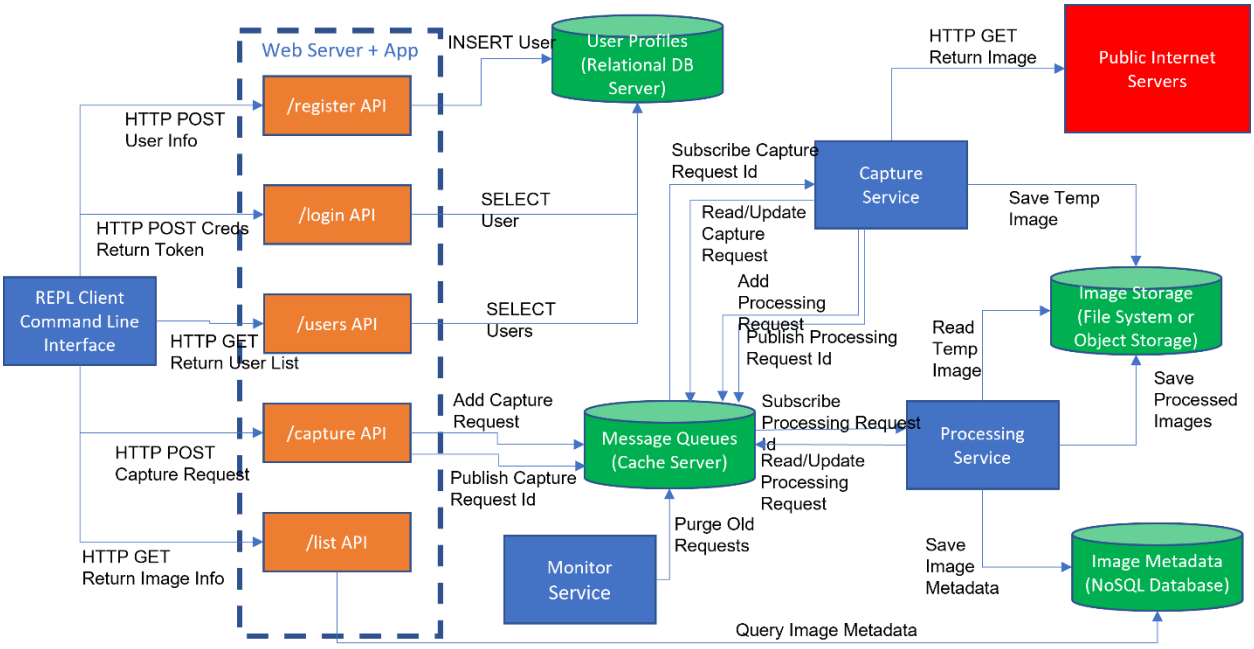
NQT Application Documentation

This document provides additional information useful for working with the NQT Application.

Application Overview

The NQT (Not Quite Trivial) Application is a small application with limited functionality structured in a way similar to many cloud-centric applications: it has relatively small modules (referred to as services) each with a specific function, and it uses several technologies that are commonly used in cloud applications.



The main function of the application is to allow users to capture image files off the internet, resize them, and store them in a personal image repository. Around this core function are typical requirements such as user authentication and mechanisms to support data storage and communication among the modules in the application. The following techniques are illustrated within NQT:

- REST APIs implemented in a web server
- JWT-based authentication (JWT = JSON Web Tokens)
- Headless background services
- Object-relational mappers (ORM)
- Relational databases (RDBMS)
- No-SQL databases
- Cache servers
- Asynchronous messaging
- Use of environment variables to manage secrets

NQT is written in Python and uses free and readily available community editions of the required server tools.

Installing and Configuring the Application

Installation steps are provided separately for each environment to which the application will be deployed. Details about environment variables for each service are covered later in this document.

