Smart Speed Cameras

[CSC8110] Cloud Computing

Pedro Silva

15-12-2016

Development Tools

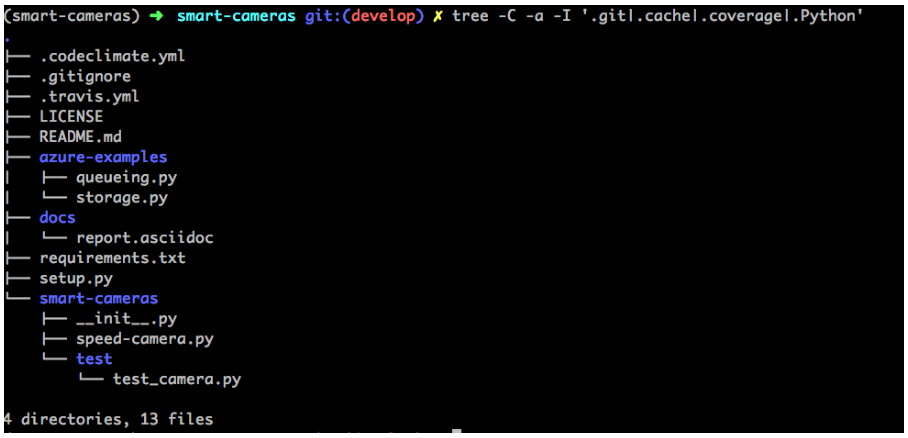
As suggested in the coursework document, I took on this challenge not only to discover how we can achieve cloud computing using Azure, but also to explore new software tools and programming languages. Although I've previously worked with *python*, I was still unfamiliar with how packaging, testing and development is accomplished in a continuous, collaborative and systematic way. Therefore, I chose to carry out this project using *python* to understand how project directories should be organized, the convention when it comes to naming directories, source and test files, find out the equivalent build tool of Maven for *python*, how to publish a library, etc.

To that extent I introduced the following technologies:

- **GitHub with git-flow**: A solution for hosting projects and building software collaboratively. Git-flow is a git extension that allows you to operate on a repository following Vincent Driessen's branching model.
- **Travis CI with CodeClimate**: Provides continuous integration on top of Github projects, i.e. on successful pushes to the remote repository (Github) a build is triggered on a fresh containerized environment (within Travis' public cloud) which installs your project and runs the entire set of test suites. Code coverage reports generated by pytest-cov are submitted to CodeClimate for assessment and developer feedback.
- **Virtualenv with virtualenvwrapper**: Used to provide separation between python projects being developed on the same machine, which may have different or even conflicting requirements.
- **Setuptools**: Used to package your python project/library so that it can be easily installed easily across heterogeneous systems.
- **Py.test with code coverage**: Tools for automated testing which generate test coverage reports.
- **Sphinx with autodoc**: Generates documentation for your project in similar fashion to javadoc.

I started by setting up the project layout and skeleton, as well as configuring most of these tools so that I can then concentrate primarily on the programming and logical components of the project. Most of these development and integration tools were suggested by Jeff Knupp on a blog post entitled "Open Sourcing a Python Project the Right Way".

At the end of this process and running the azure examples, I obtained the following file structure:



```
(smart-cameras) ➜  smart-cameras git:(develop) ✗ tree -C -a -I '.git|.cache|.coverage|.Python'
.
├── .codeclimate.yml
├── .gitignore
├── .travis.yml
├── LICENSE
├── README.md
├── azure-examples
|   ├── queueing.py
|   └── storage.py
├── docs
|   └── report.asciidoc
├── requirements.txt
├── setup.py
└── smart-cameras
    ├── __init__.py
    ├── speed-camera.py
    └── test
        └── test_camera.py

4 directories, 13 files
```

