A logo with a star

Description automatically generated

Task 2 ICT320 Database Design:

Find a Campsite Login System

Student Name: Liam Wyer

Student Number: 1167044

Submission Date: 22/09/2025

Table of Contents

[Introduction 4](#_Toc209600677)

[System Design and data model 5](#_Toc209600678)

[Testing and Results 8](#_Toc209600679)

[Future Enhancements 15](#_Toc209600680)

Figure 1User record stored as a Redis HASH; secrets are stored only as bcrypt hashes (no plaintext). 7

Figure 2 Catalogue of approved security questions stored once as a Redis LIST. 8

Figure 3 Application startup confirms Redis connectivity and presents the CLI menu. 10

Figure 4 New account created with normalised login and hashed secrets and Immediate sign-in verifies write/read path and bcrypt verification. 11

Figure 5 Recovery with correct answer → “Password updated 11

Figure 6 Successful sign-in using the newly set password. 12

Figure 7 First CSV import creates users from the assessment file. 13

Figure 8. Second CSV import is idempotent—existing users are skipped. 14

Figure 9Redis Insight sec\_questions LIST contains the five approved questions 15

Figure 10 Structured audit trail of authentication events (success and failure) 16

# Introduction

This report presents the design and implementation of a lightweight user-authentication capability for the Find-a-Campsite platform, built on Redis as a key–value data store. Redis is an in-memory system known for sub-millisecond latency and horizontal scalability. The solution addresses a clear operational need: securely onboard new customers, authenticate returning users with minimal latency, and provide a controlled path to password recovery.

Redis was selected for its performance, simple key schema and mature ecosystem. User information is stored in a consistent structure and protected using hashes, a one-way method that converts a secret (such as a password or security answer) into a value that cannot be reversed. Only hashed values are stored; even if the database were accessed by an unauthorised party, credentials would remain unreadable. During sign-in or reset, the system hashes what a customer enters and compares it with the stored hash. The catalogue of approved security questions is maintained separately to keep lookups fast and behaviour consistent at scale.

The authentication service is delivered as a single Python application that runs reliably on developer workstations and in automated pipelines. Configuration is externalised through a .env file, an environment configuration file read at start-up, that holds connection strings and other secrets outside the codebase allowing credential rotation and easy targeting of different Redis endpoints without code changes. Here, *secrets* means sensitive operational values such as the Redis connection string/password, API keys and tokens—items that must not be stored in source control or logs. Core capabilities include secure account creation, user sign-in, password recovery using knowledge-based verification and bulk user provisioning from CSV. Sensitive values are never stored in plaintext; passwords and security answers are protected using industry-standard hashing (bcrypt).

This document describes the data model and application design, traces the workflow from account creation through recovery and demonstrates correct behaviour with representative runs. A dedicated Future enhancements section then returns to the themes of security, speed and simplicity, proposing improvements in observability, configuration hardening and user experience as usage scales.

# System Design and data model

A screenshot of a computer

AI-generated content may be incorrect.The service adopts a deliberately small data model to keep performance predictable and behaviour easy to reason about. Each customer record is stored under a single key derived from the login identifier, allowing the application to fetch all required fields with a constant-time lookup. The record contains the first name, the selected security question and the hashed forms of the password and answer. This keeps per-user state self-contained while avoiding joins, secondary indexes or multi-step reads that add latency and complexity.

Figure 1User record stored as a Redis HASH; secrets are stored only as bcrypt hashes (no plaintext).

Shared reference data, the catalogue of approved security questions, is maintained separately from user records. Storing the catalogue once and referencing its values by text ensures a consistent experience at sign-up and recovery, while it allows the list to evolve without touching existing accounts. This separation of concerns means operational changes, such as adding or retiring questions, can be deployed quickly without reprocessing user data.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 2 Catalogue of approved security questions stored once as a Redis LIST.

Key naming follows a simple, human-readable convention so operations remain transparent as the dataset grows. User records are scoped under a dedicated prefix tied to the login identifier, reducing the chance of collision with other application data and making targeted tasks, such as inspection, backup or selective deletion, straightforward. The scheme is intentionally conservative: one key per user, one structure per key, no embedded formatting or versioning in the value. That restraint translates to lower risk and simpler maintenance.

Security controls are built into the model rather than layered on afterwards. Secrets are never stored directly; the system persists only their hashes and verifies access by hashing new input and comparing it with the stored value. Because the hashes cannot be reversed to recover the original secret, the blast radius of any inadvertent disclosure is materially reduced. At the same time, the structure remains compact enough to support very low-latency reads, which is essential for a responsive sign-in experience.

