gamification-engine Documentation

Release 0.3.0

ActiDoo GmbH

May 11, 2018

Contents

gamification-engine is a flexible open source gamification solution that allows you to easily integrate gamification features into your own products.

Contents:

CHAPTER 1

Installation

1.1 Requirements

The gamification-engine requires an installed python distribution in version 3.x. It uses several language structures which are not supported in Python 2.x. Furthermore, the only currently supported persistence layer is PostgreSQL. Also the corresponding development packages are required (for Ubuntu/Debian: libpq-dev and python3-dev).

1.2 Installation from PyPI

The gamification-engine is available as a python package. You can install it by invoking

```
$ pip install gamification-engine
$ gengine_quickstart mygengine
$ cd mygengine
```

In the latest version, there are some optional dependencies for auth pushes and testing. To use these features install it in the following way:

```
$ pip install gamification-engine[auth,pushes,testing]
```

Afterwards edit production.ini according to your needs.

1.3 Database

The only currently supported persistence layer is PostgreSQL as we make use of its timezone-related features.

To create the tables run:

```
$ initialize_gengine_db production.ini
```

1.4 Caching

For caching we make use of two different approaches:

- using dogpile.cache for caching database lookups and computations
- using memcached as a URL-based cache that can be served directly by nginx

The second approach is optional but highly recommended, it can be deactivated by setting urlcache_active = false in your ini-file.

1.5 Serving

You can use any WSGI-supporting webserver. (e.g. nginx as a reverse-proxy to uwsgi)

To quickly get started, you can run:

```
$ pserve production.ini
```

1.6 Heroku-style

There is also an Heroku-like Project (we use dokku) at gamification-engine-dokku

1.7 Commercial Support

Commercial support is available at https://www.gamification-software.com or together with app development at https://www.appnadoo.de

CHAPTER 2

Upgrading

2.1 From 0.1 to 0.2

In version 0.2 we have introduced **breaking changes** that make it impossible to do an automatic upgrade. If you are happy with 0.1, there is no need to upgrade. Furthermore, we have switched to Python 3.x as our main target environment. For performing a manual upgrade the following steps are required:

- Install a new instance of 0.2
- Recreate all settings / achievements manually using the new goal condition syntax
- Recreate users
- Copy values data

For future updates we will try to keep the goal condition syntax backwards compatible.

CHAPTER 3

Concepts

Assumption: You installed the gamification-engine and you can open the admin interface at /admin/

3.1 Users

Gamification is always about users. As the gamification-engine include location-based, time-based and social features, it needs to know some information about the user:

- lat
- lon
- country
- city
- region
- friends
- groups

3.2 Variables / Values / Events

Variables describe events that can happen in your application.

When such an event occurs, your application triggers the gamification engine to increase the value of the variable for the relevant users.

The storage of these values can be grouped by day, month or year to save storage. Note that if you want to specify time-based rules like “event X occurs Y times in the last 14 days”, you may not group the values by month or year.

In addition to integers, the application can also add additional keys to the variables to model application-specific data.

3.3 Goals

Goals define conditions that need to be fulfilled in order to get an achievement.

- goal: the value that is used for comparison
- operator: “geq” or “leq”; used for comparison
- condition: the rule in json format, see below
- **group_by_dateformat: passed as a parameter to to_char (PostgreSQL-Docs)** e.g. you can select and group by the weekday by using “ID” for ISO 8601 day of the week (1-7) which can afterwards be used in the condition
- group_by_key: group by the key of the values table
- timespan: number of days which are considered (uses utc, i.e. days*24hours)
- maxmin: “max” or “min” - select min or max value after grouping

The conditions contain a python expression that must evaluate to a valid parameter for SQLAlchemy’s where function.

Examples:

When the user has participated in the seminars 5, 7, and 9, he should get an achievement. We first need to create a variable “participate” and tell our application to increase the value of that variable with the seminar ID as key for the user by 1. The constraint that a user may not attend multiple times to one seminar is covered by the application and not discussed here. In the gamification-engine we create a Goal with the following formular:

```
{
  "term": {
    "type": "literal",
    "variable": "participate",
    "key": ["5","7","9"],
    "key_operator": "IN"
  }
}
```

Whenever a value for “participate” is set, this Goal is evaluated. It sums up all rows with the given condition and compares it to the Goal’s “goal” attribute using the given operator.

Another simple example is to count the number of invited users. After inviting 30 other users to the application, the user should get an achievement. We create a variable “invite_users” and set the condition as follows:

```
p.var=="invite_users"
{
  "term": {
    "type": "literal",
    "variable": "invite_users"
  }
}
```

Furthermore we set the Goal’s goal to 30 and the operator to “geq”.

If you want to make use of Goals with multiple levels, you probably want to increase the goal attribute with every level. Therefore, you can mathematical formulas.

Example:

For the first level, the user needs to invite 5 other users, for the second level 10 other users and so on.

```
5*level # level is set by the gamification engine
```

For further information about the rule language, we currently need to refer to the sources .

3.4 Achievements

Achievements contain a collection of rewards that are given to users who reach all assigned Goals of the Achievement. To allow multiple levels, you can set the *maxlevel* attribute.

