Functional Exam Report 2021

Daniel Skjold Toft - datof16@student.sdu.dk
Mads Lind Larsen - madli19@student.sdu.dk

Introduction

We decided to work with the Orbit-API. It works as a type of synchronizer for files, much like services such as Dropbox or Onedrive. The user is given a client program to use, but underneath the hood it works as a client/server application where all files are kept in sync across an entire system, but still keeping all the files locally on each user's machine. Local files are sent to the server whenever there are changes, and it then updates the files for the other users, a primary functionality is that it only allows for updates to be performed if the file the user is trying to edit is identical to the file the system currently has stored. Thus two users editing a file at the same time could create a synchronization problem, but instead create a conflicting copy for the last person who saves their edit.

Quickcheck port

We worked with the FastCheck[1] framework, which is a Javascript/Typescript port of QuickCheck. It gives us access to the strong types from Typescript and is continually being developed, while also supporting a range of testing frameworks. Thus it seemed an appropriate choice for us to work with and learn. It has a lot of the same capabilities as the OCaml Quickcheck, but of course, with quite a different host language.

Design & implementation

We will briefly describe the overall design and implementation of our solution in the following section, but will not give a complete code-overview.
The full implementation can be found at https://github.com/mads5606/FunctionalExam

Model-based approach

Since the Orbi API is a stateful system, we adopted a model-based approach to test it. To do this we create an abstraction of the system we are testing, using an object oriented approach, where the necessary information is stored in the model together with an array of commands that can be executed on it.

[1] https://github.com/dubzzz/fast-check

```
import fc from "fast-check";
import {IOrbit} from "./Orbit";

export class OrbitModel {
  dirs: DirModel[] = [];
  safeDirs = [17, 18, 15];

  validUsers = [100, 101, 102];
  files: FileModel[] = [];

  constructor() {
    this.seedDirs();
    this.seedFiles();
  }

  findDirById(id: number): DirModel {
    return this.dirs.find(f => f.id == id);
  }

  findFileById(id: number): FileModel {
    return this.files.find(f => f.id == id);
  }
```

Fig 1: Part of the Orbit model

This model is supplemented by an implementation with an interface for the actual Orbit Api:

```
import {RequestResponse} from "./RequestResponse";
import {deleteRequest, getRequest, postRequest, uploadRequest} from "./http";

export interface IOrbit {
  validUser(user: number): boolean;

  validUserList(): number[];

  listFiles(): Promise<RequestResponse>;

  createFile(parentId: number, name: string): Promise<RequestResponse>;

  getFile(id: number): Promise<RequestResponse>;

  uploadFile(id: number, version: number, content: string): Promise<RequestResponse>;

  deleteFile(id: number, version: number): Promise<RequestResponse>;

  listDirs(): Promise<RequestResponse>;

  createDir(parentId: number, parentVersion: number, name: string): Promise<RequestResponse>;

  deleteDir(id: number, version: number): Promise<RequestResponse>;

  updateTimestamp(id: number, version: number, timestamp: number): Promise<RequestResponse>;

  metaCheckOnFileId(id: number): Promise<RequestResponse>;

  metaCheckOnName(dirId: number, fileName: string): Promise<RequestResponse>;
}
```