Figure 1. Basic project layout and directory structure

Software Design and Implementation

When using python for this job, one could first think about coding it in a more ad-hoc style, i.e. as a large script or series of connected scripts. However, I'm often interested in developing a cohesive set of blocks that can be re-used later or extended by exposing an Application Programming Interface (API). In fact, as most programming languages, python can also be used easily for this intent, especially when considering that it is an object-oriented language. Therefore, I tried at each step to encapsulate all required behavior in classes, while modeling the entities outlined in the coursework (e.g. speed camera). Although this sort of process often requires extra time spent going back through the code, writing proper tests and refactoring, it usually pays of in the end because we have a more elegant and developer-friendly solution. The development process took place as follows:

1. Write code (whether is template of class/method or advanced functionality).
2. Write test.
3. Fix code to pass the test.
4. Refactor and clean code.
5. Take a look at the whole picture - refactor, improve code accordingly.
6. Verify that tests pass.
7. Iterate.

I considered using Test-Driven-Development (TDD), but I feel more comfortable using a more flexible back and forth methodology (which arguably ends up being very similar to TDD if we consider the iterative refactoring step in TDD).

Task 1: Smart Speed Camera

For this task I modeled the following entities using classes:

SpeedCamera

Represents a speed camera located at a given street and city in on of two states: active or not active. If active then it has an associated speed limit and simulates vehicle sightings according to a poisson process with rate λ by sampling arrival times from an exponential distribution with rate 1/λ. A speed camera can be activated and deactivated using the respective methods. Upon activation and deactivation it publishes itself by sending a message to an azure topic. The topic name is given by a constant defined as a class variable (reference). Once activated the camera enters an infinite loop that can be terminated by calling *deactivate*. The status of a *SpeedCamera* instance can be determined by querying the value of boolean attribute *isActive*. While active, the camera computes the next vehicle arrival time, sleeps until that time has elapsed (can be woken up by *deactivate*) and then publishes the sighting onto the same topic as before. Object serialization is done using a simple **json** encoding. Deserialization is only supported for *Vehicles* and not for *SpeedCameras* as the first is an immutable object while the second is not. Besides camera and vehicle information, extracted directly from the classes' attributes, we also passed an extra value denoting the kind of event being published: camera activation, camera deactivation or vehicle sighting. However, as we will see on the next section, Azure's *Rules* could be used more effectively to provide message filtering (based on the different events that generated them). And indeed they were, but I only implemented them at a later stage. Finally, offline operation is supported but only during program execution (messages are stored in a queue and flushed once there on the next successful publish attempt). Ideally, we would backup messages in a local database such as SQLite, in order not to lose any information, but I felt that this was unnecessary at this stage.

Vehicle

Represents a vehicle sighting, with attributes *plate*, *type* and *speed* of vehicle. Plate is simulated by uniformly sampling 2 letters and 2 numbers followed by 3 letters. Speed is simulated by sampling from a normal distribution with standard deviation equal to one fifth of the speed limit and mean equal to the speed limit minus one standard deviation. This implies that about 84% of all sightings will be below the speed limit (CDF of normal distribution at mean + standard deviation = 84%) while the rest of observations will be speeding vehicles. At the same time, we restricted the values of speed limit to equal or great than 30 (which is not a totally unrealistic assumption), in order to ensure that we would not sample negative speed values. Therefore, at the speed limit of 30 units (units can represent either kilometers or miles per hour) the probability of sampling a vehicle with speed less of greater than 10 units is equal to approximately 0.003167 % (CDF at mean - 4 standard deviations).

AzureHook

A wrapper class for azure's service bus that tries to simplify method calls to Service Bus. Ideally we would like our speed camera to be completely unaware of who the cloud provider is and operate exactly the same way whether the provider is: Microsoft, Amazon, Google or other. Providing loose coupling between classes would enable us to use one class irrespectively of the other, and changing provider would not mean we had to modify any of the code we wrote for speed camera. This could be accomplished by writing an interface or abstract class that provides the methods for cloud integration, such as authentication, sending and receiving data to and from some component (queues and topics in Azure's case), publish and subscribe entities, store and query information, among others. The camera could then take an instance of the abstract class as an input (in the constructor for example) and access the methods it requires. AzureHook would then be an implementation or child class of such interface or abstract class. However, this functionality would be nice to have, it is hard to normalize a set of methods across a single abstract class in such short period and it would not be very useful since only one implementation would be provided - Azure's. *CloudHook* represents an attempt to build such abstract class in python.

