Sentinel Documentation

Release 0.2.0-SNAPSHOT

Wibowo Arindrarto

November 23, 2016

Contents

Sentinel is a JSON-based database for various next-generation sequencing metrics. It is meant for storing various metrics from various phases of an analysis pipeline run. For a given pipeline run, users upload a JSON file containing the metrics of that pipeline. The metrics can then be queried using one of the predefined HTTP endpoints.

Please use the navigation bar on the left to explore this site.

CHAPTER 1

Introduction

1.1 Why Sentinel

The modern sequencing analysis ecosystem is growing rapidly, both in volume and complexity. An enormous amount of data from various organisms is being generated daily by a plethora of machines. Depending on the research question, the data then must be passed through a specific data analysis pipeline, composed of various tools and (often) ad-hoc scripts. These pipelines usually depend on a number of different external data sources as well, such as genome assemblies and/or annotation files.

In order to properly answer the biological question at hand, a researcher must take into account all of these moving parts. However, grappling with such huge amount of data and variation is not a trivial task. Questions such as 'Is my sequencing run good enough?' or 'How does my sequencing run compare with others?' are often answered only using anecdotal evidence.

To address this issue, we developed Sentinel. Sentinel is a database designed to store various metrics of various sequencing analysis pipeline runs. It provides a systematic way of storing and querying these metrics, with various filter and selection capabilities. We believe that gathering sufficient data points is the first step to make informed decisions about a sequencing experiment.

1.2 At a Glance

Sentinel is implemented as a MongoDB-based database which is exposed through an HTTP RESTful interface. Users upload their sequencing metrics in a single JSON file (which we also call the summary file, since it contains summary of a pipeline run). The uploaded JSON is then parsed and stored in the database.

The structure of the JSON file is very loosely defined. In principle it can be of any form, though Sentinel does require that it conforms to a certain structure (see Describing Run Summaries for the full requirements). Most important is that Sentinel knows how to parse and store the particular JSON file. This entails extending the core Sentinel methods with user-defined parsing code. Sentinel enforces the parsed objects through various interfaces, which in turn makes a wide range of data format compatible for querying.

All uploaded JSON files are only accessible to the uploader and site administrators. The data points contained in the JSON file however, are available to anybody with access to the HTTP endpoints. These data points are anonymized by default. Only after (optional) authentication, can a user see the names of the data points.

CHAPTER 2

Terminologies

In the context of next-generation sequencing, the same words are often used to refer to multiple things. Here we list terms that are used repeatedly in the Sentinel documentation.

2.1 Read Group

A read group denotes a single execution / run of an NGS machine. It may consist of a single sequence file (in the case of single-end sequencing) or two sequence files (paired-end sequencing). Read groups are often used when a single sample needs to be sequenced more than once (e.g. because its sequencing depth is less than desired) or when one sample is sequenced in different lanes.

2.2 Sample

A sample denotes a single experimental unit being investigated. It may be RNA isolated from a single treatment, DNA isolated from a single individual, or something else, depending on the experiment. One sample may be sequenced multiple times (for example when the sequencing depth is inadequate). In this case, the sample would be composed of multiple read groups. It follows that a sample has at least one read group.

2.3 Run

A run is a single data analysis pipeline execution. It may contain one or multiple samples, each possibly containing one or more libraries, depending on the data analysis pipeline.

2.4 Run Summary File

A run summary file is a JSON file that contains metrics of a run. This is the file uploaded to Sentinel. It is up to you/your pipeline to create this file. Our other project, the Biopet pipeline framework, is an example of pipelines that generate such JSON files.

CHAPTER 3

Local Development Setup

3.1 Dependencies

The minimum requirements for a local development environment are:

- git (version >= 1.9)
- Java (version >= 1.8)
- MongoDB (version >= 3.2)

Note that for testing, Sentinel relies on an embedded MongoDB server which it downloads and runs automatically. If you are only interested in running tests or are confident enough not to use any development servers, you can skip MongoDB installation.

For building documentation, you will also need Python Python (version 2.7.x), since we use the Sphinx documentation generator. A complete list of python libraries is listed in the `requirements-dev.txt` file in the root of the project.

While the following packages are not strictly required, they can make your development much easier:

- IntelliJ IDE, with the Scala plugin.
- httpie, a command-line HTTP client for issuing HTTP requests.

And finally, we should note that the current repository contains two packages: `sentinel` for all core methods and `sentinel-lumc` for code specific to our setup in the LUMC. In the future we will most likely separate `sentinel-lumc` out into its own repository.

3.2 Third party libraries

There are several heavily-used libraries that Sentinel depend on. It is a good idea to get familiar with them if you wish to extend Sentinel. These libraries are:

- Json4s, for processing JSON uploads
- Scalaz – particularly the disjunction type (`\/`), to complement the standard library functions.
- *Casbah <https://mongodb.github.io/casbah/>_*, for working the MongoDB backend

3.3 Starting Up

3.3.1 Quick Links

- Source code: https://github.com/lumc/sentinel
- Git: https://github.com/lumc/sentinel.git

3.3.2 On the Command Line (without an IDE)

1. Head over to the command line, clone the repository, and go into the directory:

```
$ git clone https://github.com/lumc/sentinel.git
$ cd sentinel
```

2. If you would like to be able to build documentation, install the python dependencies. We recommend that you use a virtual environment to avoid polluting the root package namespace:

```
$ pip install -r requirements-dev.txt
```

3. If you would like to set up a local development server, make sure MongoDB is running locally on port 27017. You will also need to set up the Sentinel MongoDB users. Sentinel comes with a bash bootstrap script to help you do so. Thes script will set up two MongoDB users by default:

   - `sentinel-owner` (password: `owner`), as the owner of the database.
   - `sentinel-api` (password: `api`), which the Sentinel application uses to connect to the database.

   The script also sets up a Sentinel user with the following details:

   - User ID: `dev`
   - Password: `dev`
   - API key: `dev`

   Remember that these are only meant for development purposes. It is strongly recommended to change these details when you deploy Sentinel.

   The bootstrap script can be run as follows:

```
$ ./scripts/bootstrap_dev.sh
```

4. Start the SBT interpreter via the bundled script. The first setup will take some time, but subsequent ones will be faster:

```
$ ./sbt
```

5. Run the full suite of tests to make sure you are set up correctly. Again, you will start downloading the dependencies if this is your first time:

```
> all test it:test
```

6. If all the tests pass, you are good to go! Otherwise, please let us know so we can take a look. All tests from the default development branch should always pass.

3.3.3 With IntelliJ

Being a Scala-based project, you can use an IDE to develop Sentinel instead of just command line editors. There are numerous IDEs to choose from, but one that we have found to work well is is IntelliJ. You can set up sentinel in IntelliJ following these steps:

1. Head over to the command line, go to a directory of your choice, and clone the repository

```
$ git clone https://github.com/lumc/sentinel.git
```

2. Open IntelliJ, choose `File` -> `New` -> `Project From Existing Source...`
3. Select the location where the project was cloned.
4. Select `Import project from external model` and choose `SBT`. Make sure the Scala plugin is installed first so that the `SBT` option is present.
5. In the dialog box, check the `Use auto-import` check box and select Java 8 for the project JDK. You may choose other checkboxes as well.
6. Click `OK` and wait.

3.4 Using SBT

Sentinel uses SBT to manage its builds. You can use its console to run tasks, or directly from the command line via the bundled *sbt* script.

It comes with many useful tasks, the most-used ones being:

- `compile`: compiles all source files and formats the source code according to the preferences defined in the build file.
- `container:start`: starts development server on port 8080.
- `container:stop`: stops a running development server.
- `browse`: opens a web browser window pointing to the development server.
- `test`: runs all unit tests.
- `it:test`: runs all integration tests.
- `package-site`: creates the Sphinx and ScalaDoc documentation in the `target/scala-2.11` directory.
- `assembly`: creates a JAR with embedded Jetty for deployment in the `target/scala-2.11` directory.
- `assembly-fulltest`: runs all tests (unit and integration) and then creates the deployment JAR.

Note that by default these commands are run for both the `sentinel` and `sentinel-lumc` packages in parallel. If you only want to run it for the `sentinel` package, then the commands must be prefixed with `sentinel/`, for example `test` becomes `sentinel/test`. Alternatively, you can also set the project scope first using the `project sentinel` command. Subsequent commands can then be run on `sentinel` without the prefix.

If you have set up development in IntelliJ, you can also run these commands from inside the IDE. Note however that you may need to unmark the `sentinel/src/test/scala/nl/lumc/sasc/sentinel/exts` directory as test since that may result in some compilation problems. It is usually enough to mark the higher-level ``sentinel/src/test/scala`` as the test source.

You can check the official SBT tutorial to get more familiar with it.

CHAPTER 4

Internal Design Notes

4.1 General Aims

The goal of Sentinel is to enable storing and retrieval of next-generation sequencing metrics as general as possible. It should not be constrained to a specific data analysis pipeline, a specific reference sequence, nor a specific sequencing technology. The challenge here is to have a framework that can be adapted to the need of a lab / institution processing large quantities of such data, when the data analysis pipelines can be so diverse with so many moving parts.

This is why we decided to implement Sentinel as a service which communicates via the HTTP protocol using JSON files. JSON files are essentially free-form, yet it still enforces a useful structure and useful data types which can store the sequencing metrics. Communicating via HTTP also means that we are not constrained to a specific language. A huge number of tools and programming languages that can communicate via HTTP exist today.

4.2 Framework

Sentinel is written in Scala using the Scalatra web framework. Scalatra was chosen since it is has a minimal core allowing us to add / remove parts as we see fit. This does mean that to extend Sentinel, you must be familiar with Scalatra as well.

The API specification is written based on the Swagger specification. It is not the only API specification available out there nor is it an official specification endorsed by the W3C. It seems, however, to enjoy noticeable support from the programming community in general, with various third-party tools and libraries available (at the time of writing). The spec itself is also accompanied by useful tools such as the automatic interactive documentation generator. Finally, Scalatra can generate the specification directly from the code, allowing the spec to live side-by-side with the code.

4.3 Persistence Layer

For the underlying database, Sentinel uses MongoDB. This is in line with what Sentinel is trying to achieve: to be as general as possible. MongoDB helps by not imposing any schema on its own. However, we would like to stress that this does not mean there is no underlying schema of any sort. While MongoDB allows JSON document of any structure, Sentinel does expect a certain structure from all incoming JSON summary files. They must represent a single pipeline run, which contain at least one sample, which contain at least one read group. Internally, Sentinel also breaks down an uploaded run summary file into single samples and potentially single read groups. It is these single units that are stored and queried in the database. One can consider that MongoDB allows us to define the 'schema' on our own, in our own code.

Considering this, we strongly recommend that JSON summary files be validated against a schema. Sentinel uses JSON schema, which itself is JSON, for the pipeline schemas.

4.4 Data Modeling

The following list denotes some commonly-used objects inside Sentinel. Other objects exist, so this is not an exhaustive list.

4.4.1 Controllers

HTTP endpoints are represented as `Controller` objects which subclass from the `SentinelServlet` class. The exception to this rule is the `RootController`, since it implements only few endpoints and is the only controller that returns HTML for browser display. API specifications are defined inside the controllers and is tied to a specific route matcher of an HTTP method.

4.4.2 Processors

Pipeline support is achieved using `Processor` objects, implemented now in the `nl.lumc.sasc.sentinel.processors` package. For a given pipeline, two processors must be implemented: a runs processor, responsible for processing incoming run summary files, and a stats processor, responsible for querying and aggregating metrics of the pipeline. They are the objects that `Controllers` use when talking to the database.

