Redis Integration Report – CRUD Operations via C# & Swagger

Overview

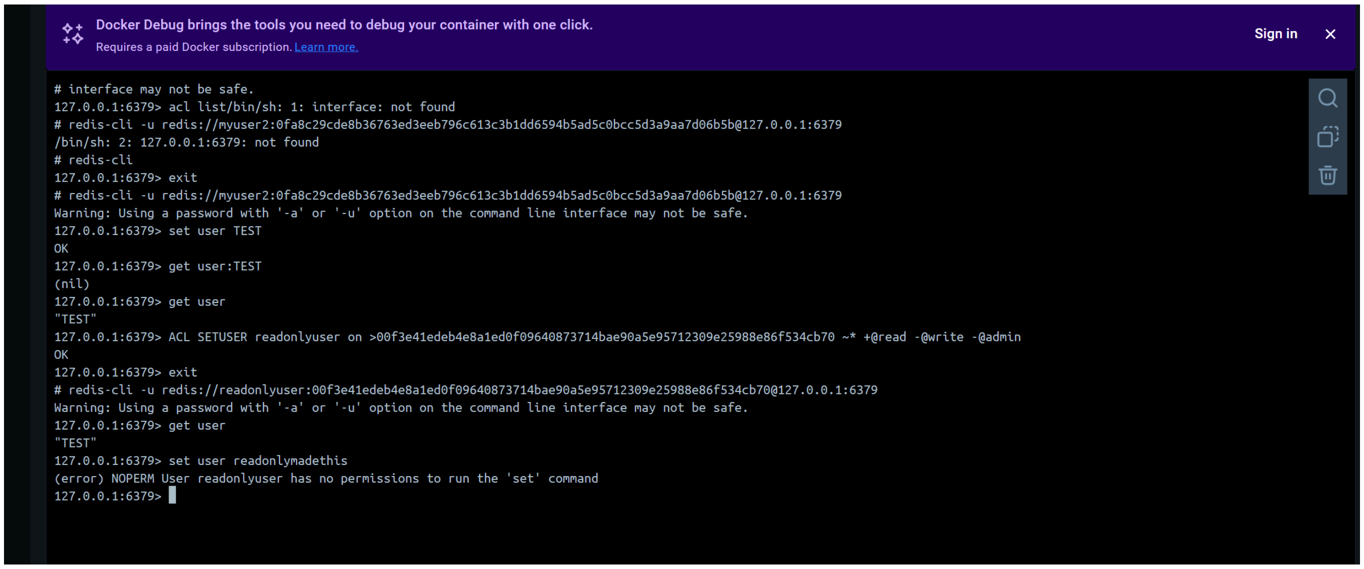
Group Members: Jamie & Isak
The goal of this implementation was to set up a Redis-based data store and build a simple C# application using Swagger for interaction. The focus was to enable full CRUD (Create, Read, Update, Delete) functionality while adhering to a basic security and expiration policy for stored user data.

We selected the Redis Configurations

- 1 - Redis with Retention Policy and-
- 4 - Redis Security
  As these were expected to integrate nicely with each other.

Implementation Steps

We began by spinning up a Redis instance locally and configuring ACL (Access Control Lists) to create isolated user access. A user named `myuser2` was assigned full privileges (`+@all`) with a hashed password for security. Additional users were created for experimentation, including a read-only user with limited permissions called **readonlyuser**


Docker Debug brings the tools you need to debug your container with one click.
Requires a paid Docker subscription. Learn more.
Sign in

