Secured Software Engineering
(COMSM0164)

Workshop 18: Contract-Driven API Engineering

Prerequisite:

(1) Visual Studio Code (VS Code)
(2) A GitHub account
(3) Some knowledge of JavaScript and Node.js for API functional implementation.
(4) Schemathesis
(5) https://editor.swagger.io/
(6) K6 for performance testing (https://k6.io)

Overview

This lab would immerse you in a complete, contract-driven API engineering workflow, beginning with the fundamentals of analysing requirements and designing high-quality OpenAPI specifications. In the first part, you would learn how to translate real-world needs into precise schemas, constraints, and error models, and explore how documentation becomes the backbone of reliable, maintainable systems. Early exercises introduce Schemathesis as a tool for automated, specification-based testing, helping you see how correctness emerges when the contract and the implementation evolve together.

As the lab progresses, you would implement their APIs in Node.js, gaining hands-on experience with validation, routing, error handling, and security. They experiment with multiple authentication schemes such as API keys and JWTs, evaluating their strengths and weaknesses in the context of a distributed sensor network. Schemathesis test cycles reveal mismatches between the specification and the implementation, and students iteratively refine both until they achieve a fully compliant, 100% passing test suite. This iterative loop reinforces the idea that robust APIs are not built in a single pass but through continuous alignment between intent, documentation, and behaviour.

The second part is a project that brings all these skills together through the GreenCity Environmental Sensor Network scenario. You will analyse the system's actors, data flows, and security requirements; derive a complete OpenAPI specification; implement the API in Node.js; and use Schemathesis to automatically generate and validate test cases. You will iterate until the system reaches full contract compliance and then reflect on the effectiveness of the authentication schemes you've implemented. By the end, you would have built a secure, well-specified, thoroughly tested API—mirroring the real-world engineering practices used in modern, high-assurance software teams.

Part 1 - HelloBooks API

Task 1: File Structure and Setup
This part is focused on running the HelloBooks API locally so you have a live system to test against. You will install the Node.js dependencies, start the server, and verify that each endpoint behaves as described in the OpenAPI specification.

1. **Configure project environment**
   Download the OpenAPILab.zip from Blackboard and use the unzipped file to create a Node.js project. The file structure is as follows:

```
/lab
  ├── server.js
  ├── package.json
  ├── main.yaml
  ├── paths
        ├── bookById.yaml
        ├── books.yaml
        ├── status.yaml
  ├── schemas
      ├── book.yaml
  ├── security
      ├── apiKey.yaml
  └── instructions.pdf  (this handout)
```

2. **Install dependencies**
   Run the following command on terminal to install dependencies:

```
npm install
```

3. **Start the API**
   To start the API, run the command:

```
node server.js
```

   You should see:

```
HelloBooks API running on http://localhost:8000
```

Task 2: Node.js API Code

Review `server.js` full implementation of the HelloBooks API in Node.js. The goal is to see exactly how the service behaves before testing it. By reviewing the code, you can connect each endpoint, response shape, and security rule to the corresponding elements in the OpenAPI specification. On completing the review, you will gain insights on how the API is structured, where potential mismatches may arise, and why property-based testing is an effective way to uncover subtle inconsistencies between the spec and the running service.

Task 3: OpenAPI Specification

Starting off with `main.yaml`, review the OpenAPI specification that defines the expected behaviour of the HelloBooks API. The goal is to understand the contract to compare against the running Node.js service. You should examine how endpoints, parameters, response schemas, and security requirements are formally described, and begin to see how these definitions translate into testable properties. By grounding your understanding in the specification before running Schemathesis, you build the foundation needed to identify mismatches, reason about failures, and appreciate how an API contract drives automated property-based testing.

Task 4: Running Schemathesis

In this step, you will study Schemathesis as the engine for property-based testing and running against the live HelloBooks API. You will install the tool, execute it using the provided OpenAPI specification, and observe how Schemathesis automatically generates a wide range of inputs—including valid, invalid, and edge-case variations—to probe the API's behaviour. This part marks the transition from understanding the API and its contract to actively testing it, exposing you to automated test generation, fuzzing behaviour, and the kinds of failures that emerge when an implementation and its specification diverge.

