This degree project has been performed on behalf of
Sillens AB – ShapeUp Club
Supervisor at *ShapeUp Club – Nicolas Grasset*

Scalable platform for health service integrations

Skalbart system för integration med hälsotjänster

Olle Lind & Joakim Hammer

Degree Project in
Computer Engineering
First cycle, 15 hp
Instructor at KTH: *Anders Lindström*
Examiner: *Ibrahim Orhan*
TRITA-STH 2013:32

Royal Institute of Technology
School of Technology and Health
136 40 Handen, Sweden
http://www.kth.se/sth

Abstract

This thesis was performed at the company ShapeUp Club located in Stockholm, Sweden. ShapeUp Club offers a digital calorie counter service for the web, iOS and Android with data synchronization across the platforms.

ShapeUp Club wants to provide their users with the option to synchronize data between ShapeUp Club and external health services. The objective for this thesis has been to develop an extension to ShapeUp Clubs current backend platform where new external health services can be plugged-in quickly and scalable. External partner APIs will be examined and implemented in the system to validate the functionality of the system. The amount of code needed to plug-in a service should be as minimal as possible for a developer to quickly add another service. To allow for scalability the platform also needs to adapt logic for how often users should be allowed to poll for data from their connected services, to minimize the database load for all parts.

To handle these demands, an extension to ShapeUp Club's current backend solution was built using the Django framework for Python.
By providing a generic base class that new services inherit from, the amount of code necessary for implementing a new service is reduced to methods for API-requests, authorization and serialization of data.
To reduce the number of redundant poll requests, users are placed into groups. Each group is a cluster of users with similar frequency of updates.
Django's cache framework is used to handle the concurrency of the sync tasks, which locks a user from syncing the same partner in parallel.

Sammanfattning

Detta examensarbete har utförts hos företaget ShapeUp Club i Stockholm.
ShapeUp Club erbjuder en digital kaloriräknare för webben, iOS och Android med synkronisering av data mellan dessa plattformar.
ShapeUp Club vill kunna erbjuda sina kunder möjligheten att synkronisera data mellan ShapeUp Club och andra externa hälsotjänster.

Målet med detta projekt har varit att implementera en ny tjänst till ShapeUp Clubs nuvarande backend-lösning där externa hälsotjänster snabbt och skalbart kan implementeras. Externa hälso-API:er har utvärderats och implementerats i samband med utvecklingen av den nya backendtjänsten, för att validera dess funktionalitet. Mängden kod som behövs för att implementera en hälsotjänst bör vara så minimal som möjligt för att utvecklare snabbt ska kunna lägga till ytterligare tjänster. För att systemet ska vara skalbart måste logik finnas för hur ofta användare ska tillåtas att fråga efter data mot de tjänster de har valt att synkronisera mot. För att tillfredställa dessa behov har en utökning av ShapeUp Clubs nuvarande backend-lösning byggts med ramverket Django för Python. Genom att ha en större, generisk klass som nya implementeringar ärver från så har mängden nödvändig kod för varje hälsotjänst-implementering minskats till metoder för API-anrop, autentisering och serialisering av data.
För att minska antalet "onödiga" poll-anrop så placerar vi användare i olika grupper beroende på om deras poll-anrop frekvent återvänder utan någon ny information. De olika grupperna bestämmer sedan hur länge användarna måste vänta innan de tillåts göra nya poll-anrop.

Prelude

We would like to thank ShapeUp Club for giving us the opportunity to write our bachelor thesis on their behalf in the best possible working environment. A special thanks goes to our supervisor at ShapeUp Club, *Nicolas Grasset*. First of all for specifying such a fun and motivating assignment, but also for his awesome technical help and support.
At last we would like to thank our mentor at KTH, *Anders Lindström,* for always finding time to meet us and give great constructive criticism on the project report progress.

The Royal Institute of Technology - School of Technics and Health, Campus Haninge.
2013-05-06

Olle Lind

[illegible]

Joakim Hammer

[illegible]

Table of content

1. Introduction

1.1 Background

This project has been made on behalf of Sillens AB (ShapeUp Club) and performed at their office in Stockholm, Sweden.

1.1.1 External partners

ShapeUp Club is currently integrated with an external health service (partner) to allow users to synchronize data between the two services. This integration has its own synchronization application that needs to be running continuously.
In the near future ShapeUp Club would like to integrate with several other partners. It is time demanding to build these integrations from scratch and to tailor for the partners specific APIs. Therefore it is desirable to have a common code base, that can easily support new partner integrations with a minimal amount of partner specific implementation code.

1.2 Project goals

For internal systems there are products to keep files and data in sync such as Windows sync center. The main goal of this project is to build a common solution for integrating with several external services and keeping data between them in sync. This problem cannot be solved with any product on the market today.

The following demands should be solved to accomplish the this goal:

- New partners should be able to be plugged in without having to alter the code base more than defining a new module for that partner and specify partner specific information.
- Partner modules should be completely uncoupled against each other.
- The system should handle authentication information for each user and its connected partners.
- The system should be integrated with ShapeUp Clubs current backend and be able to serialize and store fetched data from partners into ShapeUp Club's current data layer.
- The system should be able to find new data in ShapeUp Club's database and determine if and where it should be pushed.
- If a partner is unable to handle pushed data, it must be re-sent at a later time.
- User data and behavior will be studied to adapt the sync frequency for users to lower the number of redundant poll requests.

The system is supposed to be able to handle at least twenty different partners and approximately a hundred thousand users per partner.
New partner integrations can't be allowed to reduce the system's overall performance exponential for the system to be scalable.
It is not critical that new data on either ShapeUp Club's service or any of the external services are synced immediately, so the system should not continuously poll for information for each user and partner.

- Research will be made to establish algorithms and methods to determine how often users need to sync their data.

Since the service will be a part of a larger system, it is important that it can be tested on command along with the rest of the system.

- The system's abstract layer should be covered by unit tests to ensure system stability when maintained by developers later on.