Running the Application

For this section, we assume that the application has been successfully installed and configured. To run the application, 8 processes need to be running:

1. Three data servers, installed and if necessary fully configured:
    a. MySQL. Software installed, application database created and populated, and service running.
    b. Memurai or Redis. Software installed and service running.
    c. MongoDB. Software installed and service running.
2. The 5 application processes must also be running within their respective Python virtual environments. The processes can have their own command window (e.g. on Windows), or they may be headless (e.g running as Linux services on a cloud VM). On Windows, you can start all the services at once by running the startup.bat script. You can start services individually by opening a command window and entering the commands manually. The 5 application processes are:
    a. The web server. Program name is app.py.
    b. The capture service. Program name is capture.py.
    c. The processing service. Program name is process.py.
    d. The monitor service. Program name is monitor.py.
    e. The user interface front end. Program name is TestREPL.py.
3. With all processes started, you can view the log messages for web server, capture, processing and monitor service. The messages will indicate whether service startup was successful for each.
4. The first time you start TestREPL, you will need to register as a user. Use the **register** command in TestREPL to do so. Remember your user id and password.
5. Each time you run the application, you will need to login. Use the **login** command to do so.
6. To exercise the application, use primarily the **capture** command to capture and process an image, and the **list** command to view details about images you have captured.
7. After doing a few captures, you can view the effects they had on the system:
    a. You can view the logs for the web server, capture, processing and monitor services to see messages they produced.

b. You can connect to MySQL using MySQL Workbench and run queries against the application database.
c. You can connect to Redis or Memurai with the redis-cli or memurai-cli commands, respectively and run queries against the cache server.
d. You can connect to MongoDB with Compass and run queries against the database.

Web Server

The web server component provides HTTP APIs for all of the externally-accessible features of the application.

Logic and Features

- Create and run a Flask web server. The web server has several API routes to support each of the required functions
- Load environment variables from the .env file and use them to parameterize objects created later
- Define a class called **User** which simultaneously defines the properties of User objects and the mapping of those properties to a relational database table called **User**.
- Connect to the relational database server using the underlying database client library
- Connect to the MongoDB server using a database name of **imagedata** and a collection name of **images**
- Connect to the cache server. In the base NQT application, the cache server is playing the role of both a request queue and an asynchronous messaging pub/sub server.
- Serve requests for the routes identified below
- The Flask server writes log messages to stdout

Technologies Used

- The Python **flask** module is used to provide the HTTP server
- The Python **werkzeug.security** module is used to encrypt passwords for storage in the database
- The Python **jwt** module is used to encrypt and decrypt JSON Web Tokens (JWT) passed between clients and the web server
- The Python **flask_sqlalchemy** module is used to provide object-to-relational mapping of Python classes to relational database tables
- The Python **pymongo** module is used to support a client connection to the MongoDB (NoSQL) database server
- The Python **redis** module is used to support a client connection to the Redis or Memurai cache server
- The Python **dotenv** module is used to read the .env file and populate environment variables

HTTP Server Routes

- **GET /**
  Returns an HTML page for testing the /login API. Not used by the TestREPL client.
- **POST /register**
  Registers a new user in the relational database that holds user names and passwords
- **POST /login**
  Authenticates the user and returns a JWT token to the client. This token must be provided by the client as an HTTP header to access protected APIs in the web server.
- **POST /capture**
  Creates a request to capture and process an image. The request message is written to the cache server, then a pub/sub publish message is also sent to the cache server to notify the capture service that a request needs to be processed. The /capture request requires an HTTP x-access-token header containing a valid JWT token.
- **GET /list**
  For the user identified in the JWT token, returns a list of images captured and their variants as a JSON document. The /list request requires an HTTP x-access-token header containing a valid JWT token.
- **GET /users**
  Returns a list of registered users and their details as a JSON document. This can be considered an administrative function rather than an end-user function. The /users request requires an HTTP x-access-token header containing a valid JWT token.

Capture Service

The Capture Service processes requests to capture an image from the public internet and store it into a temporary storage location for later processing.

Logic and Features

- Load environment variables from the .env file and use them to parameterize objects created later
- Connect to the cache server. In the base NQT application, the cache server is playing the role of both a request queue and an asynchronous messaging pub/sub server.
- Start worker threads to process requests.
- Execute an infinite main loop on the main thread. The main loop subscribes to the “capture” topic from the pub/sub server and queues each request received for processing by a worker thread.
- Each worker thread processes requests with the following steps:
  - Mark the capture request as in-progress in the cache server
  - Make an HTTP(S) request to retrieve an image from a public internet server
  - Save the retrieved original image to storage under a generated file name.