4.4.3 Adapters

Adapters are traits that provide additional database-related functions. For example, the `SamplesAdapter` trait defined in the `nl.lumc.sasc.sentinel.adapters.SamplesAdapter` provides functions required for writing sample-level data to the database. They are meant to extend processors, but in some cases may be instantiated directly.

4.4.4 Extractors

Extractors are traits that read the uploaded JSON files to extract the metrics data contained within. They are also meant to extend processors, specifically run processors, to provide JSON reading functions. The core `sentinel` package provides two base extractors: `JsonExtractor` and `JsonValidationExtractor`. The latter is an extension of `JsonExtractor` that can take a user-defined JSON schema and perform validation based on it.

4.4.5 Records

These objects are more loosely-defined, but most of the time they are case classes that represents a MongoDB object stored in the database. While it is possible to interact with raw MongoDB objects, we prefer to have these objects contained within case classes to minimize run time errors.

CHAPTER 5

The Codebase

Before diving deeper into the code, it is useful to see how the source code is organized.

Starting from the root, we see three directories:

- `project`, where the build definition files are located.
- `scripts`, where helper scripts are located.
- `sentinel` and `sentinel-lumc`, where the actual source code files are located. `sentinel` is meant to contain the sentinel core code, while `sentinel-lumc` contains code specific to LUMC pipelines. Each of these directories contain a directory called `src` that points to the actual source code.

Inside each `src`, we see four more directories. This may look unusual if you come from a Java background, less-so if you are already used to Scala. They are:

- `main`, where the main source files are located.
- `test`, where unit tests are defined.
- `it`, where integration tests are defined.
- `sphinx`, where the raw documentation source files are located.

From here on, you should already get a good grip on the contents of the deeper level directories. Some are worth noting, for reasons of clarity:

- `test/resources` contains all test files and example run summaries used for testing. It is symlinked to `it/resources` to avoid having duplicate testing resources.
- `main/resources` contains run-time resource files that are loaded into the deployment JAR. In most cases, these are pipeline schema files.
- `main/webapp/api-docs` contains a distribution copy of the swagger-ui package. The package is also bundled into the deployment JAR, to help users explore the Sentinel APIs interactively.

CHAPTER 6

Extending Sentinel

Sentinel can be extended with support for capturing metrics of additional pipelines. Adding support for new pipeline metrics can roughly be divided into three steps:

1. Defining the JSON run summary file, preferably with a schema document.
2. Adding the internal objects for the pipeline metrics, which include the runs processors, stats processors, sample and read group records, and other statistics container.
3. Updating the HTTP controllers with new endpoints.

Note: Before adding new pipelines, it is a good idea to familiarize yourself with the project setup, internal data models, and Scalatra first. These are outlined in Internal Design Notes and The Codebase. There is also an internal API documentation (link in the sidebar) for all the internal objects used by Sentinel.

This part of the documentation will walk you through implementing support for a simple pipeline. The pipeline takes one paired-end sequencing file, aligns it to a genome, and calls SNPs on the aligned data. It is meant to be simple as it is meant to highlight the minimum things you need to implement for supporting the pipeline. Later on, you should be able to implement support for your own pipeline easily.

Note: We will not be implementing the actual code for the pipeline itself, rather we will start from after the pipeline has finished running (hypothetically).

We'll start off with a name: since our pipeline is quite simple, let's call it the Minimum (variant calling) Analysis PipeLinE (or *Maple* for short).

CHAPTER 7

Describing Run Summaries

7.1 What Sentinel Expects

In principle, Sentinel accepts any kind of JSON structure. Most important, however, is that a single JSON run summary file contains a full run, with at least one sample containing at least one read group. Usually this means storing the samples as properties of a run object, and read groups as properties of a sample, although you are not limited to this structure.

Note: The exact definitions of samples, read groups, and runs that Sentinel uses are listed in Terminologies.

7.2 What to Store

Having decided on the pipeline name, we need to outline first what the pipeline will store. The following metrics should be simple enough for our purposes:

- Total number of FASTQ reads.
- Total number of reads in the BAM file (mapped and unmapped).
- Number of mapped reads in the BAM file.

We'll also store a name of the pipeline run (which will differ per pipeline run) so we can trace back our runs.

Our hypothetical pipeline can analyze multiple samples and multiple read groups at the same time. It generates a JSON run summary file like this:

```
{
  "runName": "MyRun",
  "samples": {
    "sampleA": {
      "readGroups": {
        "rg1": {
          "nReadsInput": 10000,
          "nReadsAligned": 7500
        }
      },
      "nSnps": 200
    },
    "sampleB": {
      "readGroups": {
```

```
        "rg1": {
          "nReadsInput": 20000,
          "nReadsAligned": 15000
        }
      },
      "nSnps": 250
    }
  }
}
```

Note the nesting structure of the run summary above. We can see that within that single run, there are two samples (`sampleA` and `sampleB`) and each sample contains a read group called `rg1`. Within the read group, we see the actual metrics that we want to store: `nReadsInput` and `nReadsAligned`. On the sample-level, we store the number of SNPs called for that sample in *nSnps*.

You are free to decide on your own structure. Perhaps you don't really care about read-group level statistics, so your pipeline omits them. Or perhaps your pipeline only runs a single sample, so you can put all metrics in the top level. You could also store the samples and/or read groups in an array instead of a JSON object, if you prefer. It is really up to you in the end (everybody has their own way of running these pipelines after all). The important thing is that your soon-to-be-written JSON reader understands the structure.

7.3 Your Pipeline Schema

Writing a schema to validate your run summaries is strongly recommended, though it is not required. Having a schema makes it easier to check for run time errors and prevent incorrect data from being processed. Sentinel uses the JSON schema specifications to define run summary schemas. You can head over to their site to see the full specification.

For our *Maple* pipeline, we'll use the schema already defined below. Save this as a file in the `src/main/resources/schemas` directory with the name `maple.json`.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Maple pipeline schema",
  "description": "Schema for Maple pipeline runs",
  "type": "object",
  "required": [ "samples", "runName" ],

  "properties": {

    "run_name": { "type": "string" },

    "samples": {
      "description": "All samples analyzed in this run",
      "type": "object",
      "minItems": 1,
      "additionalProperties": { "$ref": "#/definitions/sample" }
    }
  },

  "definitions": {

    "sample": {
      "description": "A single Maple sample",
      "type": "object",
      "required": [ "readGroups", "nSnps" ],
```

```
      "properties": {

        "readGroups": {
          "description": "All read groups belonging to the sample",
          "type": "object",
          "minItems": 1,
          "additionalProperties": { "$ref": "#/definitions/readGroup" }
        },

        "nSnps": {
          "description": "Number of SNPs called",
          "type": "integer"
        }
      }
    },

    "readGroup": {
      "description": "A single Maple readGroup",
      "type": "object",
      "required": [ "nReadsInput", "nReadsAligned" ],

      "properties": {
        "nReadsInput": { "type": "integer" },
        "nReadsAligned": { "type": "integer" }
      }
    }
  }
}
```

If the above code looks daunting, don't worry. You can copy-paste the code as-is and try to understand the JSON schema specifications later on. If you want to play around with the schema itself, there is an online validator available here. You can copy-paste both the JSON summary and JSON schema examples above there and try tinkering with them.

CHAPTER 8

Defining the Data Models

Having created the schema, let's now move on to implementing the processors. We will write two processors, the runs processor for processing the raw run summaries, and the stats processor for querying the metrics. Before that, we will first write the internal models for our samples, libraries, and the statistics containers.

We will put all of them inside the `nl.lumc.sasc.sentinel.processors.maple` package, since everything will be specific for the *Maple* pipeline support.

Note: Since we will be using the internal models for this part, it is useful to browse the ScalaDoc along the way. Link to the most recent ScalaDoc is available in the sidebar.

To start off, we first consider the types of object we need to define:

- For the run itself, we'll define a `MapleRunRecord` that subclasses `nl.lumc.sasc.sentinel.models.BaseRunRecord`.
- For the samples, we'll define `MapleSampleRecord` that subclasses `nl.lumc.sasc.sentinel.models.BaseSampleRecord`.
- Likewise, for the library, we'll define `MapleReadGroupRecord` subclassing `nl.lumc.sasc.sentinel.models.BaseReadGroupRecord`.
- And finally, for the statistics, we'll define `MapleStats` for the single data points and `MapleStatsAggr` for aggregated data points.

The definitions of these objects are outlined below. Note that while we are defining these objects once per file, you have the freedom to create them in one large file. The important thing is they have the correct package name (`nl.lumc.sasc.sentinel.maple` in this case).

8.1 MapleRunRecord

Let's start with the first one: `MapleRunRecord`. Open a `MapleRunRecord.scala` file in the appropriate directory and add the following contents (you can use your own package name, if you prefer):

```
1 package nl.lumc.sasc.sentinel.exts.maple
2
3 import java.util.Date
4
5 import org.bson.types.ObjectId
6
7 import nl.lumc.sasc.sentinel.models._
```

```
8  import nl.lumc.sasc.sentinel.utils.utcTimeNow
9
10 /** Container for a Maple run. */
11 case class MapleRunRecord(
12   runId: ObjectId,
13   uploaderId: String,
14   pipeline: String,
15   sampleIds: Seq[ObjectId],
16   readGroupIds: Seq[ObjectId],
17   runName: Option[String] = None,
18   deletionTimeUtc: Option[Date] = None,
19   creationTimeUtc: Date = utcTimeNow) extends BaseRunRecord
```

From the definition above, you can already notice a few properties :

1. Our run record stores most of its IDs as `ObjectId`, which is the default ID type for MongoDB databases. The uploader ID is kept as a `String` for later use.
2. We also store the date when the record is created in `creationTimeUtc`. We use the `utctTimeNow` function from the `utils` package to get the current UTC time.
3. There is also a `deletionTimeUtc` attribute that stores when the record is deleted. The default is set to `None`, since when an object is created it is not yet deleted.

8.2 MapleSampleRecord

Now let's move on to the sample record definition. In the same file, add the following `MapleSampleRecord` definition:

```
1  /** Container for a single Maple sample. */
2  case class MapleSampleRecord(
3    stats: MapleSampleStats,
4    uploaderId: String,
5    runId: ObjectId,
6    sampleName: Option[String] = None,
7    runName: Option[String] = None) extends BaseSampleRecord
```

In contrast to `MapleRunRecord`, our sample record can be quite short since it needs to store less information. The actual metrics itself will be stored in a yet-defined `MapleSampleStats` object, under the `stats` attribute. The name `stats` itself is free-form, you are free to choose the attribute name for your metrics object. You can even define multiple attributes storing different statistics. This is useful for storing different types of metrics on the same level, for example storing alignment metrics and variant calling metrics for a given sample.

Notice also that there is no `deletionTimeUtc` attribute. This is because when sample records are removed from the database, Sentinel removes it completely and does not keep a record of which samples are removed. This is mainly because Sentinel never shows the sample document in the HTTP interface, so it is free to add and remove samples. The run record, on the other hand, are shown to users, and sometimes it is useful to keep track of ones that have been deleted.

Finally, notice that now we store the sample name under `sampleName` in addition to the run name.

8.3 MapleReadGroupRecord

Next up, is the read group record:

```
/** Container for a single Maple read group. */
case class MapleReadGroupRecord(
  stats: MapleReadGroupStats,
  uploaderId: String,
  runId: ObjectId,
  isPaired: Boolean = true,
  readGroupName: Option[String] = None,
  sampleName: Option[String] = None,
  runName: Option[String] = None) extends BaseReadGroupRecord
```

This is almost similar to `MapleSampleRecord`, except:

1. There is an attribute called `isPaired`, which as you can guess, denotes whether the library comes from paired-end sequencing or not. Since *Maple* handles paired-end files, we can set this definition by default to `true`.
2. There is an additional name attribute: `readGroupName`, for storing the read group name.

8.4 Statistics container

Finally, we come to the definition of our actual metrics container. Since we store the metrics on two levels, sample and read group, we need to define the metrics container for each of these levels. This is what they look like:

```
/** Container for a single Maple sample statistics. */
case class MapleSampleStats(
  nSnps: Long,
  labels: Option[DataPointLabels] = None) extends LabeledStats

/** Container for a single Maple read group statistics. */
case class MapleReadGroupStats(
  nReadsInput: Long,
  nReadsAligned: Long,
  labels: Option[DataPointLabels] = None) extend LabeledStats
```

For each level, we define a case class that extends `LabeledStats`. This trait enforces the use of the `labels` attribute to tag a particular metrics data point with labels. For any given data point, it must at least be labeled with the database ID of the run record (`runId`). Optionally, it may also be labeled with the run name, read group name and/or sample name. All this is contained within the `DataPointLabels` instance stored in the `labels` attributed.

The objects defined above stores single data points of our metrics. They are instantiated for each sample or read group that is present in the uploaded JSON summary file. We enforce the use of a case class here based on several reasons:

1. To minimize potential runtime errors, since the case class ensures our stored metrics are all typed. The type information is also used to ensure user-defined metrics works well with the Sentinel core methods.
2. Case classes play nicely with Swagger's automatic API spec generation. Supplying these as type parameters in our controllers later on results in Swagger generating the JSON object definitions.

In addition to the two case classes defined above, we may also want to define the following case classes for storing aggregated data points instead of single data points:

```
/** Container for aggregated Maple sample statistics. */
case class MapleSampleStatsAggr(nSnps: DataPointAggr)

/** Container for aggregated Maple read group statistics. */
case class MapleReadGroupStatsAggr(
  nReadsInput: DataPointAggr,
  nReadsAligned: DataPointAggr)
```

You'll notice that these are almost similar to the previous case classes, except:

1. All the attribute types are `DataPointAggr`.
2. There are no labels anymore.

The `DataPointAggr` is another case class that contains aggregated statistics like *avg*, *max*, or *median*. It is likely that we will use macros to generate these in future Sentinel versions, since they are very similar to the case classes that define the single data points.

That concludes our first part of the processors tutorial! Now we can move on the the actual implementation of the processors. Before you go on, however, we would like to note that the processors make use of Scalaz's disjunction type (popularly known as `\/`), its `EitherT` type, and the standard library `Future` type. If these do not sound familiar, we strongly recommend that you go over our short guides on them first: Composable Error Handling and Asynchronous Processing. Otherwise, feel free to go to the processors tutorial: Creating the Processors directly.

CHAPTER 9

Composable Error Handling

Sentinel, being a framework that interacts with a database, faces common problems. It needs to validate user-supplied data, write and read from the database, and so on. These are not new problems and various other frameworks and/or languages have solved it in their own way.

Since Sentinel is based on Scala, we'll take a look at how we can actually achieve these things nicely in Scala. It may look daunting at first, but the result is worth the effort: composable clean code.

We'll cover three topics in particular:

1. Dealing expected failures such as user-input errors, in this guide.
2. Launching task asynchronously using `Future`, in the next guide: Asynchronous Processing.
3. How Sentinel composes error handling asynchronously, also in the next guide: Asynchronous Processing.

Note: This guide is geared towards use cases in Sentinel and is by no means comprehensive. Readers are expected to be familiar with Scala, particularly using for comprehensions and pattern matching. It is also a good idea to be familiar with the scalaz library as Sentinel also makes considerable use of it.

9.1 Dealing with expected failures

Upon receiving a file upload, Sentinel parses the contents into a JSON object and extracts all the metrics contained within. This means that Sentinel needs to ensure that:

1. The file is valid JSON.
2. The JSON file contains the expected metrics values.

If any of these requirements are not met, Sentinel should notify the uploader of the error since this is something that we can expect the uploader to correct.

Let's assume that the uploaded data is stored as a `Array[Byte]` object and the data is parsed into a `JValue` object, as defined by the Json4s library. We can use Json4s's `parse` function to extract our JSON. It has the following (simplified) signature:

```
// `JsonInput` can be many different types,
// `java.io.ByteArrayInputStream` among them.
// `Formats` is a Json4s-exported object that
// determines how the parsing should be done.
def parse(in: JsonInput)(implicit formats: Formats): JValue
```

We can then write a function to extract JSON out of our byte array using `parse`.

```
import java.io.ByteArrayInputStream
import org.json4s.JValue
import org.json4s.jackson.JsonMethods.parse

// For now, we can use the default `Formats`
// as the implicit value
implicit val formats = org.json4s.DefaultFormats

// first attempt
def extractJson(contents: Array[Byte]): JValue =
    parse(new ByteArrayInputStream(contents))
```

In an ideal world, our function would always return a `JValue` given a byte array. In reality, our function will be faced with user inputs which should be treated with extreme caution. That includes expecting corner cases, for example if the byte array is not valid JSON or if the byte array is empty. Those cases will cause `parse` to throw an exception that must be dealt with, otherwise our normal program flow is interrupted.

9.2 The `Option` type

How should we handle the exception? Having a strong functional flavour, Scala offers a nice alternative using a type called the `Option[T]` type. The short gist is that it allows us to encode return types that may or may not exist. It has two concrete values: `Some(_)` or `None`. For example, if a function has return type `Option[Int]` it can either be of value `Some(3)` (which means it has the value 3) or `None` (which means no number was returned).

While exceptions may be more suitable for some cases, `Option` offers interesting benefits of its own. With `Option`, we explicitly state the possibility that a function may not return its intended value in its very signature. Consequently, this informs any caller of the function to deal with this possibility, making it less prone to errors.

It turns out also that the `Option` pattern occurs frequently in code. Functions that perform integer division, for example, needs to acknowledge the fact that division by zero may occur. Another example is functions that return the first item of an array. What should the function do when the array is empty? `Option` fits well into these and many other cases. Over time, this has resulted in a set of common operations that can be applied to `Option` objects that we can use to make our code more concise without sacrificing functionality. You can check out the official documentation for a glance of what these operations are.

Tip: For a more in depth treatment of `Option`, we find the guide here informative.

Some operations worth highlighting are `flatMap` and `withFilter`. They are used by Scala's for-comprehension, which means you can chain code that returns `Option` like this:

```
def alpha(): Option[Int] = { ... }
def beta(n: Int): Option[String] = { ... }
def gamma(s: String): Option[Double] = { ... }

val finalResult: Option[Double] = for  {
    result1 <- alpha()
    result2 <- beta(result1)
    result3 <- gamma(result2)
    squared = result3 * result3
} yield squared
```

In the code snippet above, `alpha` is called first, then its result is used for calling `beta`, whose result is used for calling `gamma`. The beauty of the chain is that if any of the functions return `None`, subsequent functions will not

be called and we get `None` as the value of `finalResult`. There is no need to do manual checks using `if` blocks. Furthermore, the for comprehension automatically unwraps `result1` and `result2` out of `Option` when used for calling `beta` and `gamma`. Finally, we can slip an extra value declaration (`squared`) which will only work if our chain produces an expected `result3` value.

Tip: `flatMap` and `withFilter` are not the only methods that for comprehensions desugars into. Check out the official FAQ on other possible methods.

Going back to our JSON extractor function, we need to update it so that it returns `Option[JValue]`. Luckily, Json4s also already has us covered here. In addition to `parse`, it also provides a function called `parseOpt` which only returns a `JValue` if the given input can be parsed into JSON. It has the following (simplified) signature:

```
def parseOpt(in: JsonInput)(implicit formats: Formats): Option[JValue]
```

Our function then becomes:

```
import java.io.ByteArrayInputStream
import org.json4s.JValue
import org.json4s.jackson.JsonMethods.parseOpt

implicit val formats = org.json4s.DefaultFormats

// second attempt
def extractJson(contents: Array[Byte]): Option[JValue] =
    parseOpt(new ByteArrayInputStream(contents))
```

9.3 The `Either` type

Our function now has a better return type for any of its caller. Notice however, that `Option` is black and white. Either our function returns the expected `JValue` or not. In contrast, there are more than one way that the parsing can fail. The JSON file could be malformed, maybe containing an extra comma or missing a bracket somewhere. There could also be network errors that cause no bytes to be uploaded, resulting in an empty byte array. These are information that is potentially useful for uploaders, so it would be desirable for Sentinel to be able to report what kind of error causes the parsing to fail. In short, we would like to encode the possibility that our function may fail in multiple ways.

Enter the `Either` type. `Either` allows us to encode two return types into one, unlike `Option` which only allows one. Its two concrete values are either `Right`, conventionally used to encode the type returned for successful function calls, and `Left` for encoding errors.

This should be clearer with an example. We will use a function that returns the sum of the first and last item of a `List[Int]` to illustrate this. Here, the given list must contain at least two items. If that's not the case, we would like to notify the caller. One way to write this with `Either` is like so:

```
def sumFirstLast(list: List[Int]): Either[String, Int] =
    if (list.isEmpty) Left("List has no items.")
    else if (list.size == 1) Left("List only has one item.")
    else Right(list.head + list.last)
```

The type that encodes the error (the left type) can be anything. We use `String` here for convenience, but other types such as `List[String]` or even a custom type can be used.

We can now further improve our `extractJson` function using `Either`. Since Json4s does not provide a parsing function that returns `Either`, we need to modify our own function a bit:

```
import java.io.ByteArrayInputStream
import org.json4s.JValue
import org.json4s.jackson.JsonMethods.parseOpt

implicit val formats = org.json4s.DefaultFormats

// third attempt
def extractJson(contents: Array[Byte]): Either[String, JValue] =
    if (contents.isEmpty) Left("Nothing to parse.")
    else parseOpt(new ByteArrayInputStream(contents)) match {
        case None     => Left("Invalid syntax.")
        case Some(jv) => Right(jv)
    }
```

9.4 The disjunction type: \/

Our iterations are looking better, but we are not there yet. `Either`, as provided by the Scala standard library, unfortunately does not play very well with for comprehensions like `Option` does. Scala does not enforce that `Either`'s `Left` encodes the error return type (and consequently, that `Right` encodes the succes type). What this means is that in for comprehensions, we have to tell whether we expect the `Right` or `Left` type for each call. This is done by calling the `Either.right` or `Either.left` method.