1.3 Limitations

The project will have these limitations to set a realistic project size for the given timespan:

- A maximum of five partners will be integrated in the finalized product.
- ShapeUp Club's mobile clients don't require a production ready interface for authentication with the system.
- An open API against ShapeUp Club will not be written for partners to push data to.
- No callback methods will be written, even for external APIs that supports them.
- Unit tests to assure availability to sync against specific partners will not be built.

1.4 Solutions

After searching the web for a solution to handle multiple APIs in a modular and generic way, there was no obvious or widely used solution that would meet the goals. It is therefore needed to write a tailored solution.

The platform and programming language that will be used in this project is Python with the framework Django together with a MySQL database. The main reason for choosing these techniques are that they are used in ShapeUp Club's current backend solution, which will ease the integration of this project's product. Also, Django is a well-established platform that uses uncoupled internal "Apps" that suits the purpose well.

Several Python libraries will be used for different features instead of writing them from scratch, one of them is Celery that will be used to queue tasks that should be executed in the near future, there are also libraries to handle OAuth authentication such as Rauth.

The service will be tested using unit tests written alongside the code by using Python's built in library for unit testing.

2. Current situation

2.1 ShapeUp Club

Since 2008 Sillens AB has had ShapeUp Club as its only product. ShapeUp Club is a digital calorie counter available on the web, iOS and Android. It can be downloaded and used for free except for some functionality that requires the user to subscribe for "gold-membership".
ShapeUp Club has over 4 million registered users and 70.000-100.000 daily active users.

2.2 Current partner integrations

ShapeUp Club has integrated RunKeeper's Health Graph API to allow their users to synchronize weight and exercise data between the two services. About 25.000 of ShapeUp Club's users have chosen to connect their accounts to their Health Graph account.

To synchronize between the services, a stand-alone application is constantly running through all users that are connected to Health Graph and performing a poll request for each user to see if there is any new data to fetch.

2.2.1 Integrating with more partners

If more partners need to be integrated then all of them will need the same kind of applications to run across their respectively users. It would be time demanding to build separate applications for each integration, but also inefficient since there is logic in the synchronization process that is part of each integration (such as querying the database for new user data). A more scalable solution would be a system that handles all the integrations and only perform logic that applies to all partners once per iteration. Developing such a system is the key goal of this project.

3. Theoretical prerequisites

This chapter will brief about various techniques used to build and solve the faced problems as well as describe the different external APIs that will be integrated and what kind of information they can provide.

3.1 Python

Python is a high-level object-oriented programming language founded by Guido van Rossum. It is an interpreted language with a clear syntax for high readability and understanding of the code. The language is built on modules to easily import or remove libraries to work with [1,2].

Python has a functionality called virtual environment [3], which allows you to isolate all python libraries into one folder and for different projects. This allows you to safely upgrade any libraries without affecting other virtual environments and their functionality.

Several big companies uses Python in their backend solutions, among these are Yahoo Maps and Spotify [4].

3.2 Django

Django is a high-level framework written in Python with the purpose of enabling developers to build web applications in a structured yet quick pace.

A Django project is built in MVC-structure (Model View Controller) and has a built in ORM (Object-relational mapping) against several different types of databases, such as SQLite, MySQL, PostgreSQL etc.

Django uses a model-first principle, meaning that the project's models will be used to define and map the project's database schema.
A Django project consists of one or more independent "Apps" that can easily be plugged in and out of a Django project.

There is built in functionality in Django to automatically create administration pages to overlook the database. One can easily create forms for the data model so that anyone, even without database knowledge can add, edit or remove data from the database.

3.2.1 Model based data structures

In order to define data structures, Django uses models for single definition of what data an entity holds. Django models are defined as Python classes and needs to subclass the django.db.models.Model class. The model's attributes then needs to be specified as one of django.db.model's predefined abstracted data types. A string field with the name "value" would look something like this:
value = models.CharField().
By writing the models in this manner, Django will automatically map the model against a database table and create database-access APIs. Models may then be stored and retrieved from persistent storage without having to write SQL

queries. If a model has an attribute of another model type, Django will create a column with foreign keys to the other model's database table. When accessing the foreign object from the code, one simply has to retrieve it as if it were a local attribute of the original object [6].

3.3 Message brokers

Message brokers are used in systems to uncouple parts of the system from each other to allow for scalability. The message broker is a component that handles communication between the different parts of the system instead of having the parts tightly coupled.

The system's different applications send messages to the message broker, which then routes the message to an endpoint worker. Imagine a system that has a number of heavy tasks that should be executed on different worker servers to balance the load. Having a message broker between the system and the cluster of worker servers allows the system to not know about which actual server that executes the task [7].

There are a number of widely used message brokers available for download such as ActiveMQ, RabbitMQ, HornetQ, Apollo and QPID.
Benchmarks made between them shows that RabbitMQ is about 3-4 times faster than the others when it comes to huge numbers (20.000-200.000) of small messages (32-1024 bytes) [8].

3.3.1 RabbitMQ

RabbitMQ is an open-source message broker written in Erlang OTP, made available for all major operative systems (Mac OSX, Windows, Linux, BSD etc) and a huge number of programming platforms (Java, Python, PHP, .NET, Ruby etc).
RabbitMQ offers decoupling between applications using asynchronous message passing to send and receive data. RabbitMQ can queue tasks and delay them before routing them to a worker in a server cluster [9].

3.3.2 Celery

Celery is a Python library that handles connection to a running RabbitMQ server and abstracts the usage of it.

3.4 Health APIs

There are a lot of services that specializes in tracking exercise, nutrition or body measurements, either through web/mobile applications or through physical devices such as Bluetooth bracelets. Some of these services offer access toward their collected data or availability to insert data through REST APIs. The service built during this project should be compatible to connect and use APIs that offers fetching or pushing of:

- Consumed nutrition data
    - Fetched data must contain a calorie intake
- Executed exercise data
    - Fetched data must either contain a total amount of burned calories or the duration of the exercise together with the amount of calories burned for a time unit

- Body measurement data such as weight, length, waist measure etc
    - Fetched data must be in metric units

Finally all fetched data must have a corresponding date for when it was consumed, executed or measured.