Fig 2: Interface export for the Orbit API

The actual testing is then done by launching a set of docker containers which the tests are executed with. The reason for launching multiple containers is to allow multiple fresh test runs, since the Orbit API lacks a reset operation that allows the testing to start fresh. Otherwise, a test run would only mean testing one list of operations.

```
import * as fc from "fast-check";
import {OrbitCommands} from "../src/OrbitCommands";
import {OrbitModel} from "../src/OrbitModel";
import {OrbitImpl} from "../src/Orbit";

// https://stackoverflow.com/questions/1880198/how-to-execute-shell-command-in-javascript
const execSync = require('child_process').execSync;

// https://stackoverflow.com/questions/16873323/javascript-sleep-wait-before-continuing
function sleep(milliseconds) {
    var start = new Date().getTime();
    for (var i = 0; i < 1e7; i++) {
        if ((new Date().getTime() - start) > milliseconds) {
            break;
        }
    }
}

const validUsers = [100, 101, 102];

test('model test', async () => {
    execSync('docker-compose up -d --force-recreate', {encoding: 'utf-8'});
    sleep(5000);
    let i = 1;
    await fc.assert(
        fc.asyncProperty(fc.commands(OrbitCommands, 100), async (cmds) => {
            const s = () => ({
                model: new OrbitModel(),
                real: new OrbitImpl(validUsers, {host: "localhost", port: (18080 + i)})
            });
            await fc.asyncModelRun(s, cmds);
            i++;
        }),
        {numRuns: 3, endOnFailure: true, verbose: true}
    );
    execSync('docker-compose down', {encoding: 'utf-8'});
    sleep(5000);
}, 60000);
```

Fig 3: Test Execution

Generators

Since Fast Check natively supports model based testing, we use the native generators of Fast Check. An example of the use of these generators is shown in figure 5 in the appendix section “Use of generators”. Here simple string and integer arbitrary generators are used to generate names and indexes. These indexes are used to choose existing files or directories, and in each command modulo is used to find the remainder, such that the index is never out of bounds. The most interesting generator is actually found in figure 2, where “fc.commands” are used to generate a list of commands based on the Orbit commands implemented in the appendix section.

Shrinkers

Regrettably, we did not have time to properly use the automatic shrinker of Fast Check. Fast Check natively supports model based testing and it also has a shrinker. However, because the Orbit API offers no reset command of the system, resetting the system (multiple times) to allow shrinking is quite time consuming, because the docker container has to be restarted every time. This is a problem we did not prioritize and therefore did not solve.

Properties that we test for

In a nutshell, property based testing frameworks try to discover inputs that cause a property to be false. A property can be described as "for any input such that precondition holds, a property is true".
The goal of this project is to test the combination of all of the commands possible at Orbit, and since the combinations are randomly generated, it would not make sense to note down all of the possible combinations that we hope to test. However, a few examples could be:

- Create file, upload content, download content, upload new content and download content: Both of the times that we download the content from the file, it should be equal to the content most recently uploaded.
- Create directory, create file in directory, move directory, read file: Did the file move with the directory?
- Create file, delete file: Is the file still present on the server?

It would also be possible to do negative testing:

- Create directory, create file in directory, delete directory: Will the server not delete the directory when it is not empty?
- Create file, upload content with illegale characters: Will the server reject the upload request?

And using the commands, simple properties can also be tested:

- Is it possible to create a new directory?
- Is it possible to use the same file name twice?
- What if we use an incorrect file version when uploading content, will the server reject the upload request?

Results

We did not find any concrete bugs in the Orbit API during our testing, but considering that we are only testing the majority of the commands; not all of them, using a single access level, and a single user, this also does seem like an expected result. If any of the functionality we have tested so far was broken, or had bugs in it, then it is the part which is most likely to have been quickly detected by the end-users, or some form of internal testing done during development. The missing coverage is the areas where we begin to get a lot of edge-case like scenarios and conflicting permissions. Adding in the extra users would generate a very large amount of further test-cases since each type of user would need to be tested against all commands, and then the users would need to be tested in consort, doing interleaved operations. Given the nature of the

API, this type of behavior, where people access the same files concurrently and with varying users, is a situation we can imagine would illustrate potential issues and also a critical area for the robustness of the product. A small example of a completed test run is shown in figure 4.