1. **Install Schemathesis**
   Run the following command to install Schemathesis

   ```
   pip install --upgrade "schemathesis[full]"
   ```

2. **Run property-based tests**
   Execute the following command:
   ```
   schemathesis run main.yaml --base-url=http://localhost:8000
   ```

   Observe:
   - How many tests were generated
   - Which endpoints were tested
   - Which failures occurred
   - How security requirements were tested

Task 5: Security Requirement Testing
Next, you will focus on how Schemathesis automatically tests the API's security requirements. This will help you understand the relationship between the OpenAPI contract and real-world authentication behaviour. You would observe how the tool generates requests with missing, malformed, or misplaced API keys, verifies that protected endpoints correctly return `401 Unauthorized`, and confirms that `/status` remains publicly accessible as defined in the specification. By examining these results, you will learn how security rules in an OpenAPI file become executable properties, how implementation drift can expose security flaws, and why automated contract-driven testing is essential for validating both functional and security-critical behaviour in modern APIs.

1. **Identify security properties**
   From the OpenAPI spec:
   - `/books` and `/books/{id}` require an API key
   - `/status` is public
   - API key must be in header `X-API-Key`
   - Missing/malformed keys must be rejected

2. **Observe Schemathesis security tests**
   Run the following with verbose output

   ```
   schemathesis run main.yaml --url=http://localhost:8000 -v
   ```

   Look for:
   - Missing API key tests
   - Wrong header name tests
   - Wrong location tests

- Valid API key tests
- `/status` public access tests

3. **Investigate failures**
   Pick one failure and answer:
   - What input did Schemathesis generate?
   - What did the OpenAPI spec expect?
   - What did the API return?
   - Is this a bug in the API or the spec?
   - How would you fix it?

4. **Classify the failures as:**
   - Schema drift
   - Security drift
   - Incorrect status code
   - Missing validation
   - Unexpected behaviour

Task 6: Achieving 100% Pass Rate
In this part of the lab, you will use the failing Schemathesis test cases as a guide to improve the API or the OpenAPI specification until the entire test suite passes. Each failure represents a mismatch between the contract and the implementation, so you must decide whether the API is wrong, the specification is wrong, or both need adjustment. You will then update the relevant code paths, response schemas, status codes, or security rules to bring behaviour and documentation into alignment. After making changes, rerun Schemathesis and verify that all test cases now pass, demonstrating a fully contract-compliant API. The goal is to reinforce the idea that correctness is not just about making tests green but about ensuring the API behaves exactly as its specification promises.

(1) **Run the current test suite**
- `schemathesis run main.yaml --url http://localhost:8000 --report junit --report-junit-path report.xml`
- **Goal:** Get a baseline: note total tests, passes, and failures.

(2) **Identify failing test cases**
- **Open** `report.xml` in an editor.
- **Find** all `<testcase>` elements that contain `<failure>` or `<error>.`
- **For each failing testcase, record:**
  - **Method:** `GET, POST`, etc.
  - **Path:** e.g. `/books, /status`
  - **Failure message:** from the `<failure>` or `<error>` content.

(3) **Classify each failure**
- For every failing testcase, decide:
  - **Spec is wrong:** The OpenAPI description doesn't match the *intended* behaviour.
  - **Implementation is wrong:** The code doesn't do what the spec (and product intent) says.
  - **Ambiguous:** You need to choose which side to "make true" for the lab.

- **Write a short note** for each: "Fix spec" or "Fix code", plus a one-line justification.

(4) **Fix specification-side issues**

- For failures where the **spec is wrong**, update `main.yaml`:
    - **Status codes:** e.g. add `401` if the API correctly returns it but the spec doesn't.
    - **Schemas:** adjust required fields, types, or formats to match real responses.
    - **Security rules:** ensure `security` and `securitySchemes` reflect actual auth behaviour.
- **Save** the spec and keep a brief changelog of what you altered and why.