3.4.1 Withings

Withings offers their customers a digital scale that measures weight, body fat and BMI. The information is then transferred through wifi to Withings' service that stores the data in their databases [10].
Fetchable: Weight, length, bodyfat in percentage and KG
Authentication: OAuth1.0 with non-renewing access-tokens

3.4.2 Health Graph powered by RunKeeper

HealthGraph, developed by RunKeeper is a centralized platform for storing and retrieving health information in form of consumed nutrition, executed exercises and body measurements. HealthGraph has over a hundred different partners that collects and pushes data to the platform [11].
Fetchable: Exercises, nutrition and various body measurements.
Authentication: OAuth2.0 with non-renewing access-tokens

3.4.3 Fitbit

FitBit uses digital bracelets that gathers information such as how many steps the wielder has taken and the distance it has traveled. Then the information is calculated into how many calories have been burned and stores the information in their database [12].
Fetchable: Exercises, weight, waist, chest and fat measurements
Authentication: OAuth1.0 with non-renewing access-tokens

3.4.4 Moves

Moves is an application for iPhone that seamlessly records the wielder's daily route of walking, running or cycling in terms of steps taken and distance traveled.
Fetchable: Exercises
Authentication: OAuth2

3.5 OAuth

OAuth is a client-server protocol allowing clients to access other services without sharing the same user credentials between the services [13]. The OAuth protocol also provides a way of identifying which client is requesting access to the data. The user credentials are after an authentication flow merged into an access token that must be sent with each request.

Today there are two different OAuth versions of the protocol, called OAuth 1.0 and OAuth 2.0. OAuth 1.0 was published in December 2007 and in April 2010 it was published as RFC 5849. The main problem with the first version of the protocol is the complexity of sorting parameters and encoding them. Eran Hammer, the main author of OAuth 1.0 describes the problem like this: "*For example, developers can quickly write Twitter scripts to do useful things by using their username and password. With the move to OAuth, these developers are now forced to find, install, and configure libraries in order to accomplish what was* *before possible with a single line of cURL script.*" [14].

OAuth 2.0 was published in May 2010 and later on published as RFC 6749 and 6750 in October 2012. One of the enhancements with the new version is that there's no more encryption and sorting of parameters and therefore no need to use an external library for making API calls. The token can be sent as a parameter or more preferably, the Authorization header [15].

3.6 Rauth

Rauth is a Python library designed to handle the OAuth protocol [16]. It supports both version 1.0 and 2.0. By using Rauth you can easily authenticate to OAuth 1.0 services without worrying about encrypting and sorting of parameters.

3.6.1 Session

Rauth's session object is an object that is authenticated to an external service. By using this object all you need is the URL to the partner service and the parameters. Rauth will sort and add protocol specific parameters to the URL.

3.6.2 Service

The Rauth service is a wrapper around a session object. The purpose of the service object is to retrieve a session object that can be used to make authenticated API calls. The parameters to create a service object vary depending on what OAuth version to support. Both versions needs a client id/key and client secret. Other parameters needed are different URLs depending on the protocol.

By giving all these parameters to the service object, several steps of the authentication flow will be an abstracted layer that Rauth will take care of.

4. Implementation

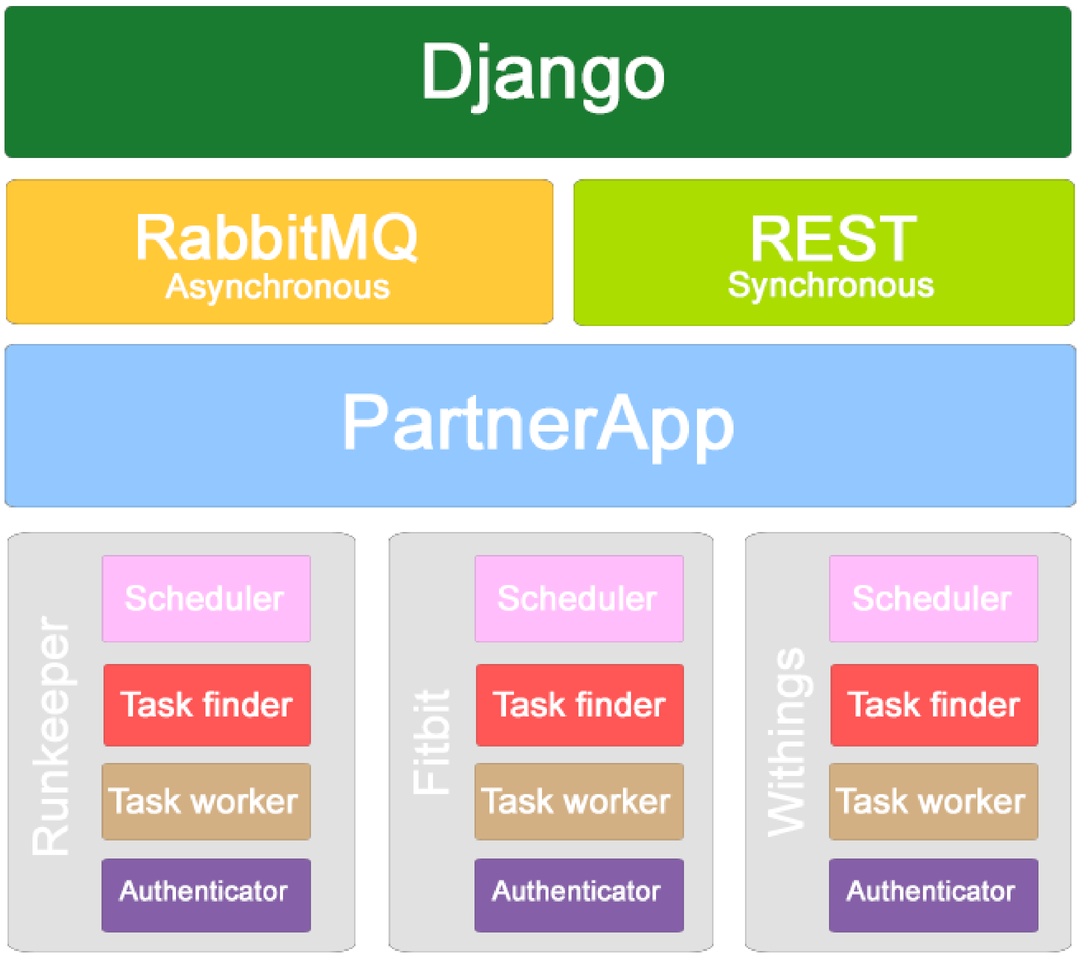


Figure 4.1 System layer hierarchy

Researching about how to build a scalable system leads to several different tips and techniques. Authors say scalability is about partitioning, concurrency, parallelization and modularity [20, 21]; and so the system has been designed to comply with all of the above.