The model accommodates bulk provisioning without distorting the design. When accounts are imported from a comma-separated file, the application normalises the login identifier, applies the same hashing routine and writes the same compact record used in interactive creation. Using one pathway for both manual and automated flows ensure uniform behaviour, simplifies auditing and helps the system scale cleanly as volumes increase.

# Testing and Results

Testing verified correct behaviour under normal use and the responsiveness expected of an in-memory key–value store. Runs were executed on a developer workstation against a managed Redis endpoint configured via the .env file. Each run began by confirming connectivity, so observations reflected authentication logic rather than environment issues.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3 Application startup confirms Redis connectivity and presents the CLI menu.

Interactive account creation was validated by registering a new customer with a unique login, first name, password and selected security question. The application normalised the identifier, hashed the secrets and wrote a single record under the user’s key. A follow-on sign-in with the same credentials succeeded immediately, demonstrating consistent write/read paths and correct bcrypt verification. Re-creating the same account returned an “already exists” response, confirming predictable handling of duplicates.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

Figure 4 New account created with normalised login and hashed secrets and Immediate sign-in verifies write/read path and bcrypt verification.

A white background with black text

AI-generated content may be incorrect.Password recovery was tested by presenting the stored question, accepting an answer and permitting a reset when the answer was correct. The new password was hashed and persisted; a subsequent sign-in with the updated secret succeeded. Incorrect answers were rejected without exposing stored values, indicating that recovery protects confidentiality and leaves records consistent.

Figure 5 Recovery with correct answer → “Password updated

A close-up of a computer screen

AI-generated content may be incorrect.

Figure 6 Successful sign-in using the newly set password.

Bulk provisioning from a comma-separated file created the expected accounts on first import and skipped duplicates thereafter. Immediately after import, representative users authenticated using the file’s credentials, and recovery behaved as above. The parity between interactive creation and bulk import shows the same validation and hashing rules apply across both paths, simplifying operations and reducing risk.

**Mini Test Plan**

|  |  |  |  |
| --- | --- | --- | --- |
| **Scenario** | **Steps** | **Expected** | **Evidence** |
| Create to Login | Create new user; login | Welcome message | Figs 1–2 |
| Duplicate create | Create same login again | “already exists” | Fig 3 |
| Login fail | Login with wrong pw | “Incorrect password” | Fig 2(fail) |
| Recover fail | Wrong answer | Rejected | Fig 4 |
| Recover success | Correct answer; set new pw | “Password updated” | Fig 5 |
| Login with new pw | Login using new pw | Welcome | Fig 6 |
| CSV import x2 | Run import twice | Created N then 0 | Figs 7-8 |
| Keys shape | Inspect user HASH & questions LIST | Fields present, hashes visible | Figs 9 |
| Audit log | Do success + fail | Events recorded | Fig 10 |