```
def uno(): Either[String, Int] = { ... }
def dos(n: Int): Either[String, String] = { ... }
def tres(s: String): Either[String, Double] = { ... }

val finalResult: Either[String, Double] = for {
    result1 <- uno().right
    result2 <- dos(result1).right
    result3 <- tres(result2).right
} yield result3
```

It seems like a minor inconvenience to add `.right`, but there is something going on under the hood with `.right` and `.left`. They do not actually create `Right` and `Left`, but `RightProjection` and `LeftProjection`, which is a different type with different properties. The practical consequence is that the code below will not compile anymore (unlike its `Option` counterpart):

```
val finalResult: Either[String, Double] = for {
    result1 <- uno().right
    result2 <- dos(result1).right
    result3 <- tres(result2).right
    squared = result3 * result3
} yield squared
```

To get it working, we need to manually wrap the `squared` declaration inside an `Either`, invoke `.right`, and replace the value assignment operator:

```
val finalResult: Either[String, Double] = for {
    result1 <- uno().right
    result2 <- dos(result1).right
    result3 <- tres(result2).right
    squared <- Right(result3 * result3).right
} yield squared
```

This is getting unecessarily verbose. We have to invoke `.right` every time and we lose the ability to declare values inside for comprehensions. To remedy this, we need to use the scalaz library.

Scalaz is a third party Scala library that provides many useful functional programming abstractions. One that we will use now is called `\/` (often called the disjunction type, since it is inspired by the mathematical disjunction operator). It is very similar to `Either`, except for the fact that it is right-biased. This means, it expects the error type to be encoded as the left type and the expected type to be encoded as the right type.

Here is a quick comparison between `Either` and `\/`:

```
import scalaz._, Scalaz._

// Type declaration.
// We can use the `\/` type as an infix
// operator as well, as shown in `value3`
// declaration below
def value1: Either[String, Int] // Scala
def value2: \/[String, Int]     // scalaz
def value3: String \/ Int       // scalaz

// Right instance creation.
// The scalaz constructor is the type name,
// plus the side we use: `\/` appended with `-`
val value4: Either[String, Int] = Right(3) // Scala
val value5: String \/ Int       = \/-(3)   // scalaz

// Left instance creation.
// The scalaz constructor is analogous to its
// right type counterpart: `\/` prepended with `-`
val value6: Either[String, Int] = Left("err") // Scala
val value7: String \/ Int       = -\/("err")  // scalaz
```

Our earlier example can now be made more concise using the disjunction type:

```
def uno(): String \/ Int = { ... }
def dos(n: Int): String \/ String = { ... }
def tres(s: String): String \/ Double = { ... }

val finalResult: String \/ Double = for {
    result1 <- uno()
    result2 <- dos(result1)
    result3 <- tres(result2)
    squared = result3 * result3
} yield squared
```

One more thing: notice that we always encode the error type / left type as `String` and we need to redeclare it every time. We can make this even shorter by creating a type alias to disjunction whose left type is always `String`. Let's call this alias `Perhaps`:

```
type Perhaps[+T] = String \/ T

def uno(): Perhaps[Int] = { ... }
def dos(n: Int): Perhaps[String] = { ... }
def tres(s: String): Perhaps[Double] = { ... }

val finalResult: Perhaps[Double] = for {
    result1 <- uno()
    result2 <- dos(result1)
    result3 <- tres(result2)
    squared = result3 * result3
} yield squared
```

And finally, going back to our JSON extractor example, we need to update it like so:

```
import java.io.ByteArrayInputStream
import org.json4s.JValue
import org.json4s.jackson.JsonMethods.parseOpt
import scalaz._, Scalaz._

implicit val formats = org.json4s.DefaultFormats

type Perhaps[+T] = String \/ T

// fourth attempt
def extractJson(contents: Array[Byte]): Perhaps[JValue] =
    if (contents.isEmpty) -\/("Nothing to parse.")
    else parseOpt(new ByteArrayInputStream(contents)) match {
        case None     => -\/("Invalid syntax.")
        case Some(jv) => \/-(jv)
    }
```

Going even further, we can replace the pattern match with a call to scalaz's `.toRightDisjunction`. This can be done on the `Option[JValue]` value that `parseOpt` returns. The argument is the error value; the value that we would like to return in case `parseOpt` evaluates to `None`.

```
...

// fourth attempt
def extractJson(contents: Array[Byte]): Perhaps[JValue] =
    if (contents.isEmpty) -\/("Nothing to parse.")
    else parseOpt(new ByteArrayInputStream(contents))
        .toRightDisjunction("Invalid syntax.")
```

We can further shorter this using the `\/>` function, which is basically an alias to `.toRighDisjunction`:

```
...

// fourth attempt
def extractJson(contents: Array[Byte]): Perhaps[JValue] =
    if (contents.isEmpty) -\/("Nothing to parse.")
    else parseOpt(contents) \/> "Invalid syntax."
```

This is functionally the same, and some would prefer the clarity of `.toRightDisjunction` instead of `\/>`'s brevity. We will stick to `.toRighDisjunction` for now.

9.5 Comprehensive value extraction

We did not use any for comprehensions in `extractJson`, though, so why did we bother to use `\/` at all? Remember that creating the JSON object is only the first part of our task. The next part is to extract the necessary metrics from the created JSON object. At this point it is still possible to have a valid JSON object that does not contain our expected metrics.

Let's assume that our expected JSON is simple:

```
{
    "nSnps": 100,
    "nReads": 10000
}
```

There are only two values we expect, `nSnps` and `nReads`. Using Json4s, extracting this value would be something like this:

```
// `json` is our parsed JSON
val nSnps: Int = (json \ "nSnps").extract[Int]
val nReads: Int = (json \ "nReads").extract[Int]
```

We can also use `.extractOpt` to extract the values into an `Option` type:

```
// By doing `.extractOpt[Int]`, not only do we expect
// `nSnps` to be present, but we also check that it is
// parseable into an `Int`.
val nSnps: Option[Int] = (json \ "nSnps").extractOpt[Int]
val nReads: Option[Int] = (json \ "nReads").extractOpt[Int]
```

Now let's put them together in a single function. We'll also create a case class to contain the results in a single object as well. Since we are doing two extractions, it's a good idea then to use the disjunction type instead of `Option` so that we can see if any error occurs.

```
...

case class Metrics(nSnps: Int, nReads: Int)

def extractMetrics(json: JValue): Perhaps[Metrics] = for {
    nSnps <- (json \ "nSnps")
        .extractOpt[Int]
        .toRightDisjunction("nSnps not found.")
    nReads <- (json \ "nReads")
        .extractOpt[Int]
        .toRightDisjunction("nReads not found.")
    metrics = Metrics(nSnps, nReads)
} yield metrics
```

Both extraction steps now combine nicely in one for comprehension. The code is concise and we can still immediately see that both `nSnps` and `nReads` must be present in the parsed JSON object. If any of them is not present, an error message will be returned appropriately.

What's even nicer, is that `extractMetrics` compose well with our previous `extractJson`. We can now write one function that does both:

```
...

def processUpload(contents: Array[Byte]): Perhaps[Metrics] = for {
    json <- extractJson(contents)
    metrics <- extractMetrics(json)
} yield metrics
```

That's it. Our `processUpload` function extracts JSON from a byte array and then extracts the expected metrics from the JSON object. If any error occurs within any of these steps, we will get the error message appropriately. If we ever want to add additional steps afterwards (maybe checking if the uploaded metrics is already in a database or so), we can simply add another line in the for comprehension so long as our function call returns a `Perhaps` type.

9.6 Sentinel's Error Type

While `String` is a useful error type in some cases, in our cases it is not exactly the most suitable type for errors. Consider a case where our uploaded JSON does not contain both `nSnps` and `nReads`. In that case, the user would first get an error message saying 'nSnps not found.'. Assuming he/she fixes the JSON by only adding `nSnps`, he/she would then get another error on the second attempt, saying 'nReads not found.'. This should have been displayed on the first upload, since the error was already present then.

This approach of failing on the first error we see (often called failing fast) is then not exactly suitable for our `extractMetrics` function. Another approach where we accumulate the errors first (failing slow) before displaying them seems more appropriate. To do so, we need to tweak our error type to be a `List[String]` instead of the current `String`. We can then add error messages to the list and return it to the user eventually.

It's only for `extractMetrics`, though. We would still like to fail fast in `extractJson` as both errors we expect to encounter there can not occur simultaneously. If the JSON file is empty, it must not contain any syntax errors and vice versa.

Sentinel reconciles this by having a custom type for its error type, called the `ApiPayload`. It is a case class that contains both `String` and `List[String]`. The `ApiPayload` type is also associated with specific HTTP status codes. This is because the error messages that Sentinel displays must be sent via HTTP and thus must be associated with a specific code.

Its simplified signature is:

```
// `httpFunc` defaults to a function
// that returns HTTP 500
sealed case class ApiPayload(
    message: String,
    hints: List[String],
    httpFunc: ApiPayload => ActionResult)
```

The idea here is that we always have a single error message that we want to display to users (the `message` attribute). Accumulated errors can be grouped in `hints`, if there are any. We also associate a specific error message with a specific HTTP error code in one place.

Note: Being based on the Scalatra framework, Sentinel uses Scalatra's ActionResult to denote HTTP actions. Scalatra already associates the canonical HTTP status message with the error code (for example `InternalServerError` has the 500 code). Check out the Scalatra documentation if you need more background on `ActionResult`.

Additionally, `ApiPayload` objects are transformed into plain JSON that are then sent back to the user. The JSON representation displays only `message` and `hints`, since `httpFunc` is only for internal Sentinel use.

An example of an `ApiPayload` would look something like this:

```
// `BadRequest` is Scalatra's function
// that evaluates to HTTP 400.
val JsonInputError = ApiPayload(
    message = "JSON input can not be parsed.",
    hints = List("Input is empty."),
    httpFunc = (ap) => BadRequest(ap))
```

It can get a bit tedious, as you can see. Some HTTP error messages occur more frequently than others, fortunately, so Sentinel already creates some predefined `ApiPayload` objects that you can use. They are all defined in `nl.lumc.sasc.sentinel.models.Payloads`.

In our case, we can use `JsonValidationError`. It is always associated with HTTP 400 and its `message` attribute is hard coded to "JSON is invalid.". We only need to supply the hints inside a `List[String]`. Moreover, our disjunction type `ApiPayload \/ T` is also already defined by sentinel in `nl.lumc.sasc.sentinel.models.Perhaps`, so we can use that.

Let's now update our functions to use `ApiPayload` (along with some style updates). We will also outline how far we have written our functions:

```
// We import a mutable list for collecting our errors
import collection.mutable.ListBuffer
import java.io.ByteArrayInputStream
```

```
import org.json4s.JValue
import org.json4s.jackson.JsonMethods.parseOpt
import scalaz._, Scalaz._
import nl.lumc.sasc.sentinel.models.{ Payloads, Perhaps }, Payloads._

implicit val formats = org.json4s.DefaultFormats

case class Metrics(nSnps: Int, nReads: Int)

// Our change here is mostly to replace
// `String` with `ApiPayload`.
def extractJson(contents: Array[Byte]): Perhaps[JValue] =
    if (contents.isEmpty) {
        val hints = JsonValidationError("Nothing to parse.")
        -\/(hints)
    } else {
        val stream = new ByteArrayInputStream(contents)
        val hints = JsonValidationError("Invalid syntax.")
        parseOpt(input).toRightDisjunction(hints)
    }

// This is where most our changes happen
def extractMetrics(json: JValue): Perhaps[Metrics] = {

    val maybe1 = (json \ "nSnps").extractOpt[Int]
    val maybe2 = (json \ "nReads").extractOpt[Int]

    (maybe1, maybe2) match {
        // When both values are defined, we can create
        // our desired return type. Remember we need
        // to wrap it inside `\/` still.
        case (Some(nSnps), Some(nReads)) =>
            \/-(Metrics(nSnps, nReads))
        // Otherwise we further check on what's missing
        case otherwise =>
            val errors: ListBuffer[String] = ListBuffer()
            if (!maybe1.isDefined) errors :+ "nSnps not found."
            if (!maybe2.isDefined) errors :+ "nReads not found."
            -\/(JsonValidationError(errors.toList))
    }
}

// This function remains the same.
def processUpload(contents: Array[Byte]): Perhaps[Metrics] = for {
    json <- extractJson(contents)
    metrics <- extractMetrics(json)
} yield metrics
```

And there we have it. Notice that even though we fiddled with the internals of `extractJson` and `extractMetrics`, our `processUpload` function stays the same. This is one of the biggest wins of keeping our API stable. Our functions all follow the pattern of accepting concrete values and returning them wrapped in `Perhaps`. This is all intentional, so that we can keep `processUpload` clean and extendable.

9.6.1 Fitting the JSON Schema in

Our `extractMetrics` function looks good now, but notice that it is already quite verbose even for a small JSON. This is why we recommend that you define JSON schemas for your expected summary files. Sentinel can then validate

based on that schema, accumulating all the errors it sees.

The Sentinel validation function is called `validateJson`, which has the following signature:

```
def validateJson(json: JValue): Perhaps[JValue]
```

You can see that it expects as its input a parsed JSON object. This means that we need to create a JSON object first before we validate it. To this end, Sentinel also provides an `extractJson` function. Its signature is the same as the `extractJson` function you have been writing. We can then combine extraction and validation together in one function like so:

```
def extractAndValidateJson(contents: Array[Byte]): Perhaps[JValue] =
    for {
        json <- extractJson(contents)
        validatedJson <- validateJson(json)
    } yield validatedJson
```

Sentinel provides `extractAndValidateJson` as well. In fact, that is also how Sentinel composes JSON extraction and JSON validation internally: using a single for comprehension.

9.7 Next Steps

We hope we have convinced you that encoding errors as the return type instead of throwing exceptions can make our code cleaner and more composable. In the next section, Asynchronous Processing, we will be combining our `Perhaps` type with Scala's `Future` so that we can process data asynchronously.

CHAPTER 10

Asynchronous Processing

Having dealt with handling expected errors, we will now take a short tour on how Sentinel does asynchronous processing. The basic rationale of using asynchronous processing in Sentinel is to ensure that multiple HTTP requests can be handled at the same time. A single HTTP request may involve writing and/or reading from the database several times, so most asynchronous functions in Sentinel are database-related operations.

Note: This guide assumes readers are already familiar with the `Future` type and attempts only to explain how it is used in Sentinel. For more comprehensive `Future` guide, we find the official overview a good starting point.

Sentinel uses the `Future` type defined in the standard library for its asynchronous functions. In some ways, this allows us to leverage the type system to make our functions composable, similar to how we handled errors earlier using the disjunction type `\/`.

10.1 `Future` with for comprehensions

Like the `Option`, `Either`, and `\/` types we have seen previously, `Future` is composable. You can, for example, use it inside a for comprehension:

```
import scala.concurrent._

// Remember that we need an execution context
implicit val context = ExecutionContext.global

def calc1(): Future[Int] = Future { ... }
def calc2(): Future[Int] = Future { ... }
def calc3(x: Int, y: Int) = Future[Int] = Future { ... }

// Launch separate computation threads
val value1 = calc1()
val value2 = calc2()

val finalResult = for {
    firstResult <- value1
    secondResult <- value2
    thirdResult <- calc3(firstResult, secondResult)
} yield thirdResult
```

The code block above will start a computation for `value1` and `value2` in parallel threads and wait until they both return their results, before using them as arguments for `calc3`.

The code also illustrates how using `Future` with for comprehensions require a little more care. Notice that we invoked `calc1` and `calc2` **outside** of the for comprehension block. This is intentional and the reason is because function calls inside the block are sequential. Had we written `finalResult` like this:

```
...

val finalResult = for {
    firstResult <- calc1()
    secondResult <- calc2()
    thirdResult <- calc3(firstResult, secondResult)
} yield thirdResult
```

then it does not matter if we write our code inside a `Future`. It will be invoked sequentially, defeating the purpose of using `Future` in the first place.

In some cases the for comprehension does allow for a value declaration inside itself (i.e. using the = operator instead of <-). This requires that the first statement inside the block is an <- statement using the abstract type the block intends to return. Since we are using `Future`, this means the first statement should be a <- statement from a `Future[_]` type. Take the following example:

```
...

def calc0(): Future[Unit] = Future { ... }
def calc1(): Future[Int] = Future { ... }
def calc2(): Future[Int] = Future { ... }
def calc3(x: Int, y: Int) = Future[Int] = Future { ... }

val finalResult = for {
    preResult <- calc0()
    // Here ``calc1`` and ``calc2`` gets executed asynchronously
    value1 = calc1()
    value2 = calc2()
    firstResult <- value1
    secondResult <- value2
    thirdResult <- calc3(firstResult, secondResult)
} yield thirdResult
```

The code block above also computes `value1` and `value2` asynchronously, similar to our first example.

10.2 Combining `Future` and `Perhaps`

In Sentinel, `Future` is often combined with the `Perhaps` type we have defined earlier. Conceptually, this means that there are cases where Sentinel invokes a function asynchronously that may or may not return its expected type. A function with the following signature, for example:

```
def storeUpload(contents: Array[Byte]): Future[Perhaps[DatabaseId]] = { ... }
```

is expected to be executed asynchronously. In this case, it is a function to store user uploads which will return the database ID of the stored file. There could be different reasons of wrapping the database ID inside `Perhaps`. One is that we may want to tell users when they are uploading files they have previously uploaded, so we can save disk space and the user do not store the same data twice.

Naturally there are still cases where we do not need our results being wrapped inside `Future`, or `Perhaps`, or even both.

Consider our earlier `extractJson` function. This is a function that we expect to execute very early upon user upload. Does it make sense to wrap it inside a `Future`? It depends on how you setup your processing of course. But it is easy to imagine that we first want to ensure that the data that the user uploads can indeed be processed into JSON

first before doing anything else. In this case, we would only need to wrap the return value inside a `Perhaps` and not a `Future` since no other processing would be done in parallel at the time we are doing validation.

On the other hand, methods that interact with the database directly are often wrapped only inside a `Future` and not `Perhaps`. An example would be a function storing sample data parsed from the JSON record:

```
def storeSamples(samples: Seq[Samples]): Future[Unit] = { ... }
```

This is the case because in most cases we do not expect database connection failures to be something the user can recover from, so there is little point in letting them know this. We should anticipate indeed that the database connection from time to time may fail, but this is something that should only be displayed in the server logs and not to the user, so we do not use `Perhaps` here.

Tip: For asynchronous methods where the error is not something the user can work on, we should let the `Future` fail. There is a built-in check in Sentinel that captures these `Future` failures and then converts it to a general HTTP 500 Internal Server Error to the user.

The fact that not all methods return `Future`, or `Perhaps`, or even `Future[Perhaps]` is something we need to take into account when composing these functions. We saw earlier that we can use for comprehensions for a series of calls that all return `Perhaps`, or a series of calls that all return `Future` in some form.

This is not the case when composing functions of these different abstract types, however. Let's say we have these functions that we wish to compose into a single processing step:

```
def extractMetrics(contents: Array[Byte]): Perhaps[Metrics] = { ... }
def storeMetrics(metrics: Metrics): Future[DatabaseId] = { ... }
def storeUpload(contents: Byte): Future[Perhaps[DatabaseId] = { ... }
```

Then this will not work because they all return different types:

```
def processUpload(contents: Array[Byte]) = {
    metrics <- extractMetrics(contents)
    metricsId <- storeMetrics(metrics)
    fileId <- storeUpload(metrics)
} yield (metricsId, fileId)
```

Important: Not only will the code above not compile, but we are also launching the all `Future` sequentially.

A solution is to make these functions return the same type. It does not necessarily mean we have to change the functions themselves. After all, we have seen that we can not force all functions to use `Future` or `Perhaps`. Not only is this conceptually wrong, it is also impractical to expect all functions we write to use a similar abstract type. What we want is to somehow 'lift' the return values of the functions into a common return type, but only when we want to compose them. This way, functions can remain as they are yet can still be composed with others when needed.

10.2.1 Lifting Into `Future[Perhaps]`

What should we use then as the common type? A good candidate is actually a function with a `Future[Perhaps[T]]` type. This type can be interpreted as types whose value are computed asynchronously with a possibility of returning an `ApiPayload` to be displayed to the user. Recall that in our case, `Perhaps[T]` is an alias for the disjunction type `ApiPayload \/ T`. `Future[Perhaps[T]]` is actually then an alias for `Future[ApiPayload \/ T]`.

Note: Why not `Perhaps[Future[T]]` instead? Since this type is an alias for `ApiPayload \/ Future[T]`, we can interpret it as types whose value when failing is `ApiPayload` and when successful is an asynchronous

computation returning `T`. In other words, it encodes the types whose error value is not computed asynchronously. This distinction does not really make sense in practice. It sort of means that we only do the asynchronous computation when we know we will not get any failures, but we could not have known this prior to the computation itself.

How do we lift into `Future[Perhaps]`? There are two cases we need to consider, lifting from `Perhaps` types and lifting from `Future` types. From `Perhaps` types, we can simply wrap it using `Future.successful`. The function, provided by the standard library, is precisely for lifting types into a `Future` without launching any parallel computation thread which `Future` normally does. In other words, it creates a completed `Future` without the unecessary work of launching the `Future`.

`Future.successful` can be used like so:

```
// A simple call to our ``extractMetrics`` function
val metrics: Perhaps[Metrics] = extractMetrics(...)

// Lifting the call into a ``Future``
val futureMetrics: Future[Perhaps[Metrics]] =
    Future.successful(extractMetrics(...))
```

For the second case of lifting `Future` into `Future[Perhaps]`, we can simply use `Future.map` to lift the results inside it. Essentially, we then only lift the inner type of `Future` into `Perhaps`:

```
// A call to launch our ``storeMetrics`` function
val dbId: Future[DatabaseId] = storeMetrics(...)

// Lifting the ``DatabaseId`` to ``Perhaps[DatabaseId]``
val futureDbId: Future[Perhaps[DatabaseId]] =
    dbId.map(id => \/-(id))
```

Now, having covered both cases, let's make our earlier for comprehension work. We will also define `UploadIds`, a helper case class for storing our uploaded IDs.

```
// Helper case class for storing uploaded IDs
case class UploadId(metrics: DatabaseId, file: DatabaseId)

def processUpload(contents: Array[Byte]): Future[Perhaps[UploadId]] = {
    // Here we lift from ``Perhaps``
    metrics <- Future.successful(extractMetrics(contents))
    // Here we lift from ``Future``. Remember
    // that we want to store asynchronously,
    // so we need to launch the computation as
    // value declarations
    asyncMetricsId = storeMetrics(metrics).map(id => \/-(id))
    asyncFileId = storeUpload(metrics)
    // This is where we actually wait for the
    // store methods in parallel
    metricsId <- asyncMetricsId
    res = UploadId(metricsId, fileId)
} yield res

val uploadResult: Future[Perhaps[UploadId]] = processUpload(...)
```

Does the code above work? Not quite. It has to do with the fact that we are now using two layers of abstract types, `Future` and `Perhaps`. This means that in this line:

```
...
    // Here we lift from ``Perhaps``
    metrics <- Future.successful(extractMetrics(contents))
```

`metrics` does not evaluate to a `Metrics` object, but instead to a `Perhaps[Metrics]` object. The for comprehension unwraps only the `Future` and not `Perhaps`. Consequently, we can not use `metrics` directly as an argument for `storeMetrics` afterwards. We have to unwrap it first, for example:

```
...
    // Here we lift from ``Perhaps``
    metrics <- Future.successful(extractMetrics(contents))
    asyncMetricsId = metrics match {
        // when metrics is an error type, we still
        // need to wrap it inside a ``Future``
        case -\/(err) => Future.successful(err)
        case \/-(ok)  => storeMetrics(ok).map(id => \/-(id))
    }
```

Not only is this too verbose, but it also reads quite unintuitively. Consider also that this is only for one part of the statement. We have to unwrap subsequent statements to make sure `Perhaps` is also unwrapped. Surely we can do better than this? Indeed we can. The answer lies in another type defined in scalaz, called the `EitherT` type.

10.2.2 Scalaz's `EitherT`

`EitherT` is a type meant to be used when the disjunction type `\/` is wrapped inside some other abstract types (in our case, a `Future`). Not all abstract types can be used here, and `Future` itself needs a little bit of enhancement.

Tip: The mathematical term for `EitherT` is a monad transformer, since `\/` is a monad, `Future` can be made into a monad, and `EitherT` transforms them both into another monad that is a combination of both. We are intentionally not using these terms here, but they are actually common abstractions that pop up here and there in various codebases. Plenty of tutorials and guides about monads can be found online. If you are interested in monad transformers in Scala in particular, we found the guide here and the guide here as good starting points.

What can we do with `EitherT`? Essentially it boils down to this: `EitherT` allows us to unwrap both `Future` and `Perhaps` in a single for comprehension statement by wrapping them into another type that combines both `Future` and `Perhaps`. The new type, in our case, is called `EitherT[Future, ApiPayload, T]`. It is not `Future[Perhaps]]`, but it needs to be made from `Future[Perhaps]`.