As shown in figure 4.1, the system has been divided into layers that are independent of each other in horizontal direction. Communication between modules and layers are only performed in vertical.

The **first layer** consist of Django, the platform the service has been built upon.

The **second layer** is input sources that communicate with the service and requests synchronization to be made. RabbitMQ triggers the sync at regular intervals while the REST API synchronously handles authentication requests.

The **third layer** is the service interface "PartnerApp", with methods to trigger synchronization and authenticate against partners. PartnerApp is also the controller of the service, delegating messages between modules of the underneath layer.

The **fourth layer** is the partner synchronization logic, which consists of four independent parts that are necessary to properly sync.

- *Scheduler* looks up if a user should synchronize
- *Task finder* fetches tasks that the sync should process
- *Task worker* performs the actual sync tasks against the partners
- *Authenticator* handles authentication against partners

This chapter will describe the components of the service architecture shown in the illustration 4.1 in more detail and important solutions for making the codebase generic.

4.1 Using python and the Django platform

4.1.1 Virtual python environment

Newly installed Python packages are placed under "/usr/lib/python2.7/site-packages" and are shared by all applications that tries to import the package. Imagine having one application that requires a certain version of a package to work properly, while another application needs the latest version of that same package. Only one of the applications would be able to function correctly. By using the python package *Virtualenv* one can create isolated python environments [3].
Typing "virtualenv python-environment" in the terminal creates a new isolated environment called *python-environment*. Then execute "source ./python-environment/bin/activate" to start using that environment instead of the global one. Now when installing, upgrading, removing or importing a Python package it is performed on the new environment.

4.1.2 Using Django

The product of this project is built as a Django app because ShapeUp Club's current backend solution is built on the Django platform.

By branching the current backend's git repository you get a local copy where you could add the new "partnerapp" by executing the terminal command:
"python manage.py startapp partnerapp".
Where "manage.py" is the django-projects manager file.
To plugin the app to the project you simply add its name to the "installed_apps" setting in the project settings file.

To be able to run the project, you have to create a local database that matches the project's settings and model schema. The current project was set to use MySQL as database manager. A local MySQL database was created with the same name as the Django project settings file was set to connect to. To map the database schema against the models of the Django project, you simply execute the command "python manage.py syncdb", that creates all the necessary tables to support the model structure.

4.2 RabbitMQ

To keep the system scalable, API requests can't be executed on the same thread or even the same physical machine as the sync logic and risk blocking the server from performing the sync logic. To achieve this, a RabbitMQ server is set up, which can be accessed from the code through Celery. The part of the sync process that does the actual API requests and stores information are called via RabbitMQ, which makes the codebase uncoupled against which actual machine executes the task.

Before Celery can tell RabbitMQ to execute a task it must be defined as a method using the decorator *@celery.task*. The parameters for the task method can't be object references since the object scope isn't shared across different machines. Only primary data types may be used. So instead of sending the actual partner and user objects that the task should process, their database ids are sent so they can be rebuilt on the worker machine by querying the database.
Invoking a RabbitMQ task through Celery is as easy as:
task_name.delay(parameters).

RabbitMQ is also configured to run tasks on a regular basis such as the sync process and searching ShapeUp Club's database for data that should be pushed to partners.

4.3 PartnerApp

As a controller interface there is a class called PartnerApp.
PartnerApp's public methods are called from RabbitMQ and through HTTP-requests.
PartnerApp contains methods to:

- Synchronize a single partner for a user
- Synchronize all partners for a user
- Synchronize all partners for all users
- Synchronize all users for a partner
- Delegate authentication to partner modules
- Fetch data from ShapeUp Club's database that needs to be synced

To access this interface from another Django App in the ShapeUp Club platform one simply has to import the PartnerApp module and execute any of the methods like this:
from shapeup.apps.partnerapp import partnerapp
partnerapp.syncAllPartners()

4.3.1 SyncAllPartners

SyncAllPartners is the name of a method that will be executed on a regular basis to perform synchronization for all users and partners that requires it.
SyncAllPartners starts of by fetching a list of names of the installed partners (conveniently called installed_partners) from the projects setting file. A partner that isn't in the installed_partner list is not officially plugged in to the service.
It then performs sync for all users connected to each partner in the list.

4.4 Sync items

To be able to map entries in ShapeUp Club's database against respectively entries on the external services, a new data structure was implemented. Without this mapping, a partner could send the same object twice, and it would be stored twice instead of overriding it. One of the project's goals is that the new system should be separated from the rest of ShapeUp Club's data flow, so it's not allowed to alter the current data models and add the attributes that are required to handle this. Instead references to items that should be pushed to a partner are stored. Items that have been fetched from a partner are also stored with the partner's internal id.

4.4.1 NoSQL approach

This was implemented by creating a new table using a NoSQL approach. For example, an exercise item would be stored with a sync type called "exercise" and an internal id to the actual exercise in ShapeUp Club's exercise table. A food item would be stored with the sync type "food", but its internal id would map against an entry in ShapeUp Club's food table. So references to any type of entry from ShapeUp Club's database along with the reference to the external service's entry can be stored in the same table. The NoSQL approach was chosen to create a loose coupling against ShapeUp Club's database. By doing so, SyncItems for food, exercise and measurements can be stored in the same table without a strict coupled foreign key to a specific table.

