API and Services Challenge Journal

Reading this journal

My thought stream appear in blue bubbles.

> This is an example of a thought.

Decisions I make appear in green bubbles.

> This is an example of a decision.

Timekeeping entries appear in yellow bubbles.

> Total machine time is 0 hours, 0 minutes.

Receiving the challenge

2022-10-27 @ 2pm – Looked at the challenge PDF for the first time.

> I estimate I should have time to complete the exercise before Tuesday 2022-11-01.

Initial thoughts and planning

2022-11-28 @ 9am – Starting journal and capturing my distilled thoughts from yesterday and this morning.

The exercise wants me to start from scratch. As a result I should treat the exercise as I would a proof of concept or prototyping endeavour. That means emphasis on unpacking the core problem space and a reduced focus on maintainability and how will this thing run.

> The core problem appears to have a few dimensions to explore: de-duplication of state update events, reflecting on an appropriate API contract for a wide range of internet connected sensors and choosing an appropriate technology to implement a solution with.

Internet of Things devices could be entirely stateless. That might suggest the API contract and de-duplication logic may be constrained in significant ways. For example, we may not be able to expect a sensor to provide a source timestamp and perhaps even an incrementing index for its state update events. We can probably expect a device to uniquely identify itself, but we might not have any more information than an id and its sensor payload.

The scope of the exercise is quite small. The service will not need much complexity. Most of the persistence, and the REST interfaces could be expressed with a lightweight framework. My instinct is to use Python Flask as it is fresh in my head and lightweight.

I am currently undecided about how to persist data, a relational database doesn't feel necessary or useful for the core behaviour. Perhaps something very simple like Redis would be most appropriate to get started.

The persistence layer should not be authoritative, I will build the API as the authoritative component and the persistence layer should be a means to an end. Other services should not access the persistence layer directly, all access should be via the API. This ensures maximum compatibility with other consumers should the chosen persistence mechanism need to change later.

As our API will be authoritative, it will be important to version the API. I will be building "v1" during this exercise.

I should make a point to consider how the API contract will be published. Thought this is a bit of a stretch goal more than a core problem to solve.

There are probably a few ways to de-duplicate events. I can heavy use of unit testing to describe a few de-duplication methods. I should be able to express them using a common function interface and have any options I explore be substituted in and out.

> The de-duplication method is likely to inform the required fields for the POST endpoint, before firming up the API contract I should explore some ideas for de-duplication.

> While the de-duplication logic will ultimately be the biggest shaper of the API contract, it feels prudent to get something with the basic endpoints built and working before digging into the additional behaviour. I will start by building the two endpoints and committing to a persistence mechanism.

> I will build things using Python Flask as a base as we don't appear to need the broad power of a framework like Play/Spring from the JVM or ASP in the .Net world. Even Django with Python feels overkill for our current requirements. There might be something lightweight in node, but I am not familiar enough with server-side JavaScript to know that off the top of my head.

> I will start by setting up a new Flask project using pip within a Python virtual environment.

Setting up a working environment

I am on a different machine than I would normally work on, so I am grabbing an editor, Visual Studio Code. Installing Python and setting up a new code repository in my GitHub to work to.

> I will make an even more deliberate effort than usual to commit regularly so that even the smallest iterations are captured in version control.

2022-10-28 @ 10am - Installing VSCode and Python

2022-10-28 @ 10:15am – Creating empty git repository and adding empty Flask app

2022-10-28 @ 10:30am – Initial commit created, the repository is available at `https://github.com/adamlynam/api-and-services-challenge`

> Total machine time is 1 hour, 30 minutes.

Building the skeleton

I want to drive all of the behaviour out using Test Driven Development. So I am going to begin by describing the GET endpoint.

I will write an API integration test leveraging pytest and the Werkzeug client. As described here https://flask.palletsprojects.com/en/2.2.x/testing/

2022-10-28 @ 10:40am – Adding pytest

2022-10-28 @ 10:45am – Start writing test for GET endpoint

Got annoyed by JSON formatting issues so installed a linting tool to run on save. Choose Pep8.