(5) **Fix implementation-side issues**

- For failures where the **implementation is wrong**, update the API code:
    - **Return codes:** align with the spec (e.g. return `404` instead of `500`).
    - **Response shape:** ensure fields, types, and nesting match the OpenAPI schema.
    - **Auth behaviour:** enforce or relax protection to match the spec's `security` section.
- **Restart** the API server after changes.

(6) **Rerun Schemathesis and compare**

- Run the same command again:

    -
      ```
      schemathesis run main.yaml --url http://localhost:8000 --report junit --report-junit-path report.xml
      ```
- **Check:**
    - Did the number of failing testcases decrease?
    - Which failures remain, and do they reveal new misunderstandings?

(7) **Iterate until all tests pass**

- Repeat steps **3–6**:
    - Re-classify remaining failures.
    - Adjust spec or code.
    - Rerun Schemathesis.
- **Stop only when:** the run reports **0 failures / 0 errors**.

(8) **Document your final state**

- **Write a short summary** (5–10 sentences) covering:
    - The main kinds of mismatches you found.
    - Which side you changed more often (spec vs implementation) and why.
    - What "100% passing" now guarantees about this API.
- Optional: include a **before/after** snippet of the spec or code that was most instructive.

Task 7: Exporting Successful Test Cases

This is an optional part that focuses on how you can take the validated results from Schemathesis and begin transforming them into inputs suitable for later performance testing. After exporting the test run to a JSON report, identify which requests succeeded—those that matched the OpenAPI contract without triggering failures—and treat these as reliable, correctness-verified interactions. They then translate those successful cases into a simple k6 script, learning how property-based testing can feed directly into load-testing workflows. The goal is to show that performance testing should never start with unverified or potentially incorrect inputs; instead, it should build on a foundation of known-good behaviour established through contract-driven testing.

1. **Export results**
   To export a JUnit XML report run the following command:

   ```
   schemathesis run main.yaml --url http://localhost:8000 --report
   junit --report-junit-path report.xml
   ```

   This will produce:
   - `report.xml` in the current directory
   - a full structured test report
   - data that can be parse for successful test cases

2. **Investigate how you can generate a k6 script for performance testing of endpoints that have passed Schemathesis test.**

Part 2 - GreenCity Environmental Sensor Network (Scenario)

The city of GreenCity operates a distributed network of environmental sensors installed on streetlights, rooftops, and public buildings. These sensors measure:

- temperature
- humidity
- particulate matter (PM2.5)
- CO_2 concentration
- battery level
- device health status

City engineers need an API that allows:

- registering new sensors
- reporting periodic sensor readings
- retrieving historical readings
- querying the latest reading per sensor
- marking sensors as offline/online
- retrieving system-wide statistics

The city plans to integrate this API into dashboards, alerting systems, and long-term analytics pipelines. The API must be well-documented, stable, and fully contract-compliant.

You will achieve a series of tasks to **design, specify, implement, test, and iteratively refine** a complete API for this sensor network until **Schemathesis reports a 100% pass rate**. This mirrors real-world API engineering: the contract is the source of truth, and correctness emerges through repeated cycles of design → implementation → testing → refinement.

1. Analyse the GreenCity Sensor Network Scenario

Begin by extracting structured requirements from the scenario:

- **Actors:** sensors, engineers, monitoring dashboards, automated alerting systems
- **Resources:** sensors, readings, system statistics
- **Operations:** register sensors, report readings, query history, update status
- **Security needs:**

    - sensors must authenticate before submitting readings
    - engineers must authenticate before modifying metadata
    - different roles may require different permissions
    - sensitive endpoints must be protected
- **Threat considerations:**
    - spoofed sensors
    - tampered readings
    - unauthorized status changes
    - replay attacks

Deliverable: a requirements document including a **security analysis**.

2. Design the OpenAPI Specification (with Security Models)
Create `openapi.yaml` describing:

Endpoints
- `POST /sensors` — register a sensor
- `PATCH /sensors/{id}/status` — update online/offline
- `POST /sensors/{id}/readings` — submit a reading
- `GET /sensors/{id}/readings/latest` — latest reading
- `GET /sensors/{id}/readings` — history
- `GET /stats` — system-wide metrics

Security Schemes
Define **at least two** authentication mechanisms in the spec, such as:
- API keys
- Bearer tokens (JWT)
- HMAC signatures for sensor-to-server communication
- Basic auth (for comparison only)

Ensure to assign schemes to endpoints using the `security` section.