4.4.2 Populating table

To populate this table, a method is used that query each sync type's table (exercise, food, measurement) and retrieves data that has been added/manipulated since the last time the method was called. This means that this method needs to be called before any data can be pushed to a partner. The items are stored once for each partner the user is connected to. To ensure that duplicates aren't created at the partner or an update request is sent instead of a create request, an integer is used that determines the actual state. The integer is updated when the item is pushed to the partner.

4.4.3 User disconnection

To be able to handle users that might disconnect from a Partner, delete previously sent items and then connect again, a decision had to be made regarding how to update the sync item.

Items that had been pushed to a partner should still be updated in the sync item's table. These changes should be pushed once the user connects to the partner again. This decision was made to meet the user experience of synchronization with the partners. A user would not want to find different versions of the same data.
Another issue that needed to be solved is if a user reconnects to the same partner but with different partner accounts. To handle this issue, something that is unique per account with the partners is needed to be stored in the database. This could be external user identification, for example. As long as the reconnecting user has the same user identification, the sync items will still be pushed to the

partner service. But if the user reconnects with different user identification, all the sync items for that user and partner need to be cleared.

4.5 Dynamic stored properties

Depending on which partner the user connects to, different properties should be stored. For example, one partner may only have the option to fetch weight data while another might have the options to store and fetch both weight and nutrition. If each possible attribute for every partner would be stored inside the model that connects a user and a partner, it would contain several unnecessary attributes that might never be used. That would require the model to have at least booleans for *push_exercise, poll_exercise, push_weight, poll_weight, push_nutrition, poll_nutrition* and more. Instead a table called PropertyType contains all the different properties that partners or users might require. When a user or a partner needs to store a property, an instance of PropertyType is created with a reference to the user or partner that owns it.

4.6 Partner superclass

To decrease the work of adding a new partner API to the system as much as possible, the common interface needed to be abstracted from the partner-specific classes. This was implemented by writing a base class that declares all methods that partner classes require and then define as many methods as possible as long as the implementation is acceptable for any given partner.
All the sync logic is implemented in a generic way inside the base class. Partner specific functionality, such as the actual API calls, parsing data from responses and authentication to the partner is implemented in the subclasses.

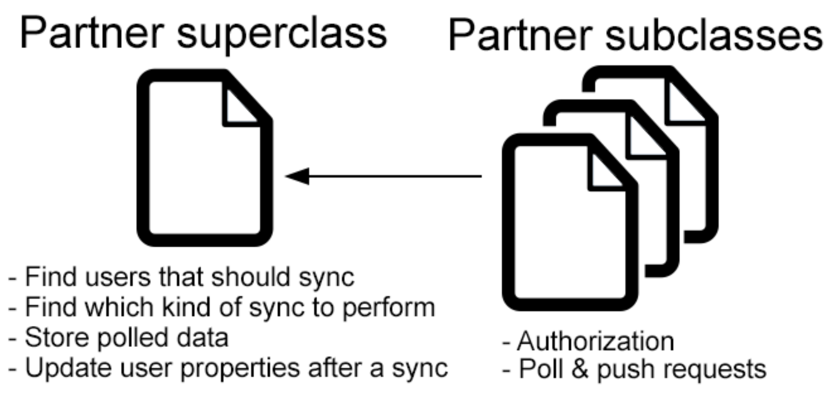

Partner superclass
Partner subclasses
- Find users that should sync
- Find which kind of sync to perform
- Store polled data
- Update user properties after a sync
- Authorization
- Poll & push requests

Figure 4.2 Partner subclasses inheritance

4.7 Scheduler

The scheduler is designed to retrieve users that are allowed to synchronize, either via push or poll. Some partners have restrictions regarding how often and how many API calls that are allowed and it is important to make sure these limitations are followed. Restricting the amount of API calls is also essential for the system's scalability. If all users were to sync every minute or so, the burden on the system would be unsustainable both for ShapeUp Club and the connected partners.

4.7.1 Retrieving users

The scheduler is inherited by all partner objects and does not need to be overridden. All users have two expiration timestamps per partner that determines when the user is allowed to synchronize. The timestamps represent when the push and poll request's blockade will elapse. When the sync starts the scheduler is called and will retrieve all users that has the push or poll timestamp expired.

4.7.2 Updating timestamps

When a sync has completed, the task worker will call for the scheduler to tell it to update the timestamps for the next possible synchronize. The task worker also sends two parameters that tell the scheduler whether or not the task worker actually made a successful push and/or poll.

ShapeUp Club has today more than 23.000 users that have connected their account to RunKeeper's Health Graph. But there are only about 7.000 users that synchronize any data per month. This means that more than 2/3 of the users are inactive during the month and there are many poll requests to RunKeeper that are redundant and doesn't return any new data. To reduce the number of redundant poll requests an algorithm needed to be written that flags inactive users that hasn't synced any new data for a while.

To achieve this all users are put in different groups depending on their activity level. A user may be in different groups with different partners. There are three different groups called group-5, group-20 and group-60, where group-5 is the starting group for all users. The number represents the sync interval, group-5 means that these users will be able to sync every five minutes.

When a user becomes inactive the user will be demoted down one level (e.g. a group-5 user will become a group-20 user). An inactive user is defined as a user that hasn't pushed or polled any data during a threshold of at least three weeks with a specific partner. This will continue until the user reaches the group-60, as shown in illustration 4.3.
An active user will always be in the group-5. That means that if an inactive user in group-20 or group-60 push or poll any data will be put in the group-5 and will stay there for at least three weeks.