```
// Alias for the new type, let's call it AsyncPerhaps
val AsyncPerhaps[+T] = EitherT[Future, ApiPayload, T]

// From a ``Future[Perhaps]``
val val1: Future[Perhaps[Int]] = ...
val lifted1: AsyncPerhaps[Int] = EitherT(v)

// From a ``Future``
val val2: Future[Int] =
val lifted2: AsyncPerhaps[Int] = EitherT.right(val2)

// From a ``Perhaps``
val val3: Perhaps[Int] = ...
val lifted3: AsyncPerhaps[T] = EitherT(Future.successful(val3))

// Even from an arbitrary type
val val4: Int = ...
val lifted4: AsyncPerhaps[T] = val.point[AsyncPerhaps]
```

Notice that `EitherT` can be used directly on values with a `Future[Perhaps]]` type in our case. For `Perhaps` values, we need to wrap it inside a `Future` still. For `Future` methods, we use a helper method in `EitherT` that

essentially maps the inner type with `Perhaps` (essentially what we did earlier when we did `Future.map` manually). In short, we still needed to lift our types into `Future[Perhaps]` first.

Using `EitherT`, our previous iteration then becomes this:

```
...

def processUpload(contents: Array[Byte]) = {
    metrics <- EitherT(Future.successful(extractMetrics(contents)))
    asyncMetricsId = storeMetrics(metrics)
    asyncFileId = storeUpload(metrics)
    metricsId <- EitherT.right(asyncMetricsId)
    res = UploadId(metricsId, fileId)
} yield res

val wrappedResult: EitherT[Future, ApiPayload, UploadResult] =
    processUpload(...)
```

There is only one thing left to do, which is to unwrap back `wrappedResult`. Our `EitherT` type can be considered a helper type that allows us to compose all our functions. The type that is actually useful for us outside of the for comprehension, though, is the `Future[Perhaps]` type, since our for comprehensions combines both `Future` and `Perhaps` already. We can convert `EitherT[Future, ApiPayload, UploadResult` back to `Future[Perhaps[UploadResult]` by invoking the `.run` method:

```
...

def processUpload(contents: Array[Byte]) = {
    metrics <- EitherT(Future.successful(extractMetrics(contents)))
    asyncMetricsId = storeMetrics(metrics)
    asyncFileId = storeUpload(metrics)
    metricsId <- EitherT.right(asyncMetricsId)
    res = UploadId(metricsId, fileId)
} yield res

val finalResult: Future[Perhaps[UploadResult]] =
    processUpload(...).run
```

And that's it. We now have combined `Future` and `Perhaps` into a single block of computation. `EitherT` has definitely improved readablity since it spares us from the need to unwrap manually. We are not completely done yet, however. There are some implicit values required by future (its `ExecutionContext`) and some implicit methods required by scalaz to make this work. The details can be ignored for our discussion. What's important to know is that these are all already defined in the `FutureMixin` trait in the `nl.lumc.sasc.sentinel.utils` package.

10.3 Sentinel's `FutureMixin`

There are two things that this trait does that helps you combine `Future` and `Perhaps`:

1. It defines all the necessary implicit methods to make `Future` suitable for `EitherT`.
2. It defines an object called `?` that you can use to make your for comprehension even more compact.

We will discuss the implicit methods here, but we would like to note the `?` object. Recall that our last iteration of the `processUpload` function is like this:

```
...

def processUpload(contents: Array[Byte]) = {
    metrics <- EitherT(Future.successful(extractMetrics(contents)))
    asyncMetricsId = storeMetrics(metrics)
```

```
    asyncFileId = storeUpload(metrics)
    metricsId <- EitherT.right(asyncMetricsId)
    res = UploadId(metricsId, fileId)
} yield res
```

The `?` object defines all `EitherT` calls necessary for lifting our type into a function called `<~`. The names are not exactly pronounceable, but for good reason. Since You can omit the dot (`.`) and parentheses (`(` and `)`) when calling an object's function, you can then do this with the `?` object:

```
...

// This must be done in an object extending the
// `FutureMixin` trait now.
def processUpload(contents: Array[Byte]) = {
    metrics <- ? <~ extractMetrics(contents)
    asyncMetricsId = storeMetrics(metrics)
    asyncFileId = storeUpload(metrics)
    metricsId <- ? <~ asyncMetricsId
    res = UploadId(metricsId, fileId)
} yield res
```

Notice there that we don't need to call `EitherT` manually again. For the first statement, for example, we are doing:

```
...

    metrics <- ? <~ extractMetrics(contents)
```

Which is essentially the same as:

```
...

    metrics <- ?.<~(extractMetrics(contents))
```

Some would still prefer to user `EitherT` here, and that is fine. The `?` object is simply there to give you the option to shorten your `EitherT` instantiations by leveraging Scala's features.

10.4 Next Steps

We have covered a lot now! Now you should be ready to implement all the things we have learned in a real Sentinel `Processor`. Head over to the next section to do just that.

CHAPTER 11

Creating the Processors

In this section we will define the processor objects required for processing summary files from our Maple pipeline. Recall that processors are the actual objects where pipeline support (that means summary file upload and statistics querying) is implemented. There are two processors we need to define: a runs processor to process data uploaded by a user and a stats processor to return statistics query.

11.1 The Runs Processor

First up is the runs processor. We mentioned earlier that it is meant for processing user uploads. This means extracting all the values we need into various record objects we have defined earlier. Additionally, Sentinel also requires that the raw contents of the file be saved into the database. The purpose of this is twofolds: to ensure duplicate files are not uploaded and to allow users to download their summary files again. Sentinel also requires you to save the extracted record objects into the database. This is the actual metrics that will be returned when a user queries for the metrics later on.

We will start with the general outline first and then define the functions required for processing the actual uploaded run summary file. All of this will be in a file called `MapleRunsProcessor.scala`

```
package nl.lumc.sasc.sentinel.exts.maple

import scala.concurrent._

import org.bson.types.ObjectId
import org.json4s.JValue

import nl.lumc.sasc.sentinel.adapters._
import nl.lumc.sasc.sentinel.models.User
import nl.lumc.sasc.sentinel.processors.RunsProcessor
import nl.lumc.sasc.sentinel.utils.{ ValidatedJsonExtractor, MongodbAccessObject }

/**
 * Example of a simple pipeline runs processor.
 *
 * @param mongo MongoDB access object.
 */
class MapleRunsProcessor(mongo: MongodbAccessObject)
        extends RunsProcessor(mongo)
        with ReadGroupsAdapter
        with ValidatedJsonExtractor {
```

```
// Our code will be here
}
```

Our runs processor extends the `RunsProcessor` abstract class that is instantiated by an object that provides access to the database. It is further enriched by the `ReadGroupsAdapter` trait since we want to process data both on the sample and read group level, and the `ValidatedJsonExtractor` since we want to parse and validate our incoming JSON. It is important to note here that the `RunsProcessor` abstract class extends `FutureMixin` so we will need to define an implicit `ExecutionContext` as well.

Let's now define some of the required abstract values and methods. We can start with two simple ones: the pipeline name and the pipeline schema. The pipeline name is how the pipeline would be used in URL parameters (that means it's safest if we only use alphanumeric characters) and the pipeline schema is the resource path to the schema we defined earlier.

```
...

class MapleRunsProcessor(mongo: MongodbAccessObject)
        extends RunsProcessor(mongo)
        with ReadGroupsAdapter
        with ValidatedJsonExtractor {

    /** Exposed pipeline name. */
    def pipelineName = "maple"

    /** JSON schema for incoming summaries. */
    def jsonSchemaUrls = Seq("/schema_examples/maple.json")
}
```

Next is to define the `Future` execution contexts and the record objects. Aliasing the record objects in the runs processor here is a requirement of both the `RunsProcessor` abstract class and the `ReadGroupsAdapter` trait. It allows the generic methods in `RunsProcessor` to work with our custom record types.

```
...

class MapleRunsProcessor(mongo: MongodbAccessObject)
        extends RunsProcessor(mongo)
        with ReadGroupsAdapter
        with ValidatedJsonExtractor {

    ...

    /** Implicit execution context. */
    implicit private def context: ExecutionContext =
        ExecutionContext.global

    /** Run records container. */
    type RunRecord = MapleRunRecord

    /** Sample-level metrics container. */
    type SampleRecord = MapleSampleRecord

    /** Read group-level metrics container. */
    type ReadGroupRecord = MapleReadGroupRecord
}
```

Now we want to define how the record objects can be created from a parsed JSON. The exact implementation of this part is completely up to you. You can do this with one function, two functions, or more. You can call the function anything you want (so long as it does not interfere with the any parent trait methods). Basically, the details will differ depending on the JSON file's structure.

In our case, one way to do this is using the `extractUnits` and a helper case class `MapleUnits` which will contain `MapleSampleRecord` and `MapleReadGroupRecord` objects. The implementation looks like this:

```
...

class MapleRunsProcessor(mongo: MongodbAccessObject)
        extends RunsProcessor(mongo)
        with ReadGroupsAdapter
        with ValidatedJsonExtractor {

    ...

    /** Helper case class for storing records. */
    case class MapleUnits(
        samples: Seq[MapleSampleRecord],
        readGroups: Seq[MapleReadGroupRecord])

    /**
     * Extracts the raw summary JSON into samples and read groups containers.
     *
     * @param runJson Raw run summary JSON.
     * @param uploaderId Username of the uploader.
     * @param runId Database ID for the run record.
     * @return Two sequences: one for sample data and the other for read group data.
     */
    def extractUnits(runJson: JValue, uploaderId: String,
                     runId: ObjectId): = {

      /** Name of the current run. */
      val runName = (runJson \ "run_name").extractOpt[String]

      /** Given the sample name, read group name, and JSON section of the read group, create a read g
      def makeReadGroup(sampleName: String, readGroupName: String, readGroupJson: JValue) =
        MapleReadGroupRecord(
          stats = MapleReadGroupStats(
            nReadsInput = (readGroupJson \ "nReadsInput").extract[Long],
            nReadsAligned = (readGroupJson \ "nReadsAligned").extract[Long]),
          uploaderId = uploaderId,
          runId = runId,
          readGroupName = Option(readGroupName),
          sampleName = Option(sampleName),
          runName = runName)

      /** Given the sample name and JSON section of the sample, create a sample container. */
      def makeSample(sampleName: String, sampleJson: JValue) =
        MapleSampleRecord(
          stats = MapleSampleStats(nSnps = (sampleJson \ "nSnps").extract[Long]),
          uploaderId, runId, Option(sampleName), runName)

      /** Raw sample and read group containers. */
      val parsed = (runJson \ "samples").extract[Map[String, JValue]].view
        .map {
          case (sampleName, sampleJson) =>
            val sample = makeSample(sampleName, sampleJson)
            val readGroups = (sampleJson \ "readGroups").extract[Map[String, JValue]]
              .map { case (readGroupName, readGroupJson) => makeReadGroup(sampleName, readGroupName,
              .toSeq
            (sample, readGroups)
        }.toSeq
```