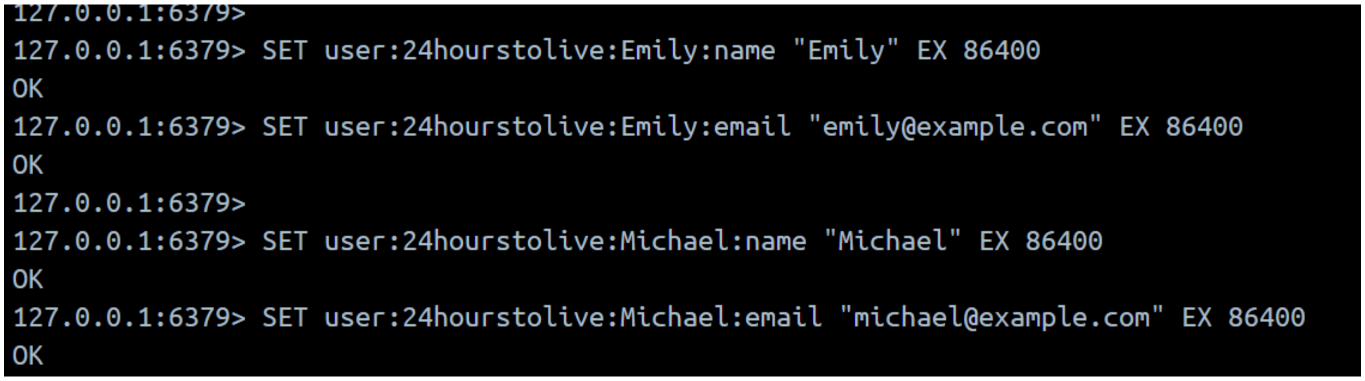
```
# interface may not be safe.
127.0.0.1:6379> acl list/bin/sh: 1: interface: not found
# redis-cli -u redis://myuser2:0fa8c29cde8b36763ed3eeb796c613c3b1dd6594b5ad5c0bcc5d3a9aa7d06b5b@127.0.0.1:6379
/bin/sh: 2: 127.0.0.1:6379: not found
# redis-cli
127.0.0.1:6379> exit
# redis-cli -u redis://myuser2:0fa8c29cde8b36763ed3eeb796c613c3b1dd6594b5ad5c0bcc5d3a9aa7d06b5b@127.0.0.1:6379
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
127.0.0.1:6379> set user TEST
OK
127.0.0.1:6379> get user:TEST
(nil)
127.0.0.1:6379> get user
"TEST"
127.0.0.1:6379> ACL SETUSER readonlyuser on >00f3e41edeb4e8a1ed0f09640873714bae90a5e95712309e25988e86f534cb70 ~* +@read -@write -@admin
OK
127.0.0.1:6379> exit
# redis-cli -u redis://readonlyuser:00f3e41edeb4e8a1ed0f09640873714bae90a5e95712309e25988e86f534cb70@127.0.0.1:6379
Warning: Using a password with '-a' or '-u' option on the command line interface may not be safe.
127.0.0.1:6379> get user
"TEST"
127.0.0.1:6379> set user readonlymadethis
(error) NOPERM User readonlyuser has no permissions to run the 'set' command
127.0.0.1:6379>
```

Altough as seen in the above screenshot with the **"24hourstolive"** users, we initially did make use of some very confusing naming conventions that most likely isn't best practice at all but that said we did improve upon that in the later examples.

Next, in the C# project, we:

1. Integrated the `StackExchange.Redis` library for communication with Redis.
2. Exposed endpoints via an ASP.NET Core Web API project.
3. Configured Swagger to display our API and allow direct interaction.
4. Ensured all keys related to user data were prefixed with `user:24hourstolive:*`, and added TTLs of 24 hours to them for automatic expiry.

The TTL was enforced at the time of data insertion using `SETEX` or `SET` followed by `EXPIRE`. This guaranteed automatic cleanup after 24 hours, aligning with our use case of temporary user data.



```
127.0.0.1:6379>
127.0.0.1:6379> SET user:24hourstolive:Emily:name "Emily" EX 86400
OK
127.0.0.1:6379> SET user:24hourstolive:Emily:email "emily@example.com" EX 86400
OK
127.0.0.1:6379>
127.0.0.1:6379> SET user:24hourstolive:Michael:name "Michael" EX 86400
OK
127.0.0.1:6379> SET user:24hourstolive:Michael:email "michael@example.com" EX 86400
OK
```