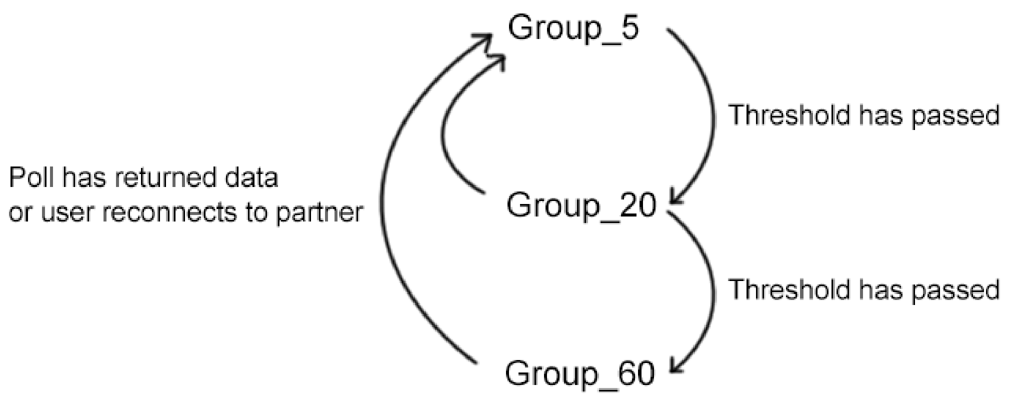


Figure 4.3 Assigning groups to users

4.8 Task finder

Each partner has a set of data types that can be synchronized and it is up to the user to choose which of those types to synchronize. For example, RunKeeper can

support synchronization of weight and exercises in both directions. From Withings, only weight and body fat information are fetched.
Lets say the user is only interested in retrieving weight and exercise from its connected partners. These options would then be stored as properties linked against the specific partners and user.

The task finder method is called with information of what user and partner that is being processed and should return a list of keywords of what tasks the synchronization should process.

First of all a query is made to the database to see if there are any items that needs to be pushed to the partner, if so, the keyword "push" is added to an empty list. After that are all properties related to the specific user and partner checked to see if it matches the partner's allowed sync types. For example, the properties that are fetched for a user connected to RunKeeper might return "poll_weight" and "poll_exercise". That would both match the allowed sync types of RunKeeper and be added to the return list.

4.9 Sync task

The sync part of the system delegates information of which tasks a user want to sync to the correct partner module's pushing/polling methods.

If a poll has returned any information, the sync task forwards that information to the correct data serialization method that parses the data into ShapeUp Club's current data model.

Since the calls will be made asynchronously a feature was implemented to handle the concurrency of the sync. The sync should only process a partner user at a time. Django has a built in cache framework [19] that suits this purpose. It is a basic low-level memcache API that can store the id of the partner user object. When the sync has completed the id can be removed from memcache, which removes the sync blockade for that user.

After a push or poll has successfully been made, a sync item is stored as described in chapter 4.4.

By executing all heavy syncing asynchronously and minimizing database calls, the system stays scalable since new machines can be added to handle the load.

4.10 Implementing a new partner

To implement a new partner in the system, the first step is to create a new partner module, preferably by duplicating the partner module template which has the necessary methods declared. The module then needs to override the abstract methods that are specific for each partner, such as authentication, poll or push requests and data serialization. These can't be written generic for all partners.
The name of the partner then needs to be added to the "installed_partners" list inside the project's settings file. The name should be identical to the name of the

partner module file. This activates the synchronization of this partner; hence the PartnerApp fetches the partner from this list.

4.11 Authenticator

The authenticator handles the authentication flow to the other partner services. It contains an interface with two methods that each partner module needs to implement.

The first method is "get_authorize_url" which will return the URI the user should be redirected to. When redirected to this URI the user must enter his/her credentials for that given partner service. Depending on which version of OAuth the partner is using there are different parameters that need to be specified. When the user is authenticated with the partner service the user will be redirected to a callback URL, that was sent with the first request to the partner service.

This is where the second method comes in. The callback URL is directed to the method "gettoken" which will validate the request and then retrieve the access token. When the request is validated, this method also creates the necessary properties for the specific partner.

If the user is not authenticated with the partner service or if the user doesn't allow the service permission to their data, the partner service will send a specific parameter indicating that something went wrong with the authentication flow.

4.11.1 Rauth

Rauth is very handy when it comes to handle the authentication flow. Each partner implementation creates a Rauth service (either compatible with OAuth1.0 or OAuth2.0) and specifies all necessary tokens, keys and callback URLs. That way you can just call a method implemented in Rauth to retrieve an OAuth1.0 callback URL, together with all the parameters sorted.

When it comes to retrieving the access token, Rauth does all the work again. Since all URLs are specified to the Rauth service, Rauth can retrieve the access token. If the partner service is returning the access token in the standard way, transforming the Rauth service to a Rauth session will retrieve the token. The Rauth session is the object that holds the token. But since some partner services have custom responses you can also get the token in a raw response.

There are other python libraries that you can use to handle OAuth authentication but Rauth is said to be the best and most updated library according to OAuth [20]. Since Rauth can do everything that is needed there's no reason to try other libraries.