```
      MapleUnits(parsed.map(_._1), parsed.flatMap(_._2))
    }
}
```

To be fair, that is still quite verbose and there are possibly other ways of doing it. As we mentioned, though, this depends largely on how your JSON file looks like. It is often the case as well that this is the most complex part of the code that you need to define.

The final part that we need to define is the actual function for processing the upload. This is where we combine all functions we have defined earlier (and some that are already defined by the traits we are extending) in one place. It covers the part after user upload up until the part where we create a `RunRecord` object to be sent back to the user as a JSON payload, notifying that the upload has been successful.

The function is called `processRunUpload` and is a requirement of the `RunsProcessor` abstract class. It has the following signature:

```
/**
 * Processes and stores the given uploaded file to the run records collection.
 *
 * @param contents Upload contents as a byte array.
 * @param uploadName File name of the upload.
 * @param uploader Uploader of the run summary file.
 * @return A run record of the uploaded run summary file.
 */
def processRunUpload(
    contents: Array[Byte],
    uploadName: String,
    uploader: User): Future[Perhaps[RunRecord]]
```

It is invoked by Sentinel's `RunsController` after user authentication (hence the `uploader` argument). You do not need to worry about `uploadName` nor `uploader` at this point. The important thing is to note the return type: `Future[Perhaps[RunRecord]]`. We have covered this in our earlier guides. This is where we now actually implement a working code for processing the user upload.

There are of course several different ways to implement `processRunUpload`. Here is one that we have, to give you an idea:

```
class MapleRunsProcessor(mongo: MongodbAccessObject)
        extends RunsProcessor(mongo)
        with ReadGroupsAdapter
        with ValidatedJsonExtractor {

    ...

    def processRunUpload(contents: Array[Byte], uploadName: String, uploader: User) = {
        val stack = for {
            // Make sure it is JSON
            runJson <- ? <~ extractAndValidateJson(contents)
            // Store the raw file in our database
            fileId <- ? <~ storeFile(contents, uploader, uploadName)
            // Extract samples and read groups
            units <- ? <~ extractUnits(runJson, uploader.id, fileId)
            // Invoke store methods asynchronously
            storeSamplesResult = storeSamples(units.samples)
            storeReadGroupsResult = storeReadGroups(units.readGroups)
            // Check that all store methods are successful
            _ <- ? <~ storeReadGroupsResult
            _ <- ? <~ storeSamplesResult
            // Create run record
```

```
            sampleIds = units.samples.map(_.dbId)
            readGroupIds = units.readGroups.map(_.dbId)
            run = MapleRunRecord(fileId, uploader.id, pipelineName, sampleIds, readGroupIds)
            // Store run record into database
            _ <- ? <~ storeRun(run)
        } yield run

        stack.run
    }
}
```

Our implementation above consists of a series of functions; beginning with parsing and validating the JSON, storing the raw uploaded bytes, extracting the record objects, and then storing the record objects. Everything is wrapped inside `EitherT[Future, ApiPayload, RunRecord]`, and stored as a value called `stack`. This of course means we still need to invoke the `.run` method in order to get the `Future[Perhaps[RunRecord]]` object which we will return.

We hope by now it is also clear to you that this single for comprehension block already has error handling with the `ApiPayload` type built in and that we always write to the database asynchronously whenever possible.

Here is our finished, complete `MapleRunsProcessor` for your reference:

```
import scala.concurrent._

import org.bson.types.ObjectId
import org.json4s.JValue

import nl.lumc.sasc.sentinel.adapters._
import nl.lumc.sasc.sentinel.models.User
import nl.lumc.sasc.sentinel.processors.RunsProcessor
import nl.lumc.sasc.sentinel.utils.{ ValidatedJsonExtractor, MongodbAccessObject }

/**
 * Example of a simple pipeline runs processor.
 *
 * @param mongo MongoDB access object.
 */
class MapleRunsProcessor(mongo: MongodbAccessObject)
    extends RunsProcessor(mongo)
    with ReadGroupsAdapter
    with ValidatedJsonExtractor {

  /** Exposed pipeline name. */
  def pipelineName = "maple"

  /** JSON schema for incoming summaries. */
  def jsonSchemaUrls = Seq("/schema_examples/maple.json")

  /** Run records container. */
  type RunRecord = MapleRunRecord

  /** Sample-level metrics container. */
  type SampleRecord = MapleSampleRecord

  /** Read group-level metrics container. */
  type ReadGroupRecord = MapleReadGroupRecord

  /** Execution context. */
  implicit private def context: ExecutionContext = ExecutionContext.global
```

```
/** Helper case class for storing records. */
case class MapleUnits(
  samples: Seq[MapleSampleRecord],
  readGroups: Seq[MapleReadGroupRecord])

/**
 * Extracts the raw summary JSON into samples and read groups containers.
 *
 * @param runJson Raw run summary JSON.
 * @param uploaderId Username of the uploader.
 * @param runId Database ID for the run record.
 * @return Two sequences: one for sample data and the other for read group data.
 */
def extractUnits(runJson: JValue, uploaderId: String,
                 runId: ObjectId): MapleUnits = {

  /** Name of the current run. */
  val runName = (runJson \ "run_name").extractOpt[String]

  /** Given the sample name, read group name, and JSON section of the read group, create a read gr
  def makeReadGroup(sampleName: String, readGroupName: String, readGroupJson: JValue) =
    MapleReadGroupRecord(
      stats = MapleReadGroupStats(
        nReadsInput = (readGroupJson \ "nReadsInput").extract[Long],
        nReadsAligned = (readGroupJson \ "nReadsAligned").extract[Long]),
      uploaderId = uploaderId,
      runId = runId,
      readGroupName = Option(readGroupName),
      sampleName = Option(sampleName),
      runName = runName)

  /** Given the sample name and JSON section of the sample, create a sample container. */
  def makeSample(sampleName: String, sampleJson: JValue) =
    MapleSampleRecord(
      stats = MapleSampleStats(nSnps = (sampleJson \ "nSnps").extract[Long]),
      uploaderId, runId, Option(sampleName), runName)

  /** Raw sample and read group containers. */
  val parsed = (runJson \ "samples").extract[Map[String, JValue]].view
    .map {
      case (sampleName, sampleJson) =>
        val sample = makeSample(sampleName, sampleJson)
        val readGroups = (sampleJson \ "readGroups").extract[Map[String, JValue]]
          .map { case (readGroupName, readGroupJson) => makeReadGroup(sampleName, readGroupName, re
          .toSeq
        (sample, readGroups)
    }.toSeq

  MapleUnits(parsed.map(_._1), parsed.flatMap(_._2))
}

/**
 * Validates and stores uploaded run summaries.
 *
 * @param contents Upload contents as a byte array.
 * @param uploadName File name of the upload.
 * @param uploader Uploader of the run summary file.
 * @return A run record of the uploaded run summary file or a list of error messages.
```

```
   */
  def processRunUpload(contents: Array[Byte], uploadName: String, uploader: User) = {
    val stack = for {
      // Make sure it is JSON
      runJson <- ? <~ extractAndValidateJson(contents)
      // Store the raw file in our database
      fileId <- ? <~ storeFile(contents, uploader, uploadName)
      // Extract samples and read groups
      units <- ? <~ extractUnits(runJson, uploader.id, fileId)
      // Invoke store methods asynchronously
      storeSamplesResult = storeSamples(units.samples)
      storeReadGroupsResult = storeReadGroups(units.readGroups)
      // Check that all store methods are successful
      _ <- ? <~ storeReadGroupsResult
      _ <- ? <~ storeSamplesResult
      // Create run record
      sampleIds = units.samples.map(_.dbId)
      readGroupIds = units.readGroups.map(_.dbId)
      run = MapleRunRecord(fileId, uploader.id, pipelineName, sampleIds, readGroupIds)
      // Store run record into database
      _ <- ? <~ storeRun(run)
    } yield run

    stack.run
  }
}
```

And that's it! You now have fully-functioning runs processor.

11.2 The Stats Processor

The final step is defining the stats processor. This step will be relatively simpler than the inputs processor, since Sentinel now has a better idea of what to expect from the database records (courtesy of the record objects we defined earlier).

```
package nl.lumc.sasc.sentinel.exts.maple

class MapleStatsProcessor(mongo: MongodbAccessObject)
    extends StatsProcessor(mongo) {

  def pipelineName = "maple"

  /* Function for retrieving Maple sample data points. */
  def getMapleSampleStats =
    getStats[MapleSampleStats]("stats")(AccLevel.Sample) _

  /* Function for aggregating over Maple read group data points. */
  def getMapleSampleAggrStats =
    getAggregateStats[MapleSampleStatsAggr]("stats")(AccLevel.Sample) _

  /** Function for retrieving Maple read group data points. */
  def getMapleReadGroupStats =
    getStats[MapleReadGroupStats]("stats")(AccLevel.ReadGroup) _

  /* Function for aggregating over Maple read group data points. */
  def getMapleReadGroupAggrStats =
```

```
    getAggregateStats[MapleReadGroupStatsAggr]("stats")(AccLevel.ReadGroup) _
}
```

And that is all we need to have a fully functioning `MapleStatsProcessor`. See also that what we are doing here is defining functions partially (notice the use of _ at the end of each function, which is how we do partial function invocations in Scala).

In addition to `pipelineName`, which has the same meaning and use as the `pipelineName` in `MapleRunsProcessor`, there are four functions we define here. These four functions calls two functions that are defined in the `StatsProcessor` abstract class: `getStats` and `getAggregateStats`.

Let's take a look at `getStats` invoked in `getMapleSampleStats` first. Here we are calling it with one type parameter, the `MapleSampleStats` type. This is a type we have defined earlier to contain our sample-level metrics. In essence this is what `getStats` does. Upon user query, it creates sample-level metrics container objects from our previously stored database records. It does that by reading the `MapleSampleRecord` object and extracting an attribute from that object whose type is `MapleSampleStats`. `getStats` is not smart enough to know which attribute that is, however, so we need to supply the attribute name as an argument as well. In our case, this attribute is called `stats` and indeed we use `stats` here as the first value argument to `getStats`. The final argument `AccLevel.Sample` simply tells `getStats` that we want this query to operate on the sample-level instead of read group-level.

`getStats` in `getMapleReadGroupStats` is called with the same logic. The difference is only the final argument, where we use `AccLevel.ReadGroup` as we want this function to operate on the read group level.

`getAggregateStats` is not exactly similar, but in this case are also called with the same logic. The main difference is that the returned object is a single object containing various aggregated values.

With this, we have completely defined the required processors and internal data models. The next step is to expose these processors via the HTTP controllers.

CHAPTER 12

Updating Controllers

Note: We are still working on making the controllers setup more modular. At the moment, adding support to new pipelines means changing the source code of the existing controllers directly.

For our *Maple* pipeline, there are two controllers that need to be made aware of the processors we wrote in the previous section. The controllers, as mentioned in an earlier section, are responsible for mapping a given URL to a specific action that Sentinel will execute. Two controllers that are tightly coupled with the processors we have written are called `RunsController`, which handles a URL for summary file uploads, and `StatsController`, which handles URLs for statistics query. Accordingly, the `RunsController` needs to be made aware of our `MapleRunsProcessor` and the `StatsController` of our `MapleStatsProcessor`.

What do we mean by making the controllers aware of our processors? Basically, it means they need to know how to initialize the processor classes. They are handled quite differently for each controllers. This is because Sentinel by default has one endpoint for uploading the summary files (by default this is a `POST` request on the `/runs` endpoint), but more than one endpoint for querying the statistics. For the current version, we can simply edit the main `ScalatraBootstrap` class to add our `MapleRunsProcessor` class while we need to manually edit the `StatsController` class to add our `MapleStatsProcessor`.

The difference in instantiating the processor classes is mostly caused by the need to document the schema returned by different statistics queries. A given pipeline may return a metrics object (the metrics case class we defined earlier) which is completely different from another pipeline. This is not the case for the summary file uploads, where all pipeline processors use the same `processRunUpload` method. This limitation may be removed in future versions.

12.1 RunsController

The `RunsController` can be made aware of our `MapleRunsProcessor` by injecting the class name in the class responsible for instantiating the controller itself. This class, the `ScalatraBootstrap` class, has an `init` method where it instantiates all controllers. What we need to do, is to have an implicit value of the type `Set[MongodbAccessObject => RunsProcessor]`. In other words, a set of functions that creates a `RunsProcessor` object given a `MongodbAccessObject` object.

The `MongodbAccessObject` object is an object representing access to our database. In production runs, this represents access to a live database server, while in testing this is replaced by a testing server. Sentinel has a helper method called `makeDelayedProcessor` in the `nl.lumc.sasc.sentinel.utils.reflect` package, which can create the function we require from the processor class.

Invoking it is then quite simple:

```
class ScalatraBootstrap extends LifeCycle {

    ...

    override def init(context: ServletContext) {

        val runsProcessors = Set(
            makeDelayedProcessor[MapleRunsProcessor])

        // continue to initialize and mount the controllers
    }
}
```

And that's it. That's all we require so that the `RunsController` class can process uploaded Maple summary files.

12.2 StatsController

The last step is to update the stats controller. There are four endpoints that we can define, each using a method that we have written in `MapleStatsProcessor`:

- For the sample-level data points endpoint, `/stats/maple/samples`
- For the sample-level aggregated endpoint, `/stats/maple/samples/aggregate`
- For the read group-level data points endpoint, `/stats/maple/readgroups`
- For the read group-level aggregated endpoint, `/stats/maple/readgroups/aggregate`

We will define the sample-level endpoints together and leave the read group-level endpoints for you to define.

Note: The first part of the endpoint, */stats* is already automatically set by Sentinel, so our route matchers only needs to define the last part.

Before we begin, we need to import the Maple stats containers and their processor in the `StatsController.scala` file:

```
import nl.lumc.sasc.sentinel.lumc.ext.maple._
```

Different from its runs processor counterpart, we will need to instantiate the `MapleStatsProcessor` directly inside the `StatsController` body:

```
val maple = new MapleStatsProcessor(mongo)
```

After this, we can start with implementing the actual endpoints.

12.2.1 /stats/maple/samples

In the `StatsController.scala` file, add the following Swagger operation definition:

```
val statsMapleSamplesGetOp =
    (apiOperation[Seq[MapleSampleStats]]("statsMapleSamplesGet")
        summary "Retrieves Maple sample-level data points"
        parameters (
          queryParam[Seq[String]]("runIds")
            .description("Run ID filter.")
            .multiValued
```

```
            .optional,
          queryParam[String]("userId").description("User ID.")
            .optional,
          headerParam[String](HeaderApiKey).description("User API key.")
            .optional)
        responseMessages (
          StringResponseMessage(400, "Invalid Run IDs supplied"),
          StringResponseMessage(401, Payloads.OptionalAuthenticationError.message)))
```

While the definitions is not required per-se, it is always useful to let users know the parameters your endpoint accepts. In this case, our endpoint accepts three optional parameters: run ID for filtering and user ID with the associated API key for optional authentication. We also define the HTTP error code we will return in case any of the supplied arguments are invalid.

Here comes the route matcher for the data points query:

```
get("/maple/datapoints", operation(statsMapleSamplesGetOp)) {

  val runIds = params.getAs[Seq[DbId]]("runIds").getOrElse(Seq.empty)
  val idSelector = ManyContainOne("runId", runIds)

  val user = Try(simpleKeyAuth(params => params.get("userId"))).toOption
  if ((Option(request.getHeader(HeaderApiKey)).nonEmpty || params.get("userId").nonEmpty) && user.isE
    halt(401, Payloads.OptionalAuthenticationError)

  new AsyncResult {
    val is = maple.getMapleSampleStats(idSelection, user)
      .map {
        case -\/(err) => err.toActionResult
        case \/-(res) => Ok(res)
      }
  }
}
```

In the code block, you can see that the first two `val` declarations capture the parameters supplied by the user. The `runIds` parameter is an optional parameter for selecting only particular run IDs. These are IDs that users get when they upload the run summary file for the first time and are assigned randomly by the database. We then proceeded to create a selector object (basically a `MongoDBObject`) which will then be used for filtering the metrics. Here we use the Sentinel-defined `ManyContainOne` helper case class, which has the effect of selecting any metrics whose run ID is contained within the user-supplied run ID. If the user does not supply any run IDs, then no filtering will be done.

The `val user` declaration allows for optional user authentication. A succesfully authenticated user will get additional information for data points that he/she has uploaded, such as the sample name. He/she may still see data points uploaded by other users, only without any identifying information.

Finally, we run the query on the database using the `AsyncResult` class provided by Scalatra. This allows our query to be run asynchronously so that Sentinel may process other queries without waiting for this to finish.

12.2.2 /stats/maple/samples/aggregate

With that set, we can now define the endpoint for aggregated queries. Let's start with the API definition as before:

```
val statsMapleSamplesAggregateGetOp =
    (apiOperation[MapleSampleStatsAggr]("statsMapleSamplesAggregateGet")
        summary "Retrieves Maple sample-level aggregated data points"
        parameters
        queryParam[Seq[String]]("runIds")
            .description("Run ID filter.")
```

```
            .multiValued
            .optional
        responseMessages StringResponseMessage(400, "Invalid Run IDs supplied"))
```

The API definition is similar to the single data points, with difference being the authentication is not present anymore. This makes sense, since aggregated data points do not have any name labels associated with them.

```
get("/maple/datapoints/aggregate", operation(statsMapleDatapointsAggregateGetOp)) {
  val runIds = getRunObjectIds(params.getAs[String]("runIds"))
  val idSelector = ManyContainOne("runId", runIds)

  new AsyncResult {
    val is =
        maple.getMapleSampleAggrStats(None)(idSelector)
          .map {
            case -\/(err) => err.toActionResult
            case \/-(res) => Ok(res)
          }
  }
}
```

This is almost the same as our previous endpoint, except that there is an extra `None` argument supplied to the function above. This is used only when our stats processor distinguishes between single-end and paired-end data. In our case, we made no such distinction and thus we can simply use `None` there.

12.3 Epilogue

The `MapleStatsController` implementation marks the end of our tutorial. You have just added a new pipeline support to Sentinel! Feel free to play around with uploading and querying the endpoints you just created. When you're more familiar with the code base, you can experiment with adding support for more complex pipelines. If that's not enough, head over to the Contributing page and see how you can contribute to Sentinel development.

CHAPTER 13

Contributing

Any type of contributions is very welcomed and appreciated :)! From bug reports to new features, there is always room to help out.

13.1 Quick Links

- Issue tracker: https://github.com/lumc/sentinel/issues
- Source code: https://github.com/lumc/sentinel/issues
- Git: https://github.com/LUMC/sentinel.git

13.2 Bug Reports & Feature Suggestions

Feel free to report bugs and/or suggest new features about our local LUMC deployment or Sentinel in general to our issue tracker. We do request that you be as descriptive as possible. Particularly for bugs, please describe in as much detail as possible what you expected to see and what you saw instead.

13.3 Documentation

Documentation updates and/or fixes are very appreciated! We welcome everything from one-letter typo fixes to new documentation sections, be it in the internal ScalaDoc or our user guide (the one you're reading now). You are free to submit a pull request for documentation fixes. If you don't feel like cloning the entire code, we are also happy if you open an issue on our issue tracker.

13.4 Bug Fixes

Bug fix contributions requires that you have a local development environment up and running. Head over to the Local Development Setup section for a short guide on how to do so.

To find bugs to fix, you can start by browsing our issue tracker for issues labeled with `bug`. You can also search through the source code for `FIXME` notes. Having found an issue you would like to fix, the next steps would be:

1. Create a new local branch, based on the last version of *master*.
2. Implement the fix.

3. Make sure all tests pass. If the bug has not been covered by any of our tests, we request that new tests be added to protect against regressions in the future.
4. Commit your changes.
5. Submit a pull request.

We will then review your changes. If it is all good, it will be rebased to `master` and we will list your name in our contributors list :).

And yes, we did say rebase up there, not merge. We prefer to keep our git history linear, which means changes will be integrated to `master` via `git rebase` and not `git merge`.

13.5 New Features

Feature implementations follow almost the same procedure as *Bug Fixes*. The difference being that you are not limited to the feature requests we list on the issue tracker. If you have a new idea for a new feature that has not been listed anywhere, you are free to go ahead and implement it. We only ask that if you do wish to have the feature merged with the *master* branch that you communicate with us first, mainly to prevent possible duplicate works.

CHAPTER 14

History

14.1 Version 0.2

14.1.1 Release 0.2.0

release date: TBD

14.1.2 Release 0.2.0-beta1

release date: January 25 2016

First beta release of the 0.2 version.

The majority of the change in this version compared to the previous version is internal. Some of the more important changes are:

- The `sentinel` package has now been split into `sentinel`, containing generic functionalities and `sentinel-lumc` containing LUMC-specific pipeline support. This separation is not yet complete, since a part of the user configuration still refers to LUMC-specific functionalities. Instead, it is meant to pave way for a complete separation which is expected to happen in future versions.
- A new type for expected errors called `ApiPayload` was created to replace the previous `ApiMessage` case class which contains only error messages. In addition to error messages, `ApiPayload` may also contain a function that returns a specific HTTP error code. This allows for a given error message to always be tied to a specific HTTP error code.
- The main pipeline upload processing function in `RunsProcessorr` has been renamed to `processRunUpload` and is now expected to return `Future[ApiPayload \/ RunRecord]` instead of `Try[BaseRunRecord]`. Related to this change, this version also makes heavier use of the scalaz library, most notably its disjunction type. This allows for a nicer composition with `Future`, which has resulted in changes across database-related functions to use `Future` as well.
- The base functions in the `StatsProcessor` abstract class underwent a major refactor. In this version, the same functionality was achieved using only two generic classes `getStats` and `getAggregateStats`. Future versions are expected to bring additional changes most notably to the MapReduce step, since newer versions of MongoDB supports most (if not all) of the metrics using its aggregation framework only.

There are also some indirect changes related to the code:

- Sentinel now comes with minimum deployment setup using Ansible. This can be optionally run in a Vagrant VM with the provided Vagrantfile.
- The required MongoDB version is now 3.2 instead of 3.0.

The complete list of changes are available in the commit logs.

The user-visible changes are quite minimum. Aside from some LUMC pipeline-specific changes, the most visible changes are:

- The *library* nomenclature has been replaced with *read group*. The previous use of *library* was incorrect and the current one is more in-line with popular next-generation sequencing tools.
- There is now a URL-parameter called `displayNull` which allows for clients to view missing JSON attributes as `null` when this value is set to `true`. When set to `false`, which is the default value, missing attributes will be omitted completely from the returned JSON payload.

14.2 Version 0.1

14.2.1 Release 0.1.3

release date: October 13 2015

Bug fix (upstream) release:

- Update CollectInsertSizeMetrics summary object. In some cases, it is possible to have a single end alignment still output a CollectInsertSizeMetrics object in the summary file wit null values, as opposed to not having the object at all.

14.2.2 Release 0.1.2

release date: July 14 2015

Bug fix (upstream) release:

- Improve summary file parsing for Picard CollectAlignmentSummaryMetrics numbers. In some cases, the number of total reads and aligned reads may be 0. In that case, we use the BiopetFlagstat value instead.

14.2.3 Release 0.1.1

release date: July 11 2015

Bug fix release:

- Fix bug caused by the Gentrap summary format having a different executable entries for JAR and non-JAR files.
- Documentation improvements (typos, etc.).

14.2.4 Release 0.1.0

release date: July 6 2015

New version:

- First release of Sentinel, with support of the Gentrap pipeline.