. Added the dependencies xml, rss, xml-js, dotenv and azure. I create a function fist which generates the RSS feed file and added items to its stream which are ordered by date. Then I refactor the function to get RSS file from Azure storage and add items to that feed. After that I upload the file to Azure storage. Add a function which converts the string/xml data to json and send that as a response from the function.

5. CDN

- Prerequisite:
    - Azure account + subscription
    - Logo created in jpg, png format
- Steps:

1. We create the resource group using the System in. subscription named cdn-logo
2. We create Storage account named “silogogoat”
3. In the Storage account, we created a Container called “logo” and we uploaded a jpg logo picture file.
4. We have the access level of Container logo set to anonymous
5. Lets add the Azure CDN
6. In the Storage account, create a new CDN profile, name it.
7. By adding the end path to the generated url we should be able to get the logo : https://silogogoat.blob.core.windows.net/**logo/silogo.png**

6. Authentication Path

- Prerequisite:
    - Have npm and node installed
    - Azure account + subscription
    - Have MongoDB database
- Steps:

1. Created folder where all the files will be located.

2. Ran “npm init” and added the dependencies for express, mongoose, swagger, dotenv, passport, jsonwebtoken and azure.

3.Created a new collection in MongoDB where the user data will be saved

4.Create the “User” model (with “email”, “name”, “hash” and “salt” fields) using mongoose in order to interact with MongoDB.

5. Created the app.js and added code to create the express server and connect to MongoDB through mongoose.

6. Created the users.js file that will contain the routes for the user (such as /register and /login). Added implementation to save the user in MongoDB through the “register” path/endpoint and find the user in the “login” path.

7. Create the utils.js file with functions to hash the user’s password using salt and a secret string and check if the provided password corresponds with the hashed version.

8. Instead of using a secret string while generating the hash from the user's password, decided to use public and private keys. So created the “generatedKeyPair.js” file to generate these keys.

9. Used the file to generate keys and started using the private key when hashing the user’s password.

10. Enhanced the “register” and “login” endpoints by using the functions to hash and check user’s password and generate a JWT token for authentication.

11. Decided to use the passport-jwt middleware for user authentication. In the “passport.js” file integrated with the passport’s “done” function by passing either the user (if it’s found in the database) or error.

12. Tested that everything works correctly with Postman.

13. Created documentation with Swagger.

14. Decided to use the Azure Service Bus to send user information to other services when the user is registering. In azure, created a Service bus resource with a topic and 2 subscribers. In the “app.js” file created a function that would get triggered when the “register” endpoint is hit and user data gets sent to the topic which will be retrieved by other services asynchronously.

7. Products Path

- Prerequisite:
    - Have npm and node installed
    - Azure account + subscription
    - Have Redis database
- Steps:

1. Created folder where all the files will be located.

2. Ran “npm init” and added the dependencies for apollo-server, sequelize, sqlite3, redis-om, dotenv, jsonwebtoken and azure.

3. Created the server.js file where the GraphQL server will be. Inside of it instantiated an ApolloSever object that will run the run server.

4. Created a schema folder and inside of it opened two files – type-defs.js there the type definitions for GraphQL schema for our “Product” entity will be, as well as the resolvers.js file that would be responsible for populating the data for the “Product” fields. After looking at the requirements for the products.db file, the “Product” entity with all the fields and data types was specified.

5. Created a db.js file. Using the sequelize ORM, defined the classes and fields needed to map the SQLite tables to js objects – “Product”, “Product_Image”, “Product_Additional_Info”. Added code to connect to the products.db file and tested that data from tables gets imported into js objects.

6. Implemented the “Query” part in resolvers.js file to resolve the Graphql schema fields, meaning added code to provide corresponding data from products.db using sequelize, when the fields get called. The client can get all the products (with their images and additional info) or a specific product. Tested the GraphQL server through the ApolloServer Explorer to check that everything is working fine.

7. Add the Redis cache layer, so that the data from the products.db file gets loaded only once and the get saved into redis database, so that future requests for the data will take less time. Using the redis-om library, in the redis.js file created “Product”, “Product_Image” classes in order to model the js objects into Redis. Added some functions to upload/remove the data to Redis database. Changed the resolvers.js file so that the data gets pulled from redis instead of the local products.db file.

8. Added additional query options to the resolvers.js so that the client can also search for products by name, filter by category or price and both (search for specific products by name that are also in a specific price range). Implemented some functions in the redis.js to use Redis Search functionality and do products filtering on the database side (not in the graphql server, but redis db).

9. Created the azure_storage.js in order to integrate with the azure storage, since the FTP server saves the products.db file in Azure Storage. Added code to retrieve the file and save it locally.

10. Updated the server.js file to stop using the local products.db file but instead call Azure and get the file from Storage. Added a cron job to call Azure once every 12 hours and check if the file was updated and in case it was, then download it. Tested that the GraphQL server serves the data from products.db file pulled from Azure

11. Added authentication to the GraphQL server by adding a function that checks if the provided JWT token (contained in the header of the request) is valid. This way only registered/logged in users can retrieve the desired data.