5. Analysis of poll optimization

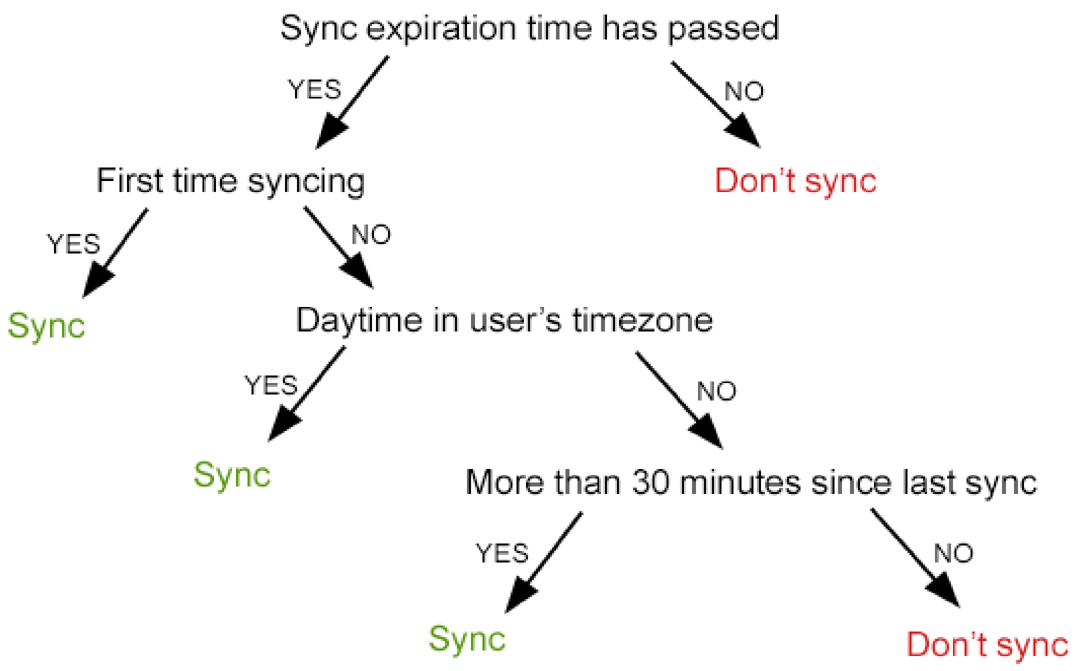
5.1 Reduce polling

To remain scalable, the system is not allowed to continuously poll information for each user and partner. Instead restrictions have to be implemented to handle when a user is allowed to synchronize.

5.1.1 Decision tree

After doing research on what kinds of algorithms are used to regulate polling frequency, the decision tree-learning algorithm, which is commonly used within data mining, would suit the purpose well if several custom criteria for deciding if a user should synchronize or not is needed [21]. An advantage with decision tree is that it's fast to compute since it's built in a tree hierarchy, which is good for a scalable system. Another advantage is that it is easy to read, modify and tweak the tree structure.

Here's an example of how a decision tree could be structured with different relevant criteria for this project:



Sync expiration time has passed
YES
NO
First time syncing
Don't sync
YES
NO
Sync
Daytime in user's timezone
YES
NO
Sync
More than 30 minutes since last sync
YES
NO
Sync
Don't sync

Figure 5.1 Example of a decision tree

It makes most sense to use quantified user data as criteria in the decision tree algorithm, since the decision it is making is user centric. But before putting the system in production there is no data to refer to. After discussing the issue with the supervisor at ShapeUp Club, it was decided to simply use the time since the last successful synchronization as a threshold for how often a user gets to sync, as described in chapter 4.7.2.
When the system has been in production for enough time, the gathered user data could be evaluated to create a relevant decision tree for the system.

5.2 Scheduled polling versus triggered polling

The implementation that was chosen was to perform synchronization for all partners and their connected users on a regular basis. An alternative would be to have triggered synchronization from user activity. For example, when a user opens the client application, synchronization would be made against ShapeUp Club and an asynchronously request trigger the partner platform to perform a sync against each of the users' partners. The problem with this would be that the sync against ShapeUp Club would most certainly finish before the sync against the different partners and any possible new data would not be shown to the user until the next time he syncs with ShapeUp Club. This will lead to the user being one step behind with the synchronization and that is not good for user experience. However, this wouldn't be the case if new server data would be pushed to the clients, then the fetched data from external services could be pushed out as it comes in.

5.3 Testing poll performance

5.3.1 Server specifications

To analyze the performance, a server has been set up. The server has 1 CPU and 1650 MB of RAM and is running on Ubuntu 12.04. When in idle, the server is using 25% of the physical memory and 2% of the CPU.

5.3.2 New Relic

When measuring the performance, a system called New Relic is used. New Relic can show the current CPU load, amount of physical memory used and also the amount of background tasks, such as the syncing tasks that will be running. Within the information about the background tasks there are also information about how many database calls that have been made and also what kind of database calls.

5.3.3 Testing the performance

The tests will be done with 1000 of ShapeUp Club's current users that are connected to RunKeeper. Three different tests will be performed;

1. In the first test all users will sync every minute, which is the worst-case scenario for the system.
2. The second test will put 2/3 of the users in group 5, which only allows the users to sync every 5 minutes. The rest will continue to sync every minute.
3. The last test will put 1/3 of the users in group 5, 1/3 in group 20 and the rest in group 60.

No data will be pushed to RunKeeper. The sync will only try to poll new data from RunKeeper and store the data in the database. The tests will be running for 30 minutes before the values are captured.

5.3.4 Evaluation of tests

The tests showed a slightly increase of CPU load in test #1, compared to the later ones as shown in figure 5.2. This is most likely due to more database calls since all users were syncing every minute. As shown in the figure, the first test didn't even manage to run all 30 tasks within the 30 minutes.

The RAM usage did not change much between the tests and is not considered a bottleneck in the system.

By using the poll group algorithm that was implemented, the system saved more than 160.000 database calls. A worker task may do several API calls to the partner, depending on their protocol. The test results show that at least 7.630 API calls was avoided with this algorithm.

Test #	CPU (%)	RAM (%)	Sync tasks (#)	Sync DB calls (#)	Worker tasks (#)	Worker DB calls (#)
1	43	25,5	20	80,120	12,600	191,369
2	38	27,8	30	34,650	7,720	147,900
3	39,6	24,9	30	22,590	4,970	87,904

5.2 Result of poll test

6. Conclusions

6.1 Evaluation of the goals

The main goal of this project has been to build a platform where ShapeUp Club's developers can plug in new external partner service modules. Adding a partner should have as minimal impact of the overall codebase as possible and be completely uncoupled against other partners.
This was achieved by writing the system logic in an generic way that allows for any possible health service partner to be implemented under the same codebase. To add a partner, one simply has to create a new module that inherits from the partner base class, that implements the sync logic. The new partner module only has to override partner specific methods such as for authentication, performing actual requests and serializing fetched data.

A user connects to a partner by an authentication flow, for now, this is always through an OAuth authentication process. When successfully authenticated, the user is given an access token. The access token is stored as a property linked to the unique persistent record of the specific user and partner. If a user later disconnects from a partner, the access token is removed.

After fetching data from a partner, the result is passed through the partner module's parsing method to extract any relevant information into a container object. The container object is a generic container of each attribute the ShapeUp Club data model is interested in.
The container is then sent to a common storage method that extracts information from the container and stores it in ShapeUp Club's database, this way a single generic storage method can be used with any partner service.

By persistently storing each item that has been pushed or fetched through the sync process information can be attached to the item. For example if pushing an item to a partner fails, a flag is set. Then when the next sync is performed, the item that hasn't been properly synced can be found and re-sent.