2022-10-28 @ 11:40am – First test written for GET endpoint, added hard coded implementation to demonstrate it working

Lots more to do still, I have baked a few assumptions into this first test. I guess we will see how things continue to evolve.

It took a little while to set everything up, hopefully with these foundations laid we can write test and implementation code more quickly from this point.

I should describe the POST endpoint next followed by extending my testing to tease out the persistence layer of the application.

This is much longer than I was expected to spend on the exercise. Maybe I am going too slow for OSHO?

2022-10-28 @ 11:45am – Good time to take a break.

2022-10-28 @ 4:15pm – Got a few minutes spare. I am going to create the POST endpoint.

2022-10-28 @ 4:30pm – Post endpoint is now created.

The POST endpoint spits back what it receives on success. This probably isn't as helpful for internet of things settings, but it can be handy for web applications.

We don't have any persistence yet. A test that consumes both endpoints to ensure updates are successful feels like the best way to tease out the persistence layer.

> The initial pain of setting things up nicely does appear to lend itself to more rapid development now.

2022-10-28 @ 4:35pm – Back to spending time with the family again.

> Total machine time is 3 hours, 0 minutes.

Adding Persistence

2022-10-29 @ 4:00pm – Have a few moments spare to work on the exercise.

> I will use Redis to persist sensor state updates. This gives maximum flexibility for payloads to change and should be super fast to save and retrieve events.

> I will continue by writing a database integration test that describes using the POST endpoint to set a value and then expects to retrieve that value immediately after via the GET endpoint. This should tease out the persistence behaviour we need.

2022-10-29 @ 4:15pm – Failing persistence test has been added.

> Now I will need to refactor both the GET and POST endpoints to call to a common persistence layer. I will start by extracting this layer and use mocking to maintain the current behaviour before I try and add Redis proper. This allows me to maintain my contract defining unit tests I added previously.

2022-10-29 @ 5:20pm – We are now wired up to the new PersistenceService.

> A real solution would have something other than an in-memory dictionary backing the persistence service, but this might actually be enough to demonstrate the important aspects of this exercise. I may keep the in-memory storage mechanism to save time and deliver on the more important aspects of the challenge.

> I need to extract the sensor identifier instead of hard coding it.

> I will address the hard coded sensor id before continuing. I will change the POST endpoint to PATCH and include the sensor identifier in the URL. This feels like the most RESTful way to continue.

The API integration test should be working now, but it is failing. Need to investigate further next time.

2022-10-29 @ 5:30pm – Need to stop for family dinner.

Total machine time is 4 hours, 30 minutes.

Trouble with persistence test

2022-10-30 @ 10:50am – Wet day in Tauranga, time for some more programming.

Overnight I remembered I should test and extend the endpoints to handle any string for the sensor id. I will try and address this later.

Overnight I thought about how `PersistenceService` is a terrible name for an implementation class, it is probably better as an interface name. I should probably create an interface for persistence and have something like `InMemoryDatastore` for the current implementation that implements that interface. This is probably not top priority though.

Overnight I also think I forgot to commit a change to add the `pytest-mock` dependency.

2022-10-30 @ 11:00am – Added the missing commit for `pytest-mock`.

This is the sort of thing that would have been caught by a well expressed testing stage in a code integration pipeline.

Need to troubleshoot the failing API integration test. I was expecting it to be passing now.

I notice that the API integration test wasn’t sending a sensor id after the refactor to use PATCH.

I was also still sending a POST, not a PATCH, this appears to have got things closer to working.

The API integration test has shown a bug that was covered up by the mocked API contract tests. There is currently no logic that adds the “current” section to the API response. It just returns the data as it was sent to the PATCH endpoint.