Challenges & Learnings

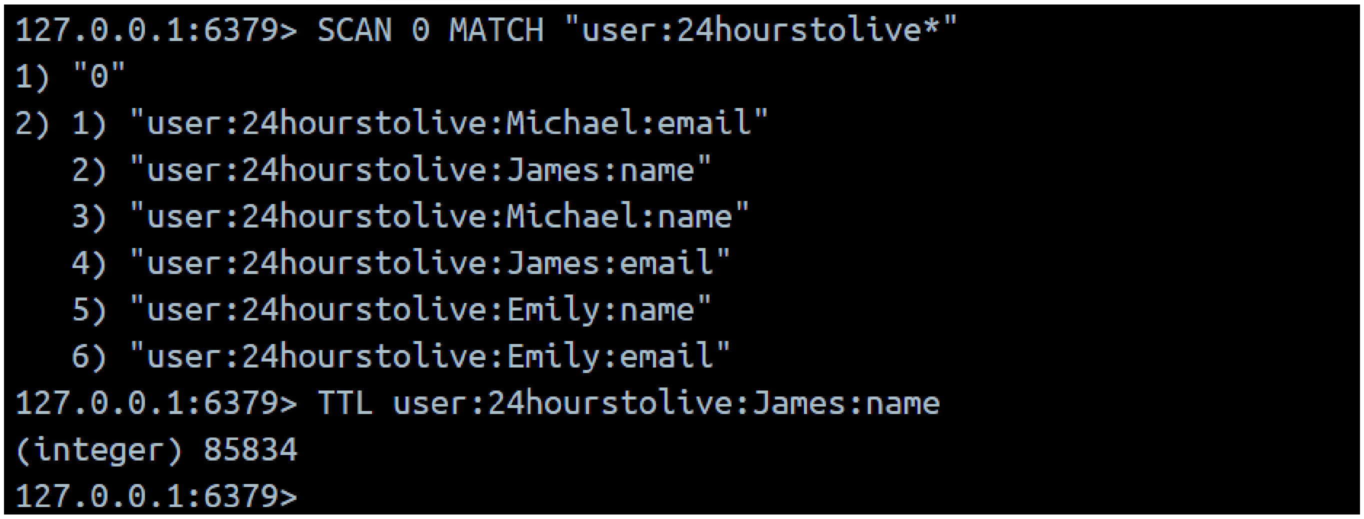
The most significant challenge was around **user authentication in Redis**, Redis does not support logging in with unhashed passwords when users are configured using the `ACL SETUSER` command with a hashed password. This created confusion and led to several failed attempts before resolving the issue by using the correct pre-hashed password in the CLI command.
Altough this didn't come as a surprise to use that it wasn't possible are long hours of working on this we became somewhat blind to the obvious and didn't realise we just should've used the hashed password the first time around.

Another frustrating aspect was the error messaging when trying to access Redis from an external source. Redis runs in "protected mode" by default and rejects outside connections unless:

- A password is set for the default user, or
- Protected mode is explicitly disabled.

We opted to **set a password for the default user**, which allowed external connections while keeping the instance relatively secure. Mostly as it felt Unsafe to disabled protected mode and we imagined it being unadvisable even tough Redis itself isn't known to be safe anyways.

Another discovery was that **Redis doesn't support querying keys like** `GET user` since keys are purely string-based and don't have types like "users." Keys had to be manually scanned using `SCAN` with a matching pattern like `user:24hourstolive*`, and TTLs had to be individually fetched using `TTL keyname`.



```
127.0.0.1:6379> SCAN 0 MATCH "user:24hourstolive*"
1) "0"
2) 1) "user:24hourstolive:Michael:email"
   2) "user:24hourstolive:James:name"
   3) "user:24hourstolive:Michael:name"
   4) "user:24hourstolive:James:email"
   5) "user:24hourstolive:Emily:name"
   6) "user:24hourstolive:Emily:email"
127.0.0.1:6379> TTL user:24hourstolive:James:name
(integer) 85834
127.0.0.1:6379>
```

Swagger integration was relatively smooth though.

Conclusion

Despite the hiccups with Redis ACLs and key scanning limitations, we now have a fully functional Redis-backed API with proper security and time-bound data expiration. The process highlighted both the power and simplicity of Redis.

Lastly here is a PDF of the working API endpoints: https://github.com/TofuBytes-Studies-Group/OLA2_Redis_Study_point/blob/main/OLA2_Redis_APP_ENDPOINTS.pdf

---