At this stage I also started thinking about the execution model of the program given the requirements about speed cameras:

- Multiple *SpeedCameras* should be able to operate concurrently
- Should be able to change the location of a camera while it is active
- We should be able to restart the camera (without killing the program?)

A simple approach would be write a simple two-threaded program (which could be implemented as the main method for the file that defines the speed camera class) which creates an instance of the *SpeedCamera* and activates it on a new thread. Then it reads input from the command line, prompting for actions to relocate, restart and stop the program. A more interesting approach would be to have a camera manager application that would serve as a broker between the user and the cameras and let you operate multiple cameras simultaneously on the same terminal. However, the ideal solution would be to decouple the camera manager application from the user issued commands. The camera manager would be deployed on a web server and exposed through a server socket. The command-line client would then connect itself to the remote application using a client socket and issue commands over the network. This would enable the server and client applications to live on different machines. For instance, the camera manager application could be deployed to the virtual machine created in Azure, while the command-line client could be installed in your machine. Similar functionality could be achieved using inter-process communication instead. My goal was to implement such elegant solution. I implemented a camera manager and a part of a command-line client but with still the communication between the two missing, I felt that I would not have enough time to do the main tasks of the coursework, so I rolled back to the simplest of approaches.

Task 2: NoSql Consumer

The topic defined in *SpeedCamera* can be subscribed in order to receive messages published by speed cameras. We can represent a generic topic subscriber by an object that takes a topic name, a subscription name and optionally a *Rule*. This object takes those values to create an Azure subscription and optionally associates a rule with that subscription. Just like *SpeedCamera*, an *AzureSubscriber* can be activated and terminated. So the same execution model applies here. While active, it tries to obtain a message from the subscription. If a message is available it is handed to an abstract method *onNewMessage* which child classes must implement. Otherwise, it sleeps for an amount of time determined according to an exponential backoff algorithm, which takes in the number of times a message was unavailable. Therefore, the time in seconds to sleep at trial *n* is given by $n = 2^{n} * 0.1$. This value is capped of at 12, where the time to sleep is of 6.66 minutes.

A possible implementation of the generic subscriber is a subscriber that receives the message, maybe do some processing on the information given, and later stores that result on an Azure Table. This is exactly the case study that is asked of us in this task. Therefore, we created *PersistentSubscriber* as a subclass of *AzureSubscriber*, inheriting all the methods (not all methods needed to inherited but that would require name mangling which is not easy to perform without previous experience). An object of this class is additionally given a table name in the constructor, which uses to create the table in Azure Table Storage if non existent. At the moment this class connects to the API in rather a hard-coded (and non-safe!!) way, but that could be improved by reading a configuration file set with the correct read and write permissions. This class is also abstract and provides an abstract method to be implemented by child classes which receive a dictionary as an argument and must return an Azure Table *Entity*. On a new message the *PersistentSubscriber* will insert the Entity into the table for you. It also provides a method for you to query the table given a query string, as well as methods to retrieve or flush an entire partition given the partition key. This made building queries for the following tasks, much easier.

We created two child classes of *PersistentSubscriber*:

- *CameraRegister* - Responsible for processing events of camera activation and deactivation and inserting them into a table.
- *VehicleRegister* - Responsible for processing events of vehicle sightings and inserting them into a

table.

On each of these classes we defined the Partition and Row Keys, as well as other information we wanted to persist. At this point there was a series of design considerations that needed to be taken into account. The combination of partition and row key must be unique and these should also be chosen in function of the kinds of queries performed. Furthermore, more efficient queries are always aware of the value of the Partition Key. These are listed on the following link.