Only a third of ShapeUp Club's RunKeeper connected users successfully fetches new data every month. This means that over 2/3 of the users are "inactive" and shouldn't be polling as much as the active users. To regulate this, users are put into different groups depending on their activity level. The groups determines how often the users are allowed to poll for data.

After research on algorithms for regulating poll requests the "Decision tree learning" algorithm was found. It's a common algorithm in data mining that suits this projects product well if custom criteria are to be used. Unfortunately there isn't any user data available before the system goes into production, so a grouping system was implemented where the time users has been inactive decides how often they get to sync.

Every abstract logic method has been covered with Python unit tests that initiate static data needed to perform a valid test instead of using dynamic data that might change in the future and cause the test to fail.

6.2 Discussion

6.2.1 When and why is it suitable to write a partner platform?

It has been quite time demanding to write the partner platform in a generic way to support any type of health service API. If the plan were to only integrate with one or two partners, it would have been more time efficient and easier to write separate but similar solutions for synchronizing users' data against them. After two separate partners, it started to feel like it would go out of control having three implementations for partner integrations, so the recommendation would be to build a stable partner platform when dealing with more than two partners.

There are many similar health services with APIs that offers the same kind of data. The services with the most active users will obviously be the most desirable service to integrate with. To build a partner platform for integrating with all of them might seem like a waste of time. What you have to consider is that the health service market for mobile devices is evolving very fast and new products and services appear more and more often. The service with most active users may switch in a fast pace; therefore it is wise to have a stable platform to quickly integrate with the services that are currently in the spotlight.

It is becoming more and more usual for health services to integrate with each other. RunKeeper has even built the health graph to connect hundreds of services to a collective database. To stay competitive, health services has a need to integrate with as many relevant partners as possible.

7. Recommendations

This chapter will give examples of useful ways for further work on the system.

7.1 Future work

7.1.1 Callbacks

Some health services such as Withings supports registration of callback methods that are called when there are new information available. To support this kind of feature, the PartnerApp system will need to have methods to either receive notifications about which user needs to be updated, or methods to store information directly. By asynchronously receiving pushes when new information is available instead of repetitively performing poll requests, the number of redundant http requests can be eliminated completely.

7.1.2 Logging

If a user reports that she is not receiving data from a partner, or the other way around, one needs to access the database and dig into stored “SyncItems” and flags to find out where the problem resides. By having an error-log that stores information if certain pre-defined error scenarios occurs the amount of time investigating errors can be diminished and makes it easier to notice errors as well. A pre-defined error scenario could be that the user has something that needs to be synced to a partner, and after the synchronization, the item still needs to be synced.

7.1.3 Implementing decision tree

As mentioned in 5.1.1, the decision tree algorithm would be suitable to use if there is enough gathered data that can be quantified into thresholds. After having the system in production for a while, gathered data should be evaluated to see if it is suitable and sufficient enough to implement the decision tree algorithm. If not, then an investigation should be made to establish what kind of data would be more suitable and should be mined in the future.

8. References

8.1 Online references

[1] List of organizations using Python, Python Software Foundation, 2012-09-22
http://wiki.python.org/moin/OrganizationsUsingPython
2013-05-16

[2] General Python FAQ – What is Python?, Python Software Foundation, 2013-05-21
http://docs.python.org/3/faq/general.html#what-is-python
2013-05-21

[3] Virtual Python Environment builder, Python Software Foundation
https://pypi.python.org/pypi/virtualenv
2013-05-16

[4] How we use Python at Spotify, Geoff Wilson, 2013-03-20
http://labs.spotify.com/2013/03/20/how-we-use-python-at-spotify/
2013-05-16

[6] Django Models, Django Software Foundation
https://docs.djangoproject.com/en/dev/topics/db/models/
2013-05-16

[7] Message Broker, Microsoft Corporation
http://msdn.microsoft.com/en-us/library/ff648849.aspx
2013-05-16

[8] A quick message queue benchmark, Muriel Salvan, 2013-04-10
http://x-aeon.com/wp/2013/04/10/a-quick-message-queue-benchmark-activemq-rabbitmq-hornetq-qpid-apollo/
2013-05-16

[9] What is RabbitMQ?, GoPivotal, Inc
http://www.rabbitmq.com/
2013-05-16

[10] Withings API documentation, 2013-04-18
http://www.withings.com/api
2013-05-16

[11] Documentation Overview, FitnessKeeper Inc
http://developer.runkeeper.com/healthgraph/overview
2013-05-16

[12] Welcome to the Fitbit API, Eric Friedman, Pavel Risenberg, 2011-12-01
https://wiki.fitbit.com/display/API/Fitbit+API;jsessionid=7F875C83948129870
6A38C614C4C2F15
2013-05-16

[13] OAuth
http://oauth.net/
2013-05-03

[14] Introducing OAuth 2.0, Eran Hammer, 2012-05-15
http://hueniverse.com/2010/05/introducing-oauth-2-0/
2013-05-03

[15] OAuth 2.0 :: RFCs 6749 and 6750, Dick Hardt, 2012-10-12
http://dickhardt.org/2012/10/oauth-2-0/
2013-05-03

[16] Rauth, Litl
https://rauth.readthedocs.org/en/latest/
2013-05-16

[17] Scalable System Design, Ricky Ho, 2011-08-04
http://architects.dzone.com/news/scalable-system-design-0
2013-05-21

[18] Scalability Principles, Simon Brown, 2008-05-21
http://www.infoq.com/articles/scalability-principles
2013-05-21

[19] Django's cache framework, Django Software Foundation
https://docs.djangoproject.com/en/dev/topics/cache/
2013-05-16

[20] OAuth libraries
http://oauth.net/code/
2013-05-16

[21] Introduction to Data Mining, Tan, Steinbach, Kumar, 2004-04-18
http://www-
users.cs.umn.edu/~kumar/dmbook/dmslides/chap4_basic_classification.pdf,
2013-05-16

8.2 Illustration references

[4.1, 4.2, 4.3, 5.1, 5.2] Illustrations by author to visualize system features