Schemas
- `Sensor`
- `Reading`
- `StatusUpdate`
- `Stats`

Deliverables:
- `openapi.yaml`
- a short explanation of chosen security schemes and why they fit the scenario

3. Implement the API in Node.js

Build the API using a Node.js framework:

Core requirements
- Implement all endpoints defined in the spec
- Enforce schema constraints
- Validate request bodies and parameters
- Maintain in-memory or simple database storage

Security implementation

Implement **all authentication schemes** from your spec:

- API key middleware
- JWT verification
- HMAC signature validation for sensor submissions
- Role-based access control for engineer endpoints
- etc

You should be able to **compare** the schemes' effectiveness and complexity.

Deliverable: a running Node.js API server with authentication middleware.

4. Generate Testcases with Schemathesis
Run the following command to generate testcases:

```
schemathesis run openapi.yaml --url http://localhost:8000 \
  --report junit --report-junit-path report.xml
```

This produces a JUnit XML report containing:

- all generated testcases
- failures
- schema mismatches
- security-related errors (e.g., missing auth, wrong status codes)

Deliverable: `report.xml`.

5. Analyse Failures (Including Security Failures)
Inspect `<testcase>` entries containing `<failure>` or `<error>.`
For each failure, classify:

- **Spec is wrong**
- **Implementation is wrong**
- **Security mismatch** (e.g., endpoint requires auth but spec doesn't reflect it)
- **Constraint violation**
- **Unexpected status code**

Pay special attention to:

- endpoints that should be protected but aren't
- endpoints that require authentication but the spec doesn't say so
- inconsistent error responses for unauthorized access

Deliverable: a failure classification table with a dedicated **security section**.

6. Iterate: Fix Spec, Implementation, or Security Rules

Refine:
Spec fixes

- add missing `security` blocks
- correct status codes (`401, 403, 404`)
- tighten schemas
- clarify authentication requirements

Implementation fixes

- enforce authentication consistently
- correct response shapes
- implement missing validation
- align error handling with the spec

Security fixes

- ensure sensors cannot impersonate each other
- ensure engineers cannot perform sensor-only actions
- ensure tokens or API keys are validated correctly

Rerun Schemathesis after each iteration.

Deliverables: updated spec + updated Node.js code.

7. Achieve a 100% Schemathesis Pass Rate

Repeat the cycle until:

- **0 failures**
- **0 errors**
- **100% passing testcases**

This demonstrates:

- full contract compliance
- consistent security enforcement
- correct schema behaviour
- stable, predictable API responses

Deliverables:

- final `openapi.yaml`
- final `report.xml`
- a reflection on how the system improved over iterations
- a comparison of the authentication schemes they implemented

8. Evaluate Authentication Schemes

Students write a short evaluation comparing the schemes they implemented:

- Which scheme is easiest to implement?
- Which is most secure for sensor-to-server communication?
- Which is best for engineer-level access?
- How do they handle replay attacks, key leakage, or token expiry?
- How well does each integrate with Schemathesis testing?

Deliverable: a security evaluation report.

9. (Optional) Extend into Performance Testing

Using the passing testcases:

- extract successful requests from the JUnit XML
- convert them into a k6 script

- run a load test
- evaluate how authentication affects performance

Deliverable: `loadtest.js` + analysis.

Submission

Submit your implementations as a zip file on Blackboard. Ensure to invite the unit lecturer to your github project.

Notes:

1. Submission points will only be open on the day of the workshop.
2. You must be physically present in the lab to submit (except permitted otherwise)
3. You will not gain marks for submitting.
4. Submissions help us differentiate learning and engagement levels when marking both group and individual coursework.

   For all submissions before group assessment, **you will lose 1 mark for each missed submission up to a maximum of 7 marks**.

   For all submissions after group assessment and before individual assessment, **you will lose 1 mark for each missed submission up to a maximum of 7 marks.**