For both tables, we used an arbitrary string for partition key, representing the underlying event (e.g. 'CameraActivation', 'VehicleSighting'), and the timestamp at the moment of the event (with millisecond precision) as the row key. I considered using a unique combination of keys such as camera id and vehicle id or timestamp, but I found that queries such as returning all sightings or registrations would be hard to perform. Therefore, considering the examples given in the website, the value of partition key should represent some very high level component while the row key should be represented by something incremental and ordered by nature, for which time was a good fit. Furthermore, I made arguably a strong assumption in this case: no two cameras or cars are activated/observed at exactly the same time. An assumption which works well, simulation-wise, but might not in reality. I also introduced other fields on each case as fit, e.g. vehicle speed, location, etc. It was a shame that I could not use nested dictionaries instead of an Azure *Entity*.

In my opinion, one way to better model the problem would have been to use a single Table with different partitions, where each partition represents a different entity or event relative to the Speed Camera, i.e. activation, deactivation, sighting, speeding, etc. That way we would use less resources (tables) and get exactly the same functionality, with better readability, query-wise because the partition key in this case can be something completely arbitrary as there is only a single partition per table.

I also read about the Azure Table query retrieval limitation: Azure only lets you retrieve up to 1000 entities per query. Therefore, applications querying on tables with more than 1000 entities stored, need to be adjusted to take that into account (using the **marker** parameter in the *TableService.query_entities* function).

Task 3 and 4: Query app and Police Monitor

Given the implementation described before, both of these tasks were very easily implemented. Speed camera activations are easily obtained by querying on partition key that matches the value for activation events on the table *speedcameras*. Slight modifications on the *Vehicle* and *SpeedCamera* classes enabled us to identify speeding vehicles and include this information on the publication in a way that it was possible to use the feature Subscriptions with Filters. To that extent, class *PoliceMonitor* subclasses *PersistentSubscriber* and writes speeding sightings on a different table. As we included rules in the generic class, associating filters with subscriptions is straightforward. The query for retrieving priority sightings from the table is accessible through the *PoliceMonitor* class.

Task 5: Vehicle Check

In this task, we created class *VehicleInspector* which is a child class of *AzureSubscriber* instead of *PersistentSubscriber* because we are not persisting any data, but instead just simulating an analytical task. Upon receiving a vehicle sighting, the *VehicleInspector* puts the request for processing a vehicle on an internal queue. Concurrently, the *VehicleInspector* is running a dedicated thread that takes requests from the queue and simulates the analysis process as specified in the coursework. *VehicleInspector* can be terminated at any time that the owned processing thread is killed with it. It follows the execution model introduced for *AzureSubscriber*.

Task 5b: Azure Auto-Scale

I did not perform this task due to time constraints.

Final Considerations

All in all, I think this was a great project as it required several problem solving skills and engaged critical thinking. Although it was great to have the possibility to explore diverse implementation options and tools, I feel that it would have been useful to have a clear goal at the end of the project. In other words, I think that it should have been clearer from the start what specifically we should be able to demonstrate working at the end of the project. This may be even more critical for people that do not come from a stats background. Alternatively, we could have been provided the skeleton of a command line client, which we would have to integrate with our program. We could be given the client-side program and just need to implement the server-side and ensure that the command line interface client works.

In my opinion, the documentation for Python's Azure SDK was lacking and in general there was not a lot of examples or questions and answers in websites like *StackOverflow*. Nevertheless, it was a great opportunity to learn how real distributed system work and how to quickly set up applications in the cloud. Once more, I saw how important tools like Git, Travis and virtualenv are for software development and how useful it can be in the long run to take a step back to try to improve and clean up your code and class models.

Development proved to be essential in the detection of issues at early stages. Code coverage also provided a good overview of how well the tests were covering the written code. Continuous integration ensured that I kept the dependencies of my project to date and that the entire test suite was triggered everytime a push to Github was performed. Finally, codeclimate was difficult to integrate as it did not work properly outside the *master* branch. As I was using **git-flow**, commits were made against the *develop* branch and only release-ready code was merged into master, only then triggering the codeclimate feature. Unfortunately I ended up not having much time to try out Sphinx, write better in-file comments and write a proper README file.