SEN 306 Software Design and Architecture

Assessment 1:
Weather Data Scraper, Storage, and API for the Maltese Islands

Marks Allocated: **60 marks**
Deadline: **16 January 2026**

Purpose:
You will design and implement a small end-to-end software system with a clear architectural vision. By completing this assessment, you should be able to:

- Translate requirements and quality attributes into an architecture and implementation plan.
- Apply architectural patterns, document design decisions (ADRs), and model the system with C4 diagrams.
- Engineer robust data ingestion from a changing third-party HTML source (web scraping).
- Design an API with a versioned contract (OpenAPI) and error handling conventions.
- Containerize the system and publish it on GitHub with documentation.

Brief:
Build a service that:

- Scrapes current and upcoming weather information from the Maltese Islands Weather website: https://www.malteseislandsweather.com
- Stores the data scraped into a system of your preference.
- Exposes a versioned HTTP API to consume the data.
- Runs in a Docker container with a reproducible build and runtime.
- Is hosted on GitHub with documentation, diagrams, tests, and CI.

Constraints, Ethics, and Professional Conduct
Respect the target website:

- Scrape responsibly: identify a reasonable cadence (e.g., every 30–60 minutes), set a custom User-Agent, implement exponential backoff and request timeouts, and avoid parallelizing requests unnecessarily.
- Cache and reuse previously fetched content when possible.

If scraping is unreliable:

- Implement your system against saved HTML snapshots (checked into your repo under a test resources folder). Your architecture must make the source swappable.

Do not publish secrets in the repo.
You may use libraries but must understand and be able to explain your design and code.
Your code must be your own.

Functional Requirements:

Scraper

- Extract current conditions and an upcoming forecast for Malta from the target site.
- Normalize scraped data to a defined internal domain model (e.g., temperature, humidity, wind, condition text/icon, rainfall, forecast for the next 6 days).
- Deduplicate records; avoid writing identical data repeatedly.
- Provide a way to trigger scraping:
    - Automated scheduling (preferred) or a manual admin endpoint.

Storage

- Choose any store (SQLite/Postgres/MongoDB/embedded KV). Justify with an ADR.
- Persist normalized weather observations and forecasts with timestamps and source metadata.

API (versioned, REST, JSON)

- Provide stable v1 endpoints:
    - GET /v1/health
    - GET /v1/weather/current
    - GET /v1/weather/forecast?days=1..6
    - GET /v1/docs (serves OpenAPI UI) — optional but recommended
- Error handling uses RFC 7807 problem+json format.
    - An example of an error response would be:

```
Content-Type: application/problem+json { "type": "https://example.com/problems/validation-error", "title": "Invalid query parameter", "status": 400, "detail": "Parameter 'days' must be between 1 and 6.", "instance": "/v1/weather/forecast" }
```

Documentation

- OpenAPI specification (YAML or JSON) for the API contract.
- C4 diagrams (Context, Container, Component).
- ADRs for key decisions (see below).

DevOps

- Dockerized build and runtime.
- GitHub repository with CI that at minimum builds the image.

Architecture Decision Records (ADRs):

Create a minimum of five (5) concise ADRs documenting:

- Architectural and rationale.
- Storage choice (e.g., SQLite vs. Postgres vs. Mongo) and schema strategy.
- Scraping cadence and politeness strategy.
- API versioning and error format (RFC 7807).
- Container base image and build choices.

Optional ADRs you can also include are: HTML parsing strategy and caching policy.

Containerization Requirements:

- Dockerfile
  - Produce a minimal runtime image.
  - Non-root user.
  - Healthcheck (curl /v1/health).
  - Configurable via environment variables.
    - Logs location
    - Port
  - Expose only the API port; avoid bundling build tools in runtime image.
  - Docker-compose.yml covering data storage and app service.

Observability and Ops:

- Logging
  - Structured logs (JSON). Include logging details for requests and scrape jobs.
- Metrics
  - scrape_success_total,
  - scrape_failure_total,
  - scrape_duration_seconds.
- Health
  - /v1/health checks DB connectivity and readiness.

Security and Compliance

- Do not hardcode credentials (if any); use environment variables and provide .env.example.
- Run as non-root in container.
- Validate and sanitize all external input (even if limited).

GitHub Requirements

- Private repository (unless instructed otherwise) shared read-only with lecturer (3zero2, Joshua Bugeja)
- Repository structure (suggested):
  - /docs
    - c4-context.png, c4-container.png, c4-component.png
    - openapi.yaml
    - adr/0001-...md etc.
  - /src (application entry)
  - /internal or /app (domain, services)
  - /pkg or /lib (adapters, repositories, http)
  - /scripts (dev scripts)
  - /testdata/html (scraping fixtures)
  - Dockerfile
  - docker-compose.yml (optional)

    - README.md
    - .github/workflows/ci.yml
    - .env.example
- CI pipeline (GitHub Actions)
    - Lint + test + build Docker image.
    - Push to GHCR

Suggested Implementation Technologies

- Language: Any (e.g., Python, Go, Node.js, Java, C#). Choose what you are familiar with and justify your choice.
- Libraries:
    - HTTP server framework (e.g., FastAPI, Express, Gin, Spring Boot, ASP.NET).
    - HTML parsing (e.g., BeautifulSoup, Cheerio, goquery, jsoup).
    - OpenAPI tooling (code-first or spec-first).
    - Database client/ORM (optional).

Deliverables

- Source code in GitHub (share repo read only with lecturer on 3zero2 (Joshua Bugeja)
- Architecture artifacts
    - C4 diagrams, OpenAPI spec, ADRs.
- Docker image buildable from repo.
- README.md covering:
    - Overview and architecture summary.
    - How to run locally (docker and non-docker).
    - Configuration, environment variables.
    - Scraping ethics employed.
- Short demo video (3–5 minutes) showing:
    - Running container.
    - Trigger or scheduled scrape.
    - API calls returning data.
    - Diagrams and ADR highlights.

Ideally all the above should be submitted through the GitHub repo. If not possible then submit through this link (https://www.dropbox.com/request/a0r50Ef29c0EbyK1KwlR) all required material in a compressed file having your name and surname.

Marking Rubric (60 points)

- Architecture and Design **(10)**
    - C4 diagrams clarity and consistency (5)
    - ADRs completeness and rationale quality (5)
- API Quality **(15)**
    - OpenAPI accuracy and versioning (7)
    - Error handling (RFC 7807) (8)
- Implementation **(15)**
    - Scraper robustness and normalization (5)
    - Storage integration and repository abstraction (5)

    - Code quality, structure, and separation of concerns (5)
- DevOps and Containerization **(10)**
    - Dockerfile quality, security (non-root), reproducibility (5)
    - CI pipeline and basic checks (5)
- Documentation and Testing **(10)**
    - README completeness and developer experience (10)

Milestones (Suggested)

By Week 3: Requirements, quality attributes, architecture draft (C4), ADRs #1–#3.
By Week 6: Scraper + parser with fixtures; basic storage; API skeleton + OpenAPI.
By Week 9: Scheduling, observability, containerization, CI; finalize ADRs and docs.
By Week 12: (buffer): Polish, tests, demo recording.