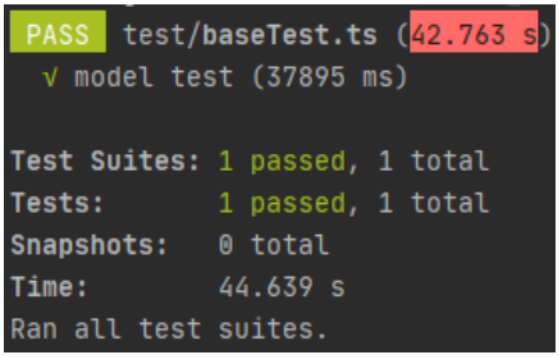

Fig. 4: Test results

Since we found no errors in our test runs, we also experimented with injecting errors in our test to verify their credibility. Some of the errors we tried to inject were changing the versions of files/folders, appending strings to names of files/folders and changing the expected parent id of some files/folders. All of these errors were caught, and it increased our confidence in our implemented tests.

Discussion

The model based testing approach we have used with fast-check aligns a lot with a fuzzing approach to general testing. We are throwing a large amount of inputs, covering the entirety of a subset of the commands, use-cases, and their permutations at the API and looking at its return-values.
Currently the running-time of our setup is rather negligible, if we were to extend the commands to include the few missing ones, and add extra users with varied access-levels then it would undoubtedly increase the duration of the test-runs. But based on our current results this should not be significant enough to be an actual issue. The framework is efficient enough in its execution that it alleviates the added running time of the many permutations we create and test.

Unimplemented tests

As previously mentioned we have cut out a rather significant part of the API, which we have not tested. The testing we have implemented at this stage could mostly serve as an overall proof that a minimal viable product is functioning, and works in a stable manner with single users. We have outlined a set of areas that we would expand our tests to, given more time, and outlined them below.

Current implementation status

We are currently only testing ten out of fourteen of the commands in the Orbit API. The commands we have implemented are the following.

- List checked out files and directories
- Upload
- Delete a file
- Setting file timestamp
- Request meta information on a file
- Structure, list all directories which are checkout out
- Create a directory
- Delete a directory
- Creating a file
- Download a file

Thus we are missing these commands:

- Acquire/release a lock on a file
- Move a directory
- Move a file
- Version check for compatibility

Especially testing the acquire/release lock command would be an interesting inclusion, but we have kept it out of our tests until now, primarily because we concluded that its functionality and testability would not be very significant until additional users have been included in the test.

Additional Users

Our current configuration only includes a single user. Thus we do not test the effects of having different users do commands. The documentation we were provided does not clearly state how the creation of users, and allocation of access-levels, was handled. Thus we have not included this part in our testing and, we generally felt this was outside the scope of our intended testing area. Given more time we would have used the three different users with different access levels that are mentioned in the documentation, but left out other user-handling actions.

Concurrent operations

Once several users are being used, we would ideally not only want to test that they all work independently, but also that if users send concurrent commands they are processed correctly, and not impeding the system functionality in an unintended manner.
This should be done as close to real concurrency as possible, but we imagine that the slightly variable delay in message delivery would make this difficult. However with the scale of the amount of tests the framework executes we believe a test that tries to send commands at the same time will get as close to testing these scenarios as would be needed, and should be enough to provide coverage for those scenarios. If need be statistics could be extracted on

request submission and responses, creating a heuristic to estimate if the tested cases were close enough to be sufficient.
Fast Check also supports Race Condition testing, and it would be quite beneficial to use this for testing concurrent operations, especially since a RESTful API should be able to handle many concurrent operations at any given time.

Interleaved operations

We think this section would end up being almost automatically covered by implementing the previously mentioned sections. But for the sake of completeness, and guaranteeing its inclusion, we would also have liked to explicitly test interleaving operations done by different users, both with the same and varying degrees of access level.

Testing different headers for requests

We currently do not test anything concerning the way requests are sent, header information or protocols. We did not initially even consider this aspect, but became aware of it part-way through the project. We do consider these two to be different scopes, but for a fully encompassing test it should of course be included.