    - Create a request to process the image in the cache server. This request will be processed by the processing service
    - Send a pub/sub publish message to the cache server to notify the processing service that a request needs to be processed
    - Mark the capture request as completed in the cache server
- The print statement is used to write log messages to stdout

Technologies Used

- The Python **redis** module is used to support a client connection to the Redis or Memurai cache server
- The Python **dotenv** module is used to read the .env file and populate environment variables
- The Python **requests** module is used to make HTTP requests to retrieve images from a public internet server
- The Python **queue** module is used to provide a thread-safe queue shared between the main loop and the worker threads

Processing Service

The Processing Service processes requests to resize and store multiple variants of a captured image. It also produces a document in the NoSQL database that describes the image and the variants that were produced.

Logic and Features

- Load environment variables from the .env file and use them to parameterize objects created later
- Connect to the cache server. In the base NQT application, the cache server is playing the role of both a request queue and an asynchronous messaging pub/sub server.
- Start worker threads to process requests.
- Execute an infinite main loop on the main thread. The main loop subscribes to the “process” topic from the pub/sub server and queues each request received for processing by a worker thread.
- Each worker thread processes requests with the following steps:
    - Mark the process request as in-progress in the cache server
    - Parse the request string to determine which resizing operations were requested
    - Save a copy of the original image in the saved file storage area
    - Process each resizing request and store the resulting image in the saved file storage area
    - Keep track of all images written to the saved file storage area, including the file name of each resized image
    - Delete the original image from the temporary storage area

    - Create a document in the NoSQL database that retains image details and the list of variants created with the file name of each
    - Mark the process request as completed in the cache server
- The print statement is used to write log messages to stdout

Technologies Used

- The Python **redis** module is used to support a client connection to the Redis or Memurai cache server
- The Python **dotenv** module is used to read the .env file and populate environment variables
- The Python **pymongo** module is used to support a client connection to the MongoDB (NoSQL) database server
- The Python **queue** module is used to provide a thread-safe queue shared between the main loop and the worker threads
- The Python **PIL** module is used to read, resize and save image files

Monitor Service

Although there is currently no API in the web server to view the status of the request queues (i.e. capture and process requests), the Monitor Service has been implemented to monitor the status of the queues an remove old queue entries. Background processes that monitor the state of the system and perform clean-up operations are a common feature of cloud applications.

Logic and Features

- Load environment variables from the .env file and use them to parameterize objects created later
- Connect to the cache server. In the base NQT application, the cache server is playing the role of both a request queue and an asynchronous messaging pub/sub server.
- Start one thread to monitor the capture request queue and one thread to monitor the process request queue.
- Each thread has an infinite loop that sleeps for an interval, then executes the clean-up process, then sleeps again. In the base NQT application, this interval is hard-coded as 5 seconds
- The clean-up process consists of the following:
    - Query the cache server to retrieve a list of existing requests of the given type – capture or process.
    - Execute a second cache server query to retrieve the request time of each existing request.

    - Build and execute a third cache server query that deletes requests that are older than a cutoff time. Currently, this cutoff time is hard-coded as 1 minute so that you can observe requests being deleted in a timely fashion.
- The print statement is used to write log messages to stdout

Technologies Used

- The Python **redis** module is used to support a client connection to the Redis or Memurai cache server
- The Python **dotenv** module is used to read the .env file and populate environment variables

TestREPL Application

The TestREPL Application provides a front-end user interface to NQT in the form of a read-evaluate-print loop (REPL). TestREPL provides simple prompted inputs to the user and then executes the appropriate web server API for the requested function.

Logic and Features

- Load environment variables from the .env file and use them to parameterize objects created later
- Start a command loop using the cmd module. When the user enters a command, process the command and write appropriate feedback to the terminal, then prompt for the next command.
- A list of supported commands appears below.