You can specify time-based constraints by setting *valid_start* and *valid_end*, and location-based constraints by setting *lat*,*lng* and *max_distance*.

The *hidden* flag can be used to model secret achievements. The *priority* specifies a custom order in output lists.

Achievements can also be used to model leaderboards. Therefor you need to assign a single Goal whose *goal attribute* is set to None. The Achievement's *relevance* attribute specifies in which context the leaderboard should be computed. Valid values are "friends", "city" and "own".

For setting up recurring achievements, set the *evaluation* to e.g. *monthly*. The *evaluation_timezone* parameter specifies when exactly the periods begin and end.

There is a *view_permission* setting that can be used when authorization is active. It specifies whether other users can see the goal progress.

3.5 Properties

A property describes an Achievement or a Goal of our system, like the name, image, description or XP the user should get. The Values of Properties can again be python formulas. Inside the formula you can make use of the level by using *level*.

Additionally, Properties can be used as Variables. This is useful to model goals like "reach 1000xp".

3.6 Rewards

From the model perspective Rewards are similar to Properties. The main difference occurs during the evaluation of Achievements, more specifically when a user reaches a new level. While the formulas for the properties are simply evaluated for the specific level, the evaluated formulas of the rewards are compared to lower levels.

The engine thus knows for each achieved level, which reward is new and can tell the application about this. In your application this could for example trigger a badge notification.

3.7 Further new concepts

Since the latest version, some complete new optional concepts and features are added to the gamification-engine:

- Authentication
- Push Notifications
- Messages

All of these features are optional and they are not required to successfully use the engine. For the moment we refer to the source code and the description of the Rest API, a detailed documentation will follow.

CHAPTER 4

REST API

4.1 Add or update user data

- **POST to "/add_or_update_user/{userId}"**
  - **URL parameters:**
    - userId (the Id of a user in your system)
  - **POST parameters:**
    - lat (float latitude)
    - lon (float longitude)
    - country (String country)
    - city (String city)
    - region (String city)
    - friends (comma separated list of user Ids)
    - groups (comma separated list of group Ids)
    - language (name)
    - additional_public_data (JSON)
- add or updates a user with Id {userId} and Post parameters into the engines database
- if friends Ids are not registered a empty record with only the user Id will be created

4.2 Delete a user

- DELETE to "/delete_user/{userId}"

4.3 Increase Value

- **POST to "/increase_value/{variable_name}/{userId}/{key}"**
  - **URL parameters:**
    - variable_name (the name of the variable to increase or decrease)
    - userId (the Id of the user)
    - key (an optional key, describing the context of the event, can be used in rules)
  - **POST parameters:**
    - value (the increase/decrease value in Double)
- if the userId is not registered an error will be thrown
- directly evaluates all goals associated with this variable_name
- directly returns new reached achievements

4.4 Increase multiple Values at once

- **POST to "/increase_multi_values"**
  - **JSON request body:**

```
{
  "{userId}" [{]
    "{variable}" [[]
      { "key" : "{key}", "value" : "{value}"
      }
    ]
  }
}
```

- directly evaluates all goals associated with the given variables
- directly returns new reached achievements

4.5 Get Progress

- get complete achievement progress for a single user
- GET to "/progress/{userId}"
- returns the complete achievement progress of a single user

4.6 Get a single achievement Level

- GET to "/achievement/{achievement_id}/level/{level}"
- retrieves information about the rewards/properties of an achievement level

4.7 Authentication

- POST to "/auth/login"
- Parameters in JSON-Body: email, password
- **Returns a json body with a token:**

  { "token" : "foobar. . . "

  }

4.8 Register Device (for Push-Messages)

- POST to "/register_device/{user_id}"
- Parameters in JSON-Body: device_id, push_id, device_os, app_version
- **Returns a json body with an ok status, or an error:**

  { "status" : "ok"

  }

4.9 Get Messages

- GET to "/messages/{user_id}"
- Possible GET Parameters: offset
- Limit is always 100
- **Returns a json body with the messages:**

  {

  "messages" [[{] "id" : ". . . ", "text" : ". . . ", "is_read" : false, "created_at" : ". . . "

  }]

  }

4.10 Set Messages Read

- POST to "/read_messages/{user_id}"
- Parameters in JSON-Body: message_id
- Sets all messages as read which are at least as old, as the given message

- **Returns a json body with an ok status, or an error:**

  { “status” : “ok”

  }

CHAPTER 5

Modules

CHAPTER 6

Roadmap

Anyone is invited to work on new features, even if they are not listed here. Features which might influence the overall performance or cause greater changes should be discussed in a feature request before.

At ActiDoo.com we implement new functions as we need them and push them as soon as they are somewhat stable.

6.1 Todo

- Review and improve tests
- Improve Caching

6.2 Future Features

- possibility to store events (values table) in noSQL systems
- implement callback for time-aware achievements
- nicer admin UI
- statistics
- maybe a possiblity to plugin authentication/authorization to allow users to directly push events to the engine - this still needs to be discussed from an architectural point of view - this would also introduce the need for security constraints to detect cheaters

CHAPTER 7

Indices and tables

- genindex
- modindex
- search