I think we ultimately want to have three distinct application layers here: HTTP controllers and persistence (which we have already) and another service layer that is responsible for understanding sensor data. However, that sort of refactoring is best done with all tests passing. I will add some logic into the view first and then use further testing to tease out the remaining behaviour at a future point.

I need to adjust the mocked data on the API contract test for the GET endpoint to address this bug. I should not be including the “current” section in the mocked response. I also shouldn’t add the “data” section either, as that is an HTTP level concern.

2022-10-30 @ 11:25am – The GET endpoint now contains responsibility for formatting the JSON that had incorrectly been pushed to the persistence layer.

The API integration test still doesn’t pass because the PATCH endpoint does not correctly strip the “data” section from the JSON response.

I need to spy on the PersistenceService during the PATCH endpoint and assert that we only get the sensor values, not the “data” section that the HTTP layer should handle.

2022-10-30 @ 11:45am – The PATCH endpoint has been tightened up. Success, all tests are now passing. We finally have the core of our application working.

I guess we only have the “data” section for convention reasons. We have an awful lot of complexity to manage having it there.

There are some pretty significant refactors I think are required, but it is probably prudent to stay focused on the core objectives of the task. A few key remaining

items are top of my mind: adding some attempt at de-duplication, keeping a history of previous sensor configuration changes and merging the current configuration with any updates from the PATCH endpoint instead of simply overwriting them.

The de-duplication will likely inform the other two key deliverables so I probably need to tackle that next.

2022-10-30 @ 12:00pm – Taking a break for some lunch.

Total machine time is 5 hours, 30 minutes.

Adding Configuration History

I just considered that I have made a rather large assumption from the start that the “configuration state” we are dealing with is an update from an IoT sensor/device that is reporting its measurements: “temperature”, “humidity”, “on/off” etc. But perhaps the intention the whole time has been to track the device itself, some metadata about how to access it and what type of device it is: “device name”, “device manufacturer”, “sensor type” etc. I figure I will carry this assumption with me till the end of the exercise. In a real development scenario I would seek clarification before plowing ahead.

2022-10-30 @ 2:45 – Continuing on the exercise. Time to start on the de-duplication.

For any de-duplication logic we will need access to the history of previous sensor configurations.

The history and current configuration are likely to be different once the PATCH endpoint supports partial updates. I should persist the two separately.

To maintain history I will add a second key to the data store for the same sensor identifier. I will maintain a list of all previous configurations recorded separate from the latest/current configuration.

I will first extend the current code to add a suffix to the current persistence id, I will add “-current” to the end. My intention is to later add “-history” as well.

2022-10-30 @ 3:00pm – Getting started on adding the suffix to the existing endpoints.

2022-10-30 @ 3:05pm – Added “-current” suffix.

2022-10-30 @ 3:15pm – Starting to add the “-history” suffix and behaviour.

> Encountered some extra complexity in asserting calls to the persistence layer in unit tests when we call the same method twice with different arguments.

2022-10-30 @ 3:25pm – Learned about `assert_any_call` in pytest-mock. Haven’t used that before.

2022-10-30 @ 3:45pm – Need to describe the behaviour of the persistence layer when a key doesn’t already exist to facilitate the initial push of configuration history.

> I will return `None` when there is no existing data persisted.

2022-10-30 @ 4:00pm – Encountered a problem connected to persisted history being polluted between tests.

> I need to add more mocks to control the unit tests properly to avoid the pollution of state during the tests.

> This will not actually solve the underlying state management problem during tests and I will be forced to confront this further if I add other integration tests.

2022-10-30 @ 4:30pm – Configuration history is built now.

> Total machine time is 7 hours, 15 minutes.

> I have still not completed the exercise, but I have spent more than double the expected time. I will tidy these notes and submit what I have as further iteration seems like it would not better represent my process and what I am capable of.

2022-10-30 @ 5:00pm – Formatting the journal is complete.

> Total machine time is 7 hours, 45 minutes.