Technologies Used

- The Python **cmd** module is used to provide the framework for the command line interpreter (also known as the REPL)
- The Python **dotenv** module is used to read the .env file and populate environment variables
- The Python **requests** module is used to make HTTP API requests against the web server

Supported Commands

- **?** or **help** command
    - Enter **?** or **help** and press enter
    - Program prints a list of available commands with no descriptions
- **help** <command>
    - Enter help followed by a command name and press enter
    - Program prints a very brief, almost useless, description of the given command
- **register** command
    - Enter **register** and press enter

    - The program prompts you for email address, name and password
    - When complete, the program executes POST /register on the web server
- **login** command
    - Enter **login** followed by your email address and press enter
    - The program prompts you for your password
    - When complete, the program executes POST /login on the web server and retains the returned token in memory to indicate a logged-in state
- **user** command
    - Enter **user** and press enter
    - If you are logged in, program prints your username and the encrypted contents of your JWT token
- **capture** command
    - Enter **capture** and press enter
    - Program prompts you for an image URL. Enter the URL of the image you want to retrieve and press enter. In order to work, the URL must return a response with a content type of image/jpg, image/jpeg, image/gif, image/png or image/bmp.
    - Program prompts you for a document name (required) and description (optional)
    - Program prompts you for the resizing operations you want performed. You can leave blank to create a thumbnail with max width of 100 and max height of 100 pixels. Or, you can enter one or more resize operations separated by commas. Two operations are supported: thumb WWWxHHH creates a thumbnail image with max width = WWW and max height HHH. Scale W.WxH.H scales the width and height of the image by factors of W.W and H.H respectively. W.W and H.H must be valid floating point numbers with reasonable values. An example of a valid operation string is “thumb 75x75, scale 1.2x1.2”. Press enter.
    - When complete, the program executes POST /capture on the web server and prints a message regarding whether your request was accepted.
- **list** command
    - Enter **list** and press enter
    - If you are logged in, program executes GET /list on the web server and prints the JSON response, which contains a list of your successfully completed image requests and the variants that are stored in your personal repository
- **users** command
    - Enter **users** and press enter
    - If you are logged in, program executes GET /users on the web server and prints the JSON response, which contains a list of users registered in the application database
- **logout** command
    - Enter **logout** and press enter

    - The program discards your JWT token and reverts to a non logged-in state
- **bye**, **exit**, and **quit** commands
    - Enter any of the above commands and press enter
    - The TestREPL application terminates normally

Environment Variables

This section identifies the environment variables that are currently available in the NQT Application. For each service, these settings reside in a file named .env in the folder that contains the service source code. The application uses the Python dotenv module to load the .env file and make the variable values available to the service.

Web Server

Variable Name	Description
SECRET_KEY	Sample value: `SECRET_KEY="just some key Im using for testing"`
DBCONN_STR	Sample value: `DBCONN_STR="mysql+mysqlconnector://cs790app:#cs-790-app@127.0.0.1/cs790app"`
REDIS_HOST	Sample value: `REDIS_HOST=localhost`
REDIS_PORT	Sample value: `REDIS_PORT=6379`
MONGO_CONN_STR	Sample value: `MONGO_CONN_STR=mongodb://localhost:27017/`

Capture Service

Variable Name	Description
REDIS_HOST	Sample value: `REDIS_HOST=localhost`
REDIS_PORT	Sample value: `REDIS_PORT=6379`
TEMP_FILE_PATH	Sample value: `TEMP_FILE_PATH=C:\Users\mdenz\source\repos\CS790Fall23\IaaS\TempFiles`
WORKER_POOL_SIZE	Sample value: `WORKER_POOL_SIZE=3`

Processing Service

Variable Name	Description
REDIS_HOST	Sample value: `REDIS_HOST=localhost`
REDIS_PORT	Sample value: `REDIS_PORT=6379`
TEMP_FILE_PATH	Sample value: `TEMP_FILE_PATH=C:\Users\mdenz\source\repos\CS790Fall23\IaaS\TempFiles`

WORKER_POOL_SIZE	Sample value: `WORKER_POOL_SIZE=3`
MONGO_CONN_STR	Sample value: `MONGO_CONN_STR=mongodb://localhost:27017/`

Monitor Service

Variable Name	Description
REDIS_HOST	Sample value: `REDIS_HOST=localhost`
REDIS_PORT	Sample value: `REDIS_PORT=6379`

TestREPL Application

Variable Name	Description
WEB_SERVER_URL	Sample value: `WEB_SERVER_URL=http://localhost:5000`