Topics in Information Systems and Programming Languages - Mini Project

Dr. Michael Bar-Sinai

Version 1.0, 2021-10-25, BGU CS Dept.

Table of Contents

Project Description

The purpose of this project is to examine how a technology stack affects the design, implementation, and thinking behind a given system. Each group will implement a rather simple system, but using a different stack. At the end of the semester we will all meet, and each group will present its design and discuss how the technology stack they've used affected it.

The system itself is a RESTful server for managing people and tasks. A principle UML class diagram of the models is available below.

IMPORTANT	The RESTful API the system has to implement is detailed here: API specification using OpenAPI/Swagger. Click the API call to see their detailed description, and to try them out (once your system works, that is).

TIP	The above documentation doubles as a front-end for making RESTful HTTP calls and inspecting responses, so you can use it as a front end to test your implementation.

NOTE	The API documentation was created using Swagger, which relies on the OpenAPI specification. The actual OpenAPI file for the above example is available mbarsinai.com/files/bgu/2022a/miniproj/swagger/todosys.yaml[here].

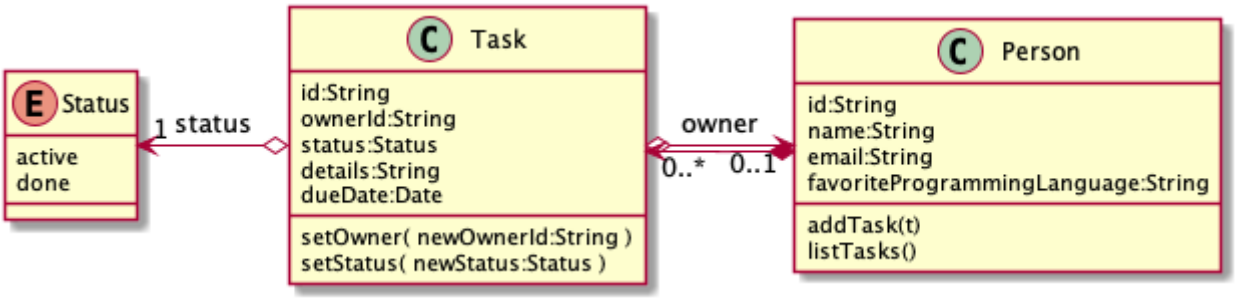

Figure 1. Principal Classes of the system. Actual implementations may of course contain different classes and properties.

Project Points

1. Each task has exactly one person owning it, always.
2. The status of each task is either `done` or `active`.
3. Each person has zero or more tasks.
4. Users should not be able to get the system into any illegal state (a.k.a data corruption).
5. Data should be stored in some permanent storage. That would normally mean some type of database (either relational or non-relational) but other solutions will also be accepted, as long as they can deal with concurrent I/O operations.

Notes on Requirements

- This specification completely ignores authentication (clients need to prove they are known to

the system), authorization (clients can only perform operations they are allowed to do), security concerns (no information is leaked from the system), and auditing (all operations can be tested for compliance in the future). These issues would, of course, be addressed in any real world system - but this is only a mini-project, so we have other fish to fry (see next point).

- As you work, try to note which features of the language/framework make things easy, and which makes things hard.
  - Does your system protects the data it stores from being corrupted? How? Did your technology stack help or made things harder?
  - How did you map entities from JSON to the data store and back? Did your technology stack help? (tip: some stacks offer serious features in these areas, e.g. Play's usage of Scala macros to perform object-json bidirectional mapping at compile time).
  - Reading entities from the data store - how bad was this? Did the language/framework help?
  - Performing summation queries on the data store (here, number of active `Task`s per `Person`) - how horrible was this? Did the language/framework help?

Required Submission

- Working system on some public core repository.
  - This should include low- and high-level documentation, as well as instruction on how to install and run the system,
- 10-15 minutes presentation to the class, detailing:
  - System design. What language, framework, and patterns did you use?
  - Language highlights - what makes it different/interesting? What parts did you like? Which parts did you hate?
    - The above can also include the language culture, e.g. "I found the community of language X to be rather rude and not inviting to new users" or "most tutorials of language Y were very well written".
  - Framework highlights - same as above.
  - Impressions and notes from the working process (see here)
  - Any other points you think would be interesting to discuss in class.

Useful links

- Class site in Moodle
- Mozilla's developer site
- HTTP response status codes @ MDN
- HTTP response status code, demonstrated with cats
- API specification using OpenAPI/Swagger. Note that this documentation doubles as a front-end for making RESTful HTTP calls, so you can use it as a front end to test your implementation.
- `curl` - the swiss army knife of making network calls from the commandline. Works with `HTTP` \ `HTTPS`, but also `DICT`, `FILE`, `FTP`, `FTPS`, `GOPHER`, `GOPHERS`, `IMAP`, `IMAPS`, `LDAP`, `LDAPS`, `MQTT`, `POP3`, `POP3S`, `RTMP`, `RTMPS`, `RTSP`, `SCP`, `SFTP`, `SMB`, `SMBS`, `SMTP`, `SMTPS`, `TELNET` and `TFTP`. You probably want to learn this at some (early) point in your career.
- HTTP Response code 418.