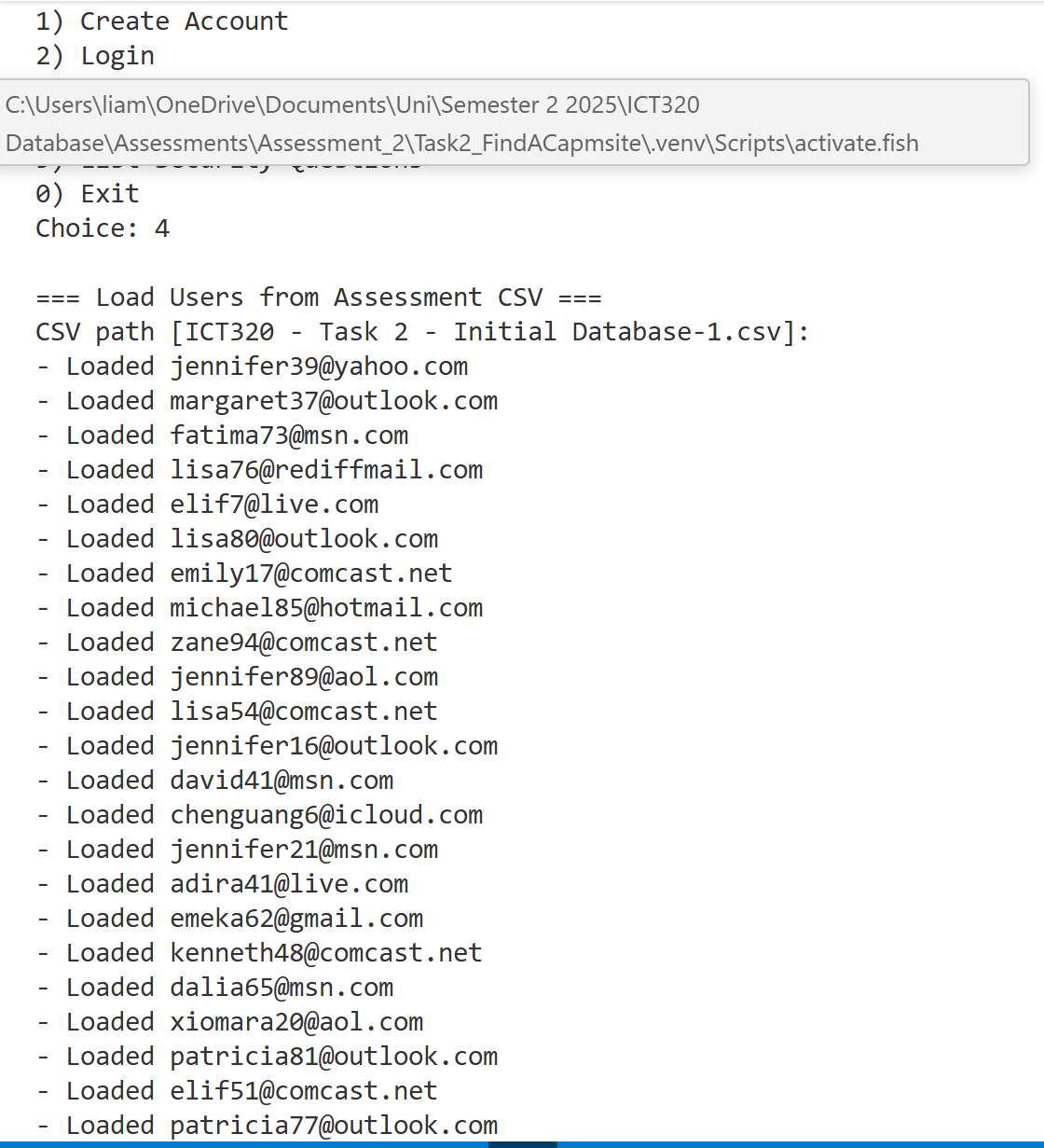


Figure 7 First CSV import creates users from the assessment file.