8. Subscribe webhook

- Prerequisite:
    - Have Azure account + subscription
    - Azure extension for VS Code
    - Have npm and node installed
    - Create an Azure storage container

- Steps:

1. Created folder where all the files will be located.

2. Used Azure extension for VS Code to create a local function

3. Ran “npm init” and added the azure-storage dependency.

4. I have set up the main function that would receive incoming requests and would send back a response.

5. Then I created a function that would call Azure Storage in order to retrieve or upload data to the remote file.

6. Added functionality to the main function so that it processes the received data from the request and saves it in the file located in azure storage through the helper function.

7. Added error handling and checking for the correct parameters.

9. FTP server

- Prerequisite:
    - Have npm and node installed
    - Azure account + subscription
    - Packages: ftpd
- Steps:

1.Initiate ftpd server.

2. Return the root folder

3. Check for https certificates if present, if not use http

4. Check username and password to coincide with .env file variables

5. Define push_to_blob to be executed on command “STOR” (uploading dile to root directory).

6. Define functionality for uploading to azure:
    6.1 Connect to azure account.
    6.2 Create container
    6.3 Access the container with azure credentials.
    6.4 Create a blob in container
    6.5 Add the file to blob

10. Websockets

- Prerequisite:
    - Have npm and node installed
    - Packages: socket.io
- Steps:

1. Create an instance of socket.io on port 3000
2. Define on connection and on disconnect functionality
3. Create socket rooms for each friend you have
4. Keep track of time since “on connect” until “on disconnect”
5. Send an event with “time spent online” to all friends
6. Each person has N nr of rooms as N nr of friends.
7. Friends who were not online don’t receive messages displayed in the room.
8. Display the list of online friends.

11. Web Scraper & Cron Job

- Prerequisite:
    - Packages needed: requests, BeatifulSoup, Pandas
    - SQLite extension for Visual Studio code installed
- Steps:

1. Create a new python file and import dependencies from prerequisites.
2. Connect to the products.db (if not created, connect method will create new db)
3. Create cursor object, that allow us to interact with the database using SQL commands
4. With using SQL cursor command Create tables products, product_images, products_additional_info with the data types and columns from the products.db schema
5. Set the base URL of the main page and override header, that stores the user agent Mozilla/5.0. Else the user agent is Python and the requests might get blocked.
6. Explore elements from https://www.thewhiskyexchange.com that we are aiming to scrape
7. Write a script to go through each products in selected category (productlist) and get a URL, append the baseurl with the URL of the product to access the individual product page.
8. In order to cover all of the pages, introduce for loop from 1 to 6 and charge the url so we show 24 results on single page.
9. Explore the following elements and find their class and positioning on the website: Price, Product additi.info, Product description, Overall rating, Product name, Product sub title, Category, Subcategory, Product ID, Image Url, Alt text, Product image additional info, Choices.

10. In Try catch methods, we extract the individual elements from the website. If not found for example: rating then the result should return empty string. Make sure that the result is matching with the data type defined at the beginning. In order to achieve this, some further modification of the result will be needed.
11. For testing purposes we can create a dataframe and print the product objects in it we can see the result in console
12. To keep going with the database, we execute commands that inserting the scraped data into our 3 tables based on the schema and we commit to the sqlite.and close the connection

Cronjob

- Prerequisite:
    - DigitalOcean account to setup droplet (Linux server in the cloud)
    - Github repository hosting the scraping python file from previous step
    - Windows Terminal, Powershell or similar
- Steps:

1. Create a new droplet run on Ubuntu in Digital Ocean (The provider is optional and can be changed, we benefited from free credits)
2. Run command in Powershell ssh root@"ipaddress will be here" to connect to the droplet.
3. Run apt-update and apt-upgrade to update the system and upgrade the packages.
4. Run apt install python3-pip to install the pip package manager so we can install all python packages needed - requests, pandas, beatifulsoup4
5. In home directory on the server clone the Github repository
6. Create cron job by typing following command : crontab -e, select the nano editor.
7. Go to https://crontab.guru/#0_8_*_*_* and explore the options in what time interval the cronjob can run, we have selected to run it every day at 8:00 in the morning and adjust the interval value to the specific value.
8. After the starts we have to add path to the python on the pc, the python file and the file, where we want to see the output of the cronjob. That might look for example like this: 0 8 * * * /usr/bin/python3 /home/scraper/scraper.py >> /home/cron.log
9. Write ls command to the home folder the new cron.log file should appear, we open the file with cat command. We should see the dataframe with the products scraped in the scraper.py, depending on what time we look into cron.log the number of result should increase by 1 after 8:00 every morning.

Frontend

- Prerequisite:
    - GraphQL set up - link to connect is provided
    - Rest api set up - link to connect is provided
- Steps

1. Create a new React project
2. Add link to other group CDN in a img tag
3. Create component Buttons for Updating Profile, Wish item, Send Invite
4. Create components Products, Login and Register and routing

5. Set up Apollo Client to connect to GraphQL
6. Create new Queries file, where the GetAllProducts query is stored
7. Import query to the Products page and fetch the products details to the form on the page.
8. Install Axios
9. In the Login and Register page create axios post request for login and signup and fetch the required data such as username, email and password with the form, use useState to track the state of the function component.
10. Make post request with axios to Update user info service of partner group, change state of user info
11. Make post request with axios to Invite user of parther group, attach it to the button
12. Add websocket listener to listen for events when someone logs in,attach to table.
13. Add request to get all of the wishes of a user from RSS, attach to button.
14. Deploy the frontend to Azure App Service