Reflection on the process

We generally found the process of working with our chosen testing framework to be relatively straightforward. We attribute a lot of this to the solid development done on Fast-check, and we think that the wide array of available testing frameworks which implement the quick-check methodology helps with adoption among developers, as they are less likely to be forced into a language they are unfamiliar with.
During our initial work with the framework we started by trying to test some of our own projects, which we had previously created, and the process of learning how to set up the tests and execute them was quite seamless. We were pleasantly surprised with the ease of testing compared to the approaches we had implemented in these projects previously, such as more standard unit and integration testing. We are both expecting to be using some sort of property based testing in the future, as it appears quite powerful. Also, having good frameworks like QuickCheck and Fast Check at hand improves the maintainability and ease-of-use enough to be a feasible solution when testing systems.

How is this different from OCaml and functional programming?

Type- and Javascript can follow the functional programming paradigm, however, we have chosen to utilize the OOP part of Typescript. Therefore all the commands have their own class, the system under test also has their own class, and even an interface. The model also contains several data models beneath the hood, to model the metadata that folder and files have attached to them.

If we were to use OCaml and functional programming, we would probably have used a framework such as QCSTM[2], which is a state-machine testing framework. This framework allows for easy addition of additional commands, and provides a way to handle most of the required components, such as generating, shrinking and setting up. The first thing that should be done is to implement custom types for each command, and then the framework will use pattern matching with these custom types. These pattern matchings are responsible for choosing what each command should do, how the shrinker works, and what preconditions have to be true before a command can be executed. This is very similar to the command objects that we create in Typescript and Fast Check.
However, one of the main reasons for choosing Typescript and Fast Check, was the natively handling of HTTP calls. To test the Orbit API, we will have to use HTTP. Implementing HTTP handling in OCaml can be quite complicated, and therefore we argued that it would make more sense to find a port of QuickCheck that supported HTTP well. Since the metadata around the Orbit API can be quite complicated, the strong type support of Typescript was also quite useful, as this allowed for simple modelling of the data coming out of Orbit.

Conclusion

We believe our project is a solid starting point for doing a complete testing check of the Orbit API by utilising Fast Check. We did not find any bugs in the Orbi API. If the additional parts that we covered in the section regarding unimplemented tests were to be implemented then we believe this would come as close as possible to a fully covering test, as we can get, using Fast Check and the techniques taught to us in the course.

[2] https://github.com/jmid/qcstm

Appendix

Use of generators

```
// Removing "/" because it does not encode or decode correctly. Removing " also,
because it is a lot harder to debug with that in JSON strings
// split-join to "replaceAll"
// Replacement is a and b, for no specific reason, simply removing the two
characters may leave the generated string empty
//const nameArbitrary = fc.string(1, 12).map(s =>
s.split("/").join("a").split('"').join("b"));
const nameArbitrary = fc.base64String(1, 12).map(s => s.split("/").join("a"));

const indexArbitrary = fc.integer(0, 10000);

export const OrbitCommands = [
  fc.tuple(indexArbitrary, nameArbitrary).map(([dirIndex, name]) => new
  createDirCommand(dirIndex, name)),
  fc.tuple(indexArbitrary, nameArbitrary).map(([dirIndex, name]) => new
  createFileCommand(dirIndex, name)),
  indexArbitrary.map(v => new deleteDirCommand(v)),
  indexArbitrary.map(v => new deleteFileCommand(v)),
  indexArbitrary.map(v => new downloadCommand(v)),
  fc.constant(new listCommand()),
  // fc.constant(new lockUnlockFileCommand()),
  indexArbitrary.map(v => new metaCommand(v)),
  indexArbitrary.map(v => new setFileTimeCommand(v)),
  fc.constant(new structureCommand()),
  fc.tuple(indexArbitrary, fc.base64String(0, 100)).map(([fileIndex, content]) =>
   new uploadFileCommand(fileIndex, content)),
  // fc.constant(new versionCommand()),
  // fc.constant(new moveFileCommand()),
  // fc.constant(new moveDirCommand()),
];
```

Fig. 5: Use of native generators. Note the outcommented commands (such as moveFileCommand and moveDirCommand) are currently not implemented.