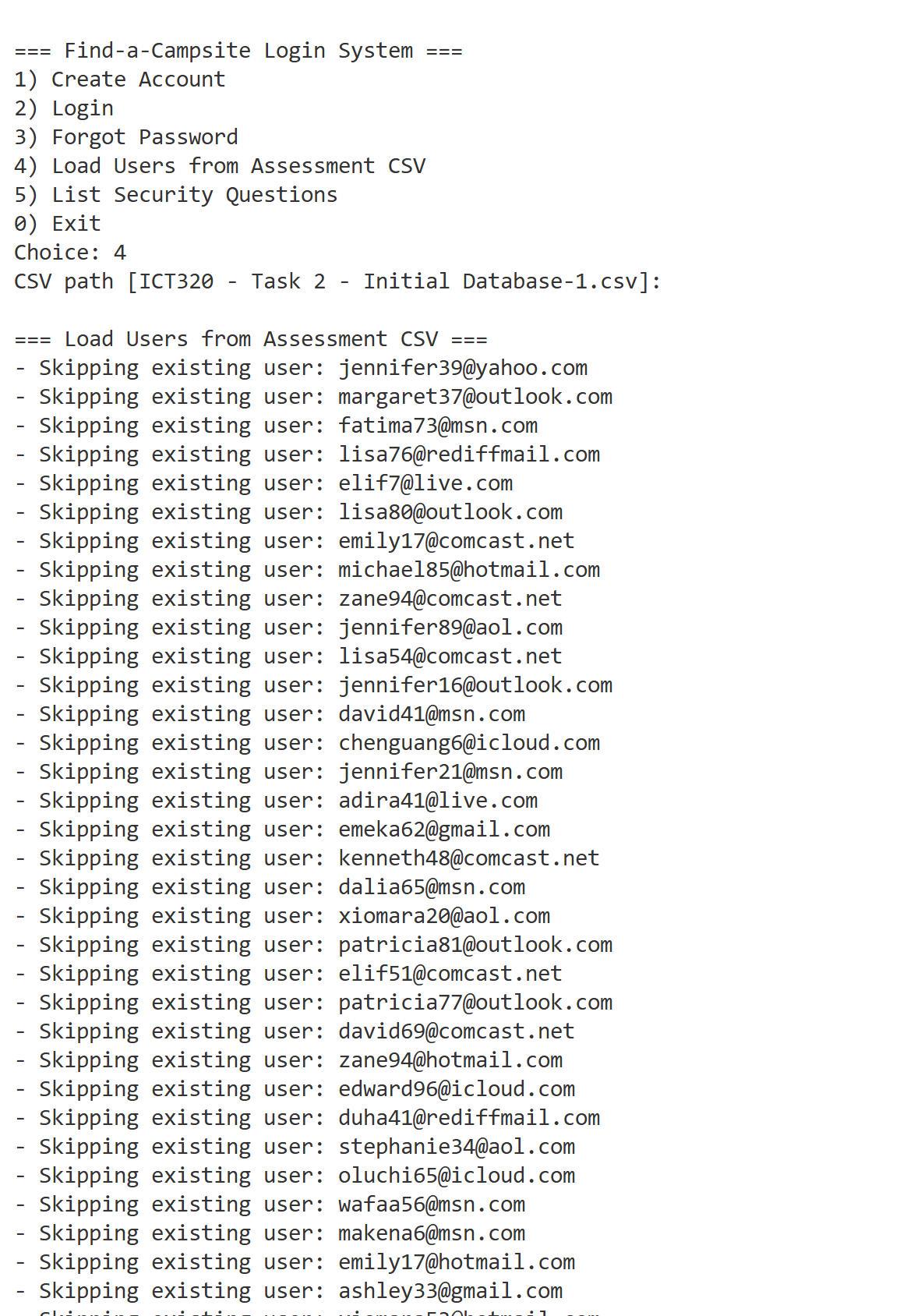


Figure 8. Second CSV import is idempotent—existing users are skipped.

To corroborate application results at the data level, keys were inspected in the Redis console. User records appeared under login-scoped keys with first name, question text and hashed password/answer as designed. The security-question catalogue was present as a single list and no plaintext secrets were observed. Interactions across scenarios were responsive and completed without noticeable delay, consistent with an in-memory store. Overall, the service delivers secure account creation, reliable authentication and controlled recovery while upholding the report’s themes of security, speed and simplicity.

The observed behaviour matches the design goals: constant-time lookups by login, secrets stored only as hashes and a compact schema that supports low-latency flows.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 9Redis Insight sec\_questions LIST contains the five approved questions

A screenshot of a computer

AI-generated content may be incorrect.

Figure 10 Structured audit trail of authentication events (success and failure)

# Future Enhancements

The service is intentionally small and dependable; the next step is to strengthen it along the same themes. Security, speed and simplicity Without increasing operational burden.

In security, the immediate improvement is credential hygiene and monitoring. Enforce password-quality checks at creation and reset, introduce configurable lockouts or rate-limiting on repeated failures and add optional multi-factor authentication for higher-risk contexts (e.g., admin accounts). Rotate secrets and endpoints automatically via environment management tooling, and capture tamper-evident audit events (account created, login succeeded/failed, recovery attempted) so unusual patterns can be detected early.

For speed and robustness, add lightweight observability: structured logs, basic metrics (logins per minute, p95 authentication latency, recovery success rate) and health checks. These provide early warning without complicating the code. If usage grows, deploy Redis with backups and high availability (AOF/RDB snapshots, multi-AZ replicas) and consider key expiry for transient artefacts (e.g., short-lived recovery tokens) while keeping core user records permanent.

On simplicity and user experience, wrap the CLI with a small web front end (Flask/FastAPI) so customers can self-serve from a browser. Keep the current data model and flows; expose the same endpoints behind a minimal API. Add a self-service profile update for first name and security question (with re-authentication), and allow admins to bulk-import with dry-run validation that reports duplicates and malformed rows before committing.

From a governance perspective, define a data retention and access policy: who can read which fields, how long to retain audit logs, and how to service deletion requests. Document operational runbooks (resetting a forgotten admin credential, rotating Redis passwords, restoring from backup) so the system remains maintainable as staff change.

These enhancements preserve the core strengths—small, fast, reliable—while making the service clearer to operate, safer to expose and easier to evolve as adoption increases.

# Appendix

Core flows (pseudocode)

CREATE(login, first, pw, q, a):

if EXISTS user:login → return "duplicate"

HSET user:login { first\_name=first, password\_hash=hash(pw), sec\_q=q, sec\_a\_hash=hash(a) }

return "ok"

LOGIN(login, pw):

if NOT EXISTS user:login → return "no\_such\_user"

rec ← HGETALL user:login

if verify(pw, rec.password\_hash) → return "welcome"

else → return "incorrect\_password"

RECOVER(login, answer, new\_pw):

if NOT EXISTS user:login → return "no\_such\_user"

rec ← HGETALL user:login

if verify(answer, rec.sec\_a\_hash) → HSET user:login password\_hash=hash(new\_pw); return "updated"

else → return "bad\_answer"