Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

A RESTful API for the AMIV

Group Project

Hermann Blum, Conrad Burchert, Alexander Dietmüller
blumh@ethz.ch, bconrad@ethz.ch, adietmue@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Barbara Keller
Jochen Seidel
Prof. Dr. Roger Wattenhofer

March 17, 2015

Abstract

As a group project we desgined and implemented an API for our student's association. This Report describes problems with the old infrastructure and the ideas behind our new design.
Attached are the User- and Developer Guide which were part of our work. They describe the API and implementation in a more detailed and technical way.

Contents

I Group Project Report **4**

1 Introduction **4**

2 Development in the past and current Situation **4**

3 Concept and Goal **4**

4 Solution **5**

5 Outlook **6**

II Developer Guide **8**

6 General **8**
6.1 Used Frameworks 8
6.2 Development status 8
6.3 Installation 9
6.4 Configuration 9
6.5 Running the tests 10
6.6 Debugging server 10

7 Architecture **10**

8 Security **11**
8.1 Authentification 12
8.2 Authorization 12

9 Roles **13**

10 Owner checks **13**
10.1 API Keys . . . 14

11 Localization **14**
11.1 Note: Testing . . . 15
11.2 Note: Automatization . . . 15

12 Files **16**

13 Validation **16**

14 Cron **16**

III User Guide **17**

15 General **17**
15.1 About this document . . . 17
15.2 Portability . . . 17
15.3 Encryption . . . 17
15.4 Date format, string format . . . 17
15.5 About REST . . . 17
15.6 Response format . . . 18
15.7 HATEOAS . . . 19
15.8 Example: First Request . . . 19

16 Authentification **20**
16.1 Example: Login . . . 21
16.2 Example: Retrieving user . . . 21
16.3 API keys . . . 23
16.4 Unregistered users . . . 23
16.5 Public Events . . . 23
16.6 Email Forwards . . . 24

17 GET queries **24**
17.1 where clauses . . . 24
17.2 Projections . . . 25
17.3 Embedding . . . 25
17.4 Sorting . . . 25
17.5 Pagination . . . 25

18 PUT, PATCH, DELETE queries **26**
18.1 If-Match . . . 26

19 Example: Use PATCH to change a password **26**

20 Localization: Content in different languages **26**
20.1 Example: Create an event with the requests library 26

21 Working with files **29**
21.1 Files in Events, Joboffers, etc. 29
21.2 Working with study documents 30

22 Common Problems **30**
22.1 PATCH, PUT or DELETE returns 403 30
22.2 How can I send boolean etc to the server using python requests? 30

Part I
Group Project Report

1 Introduction

We are members of the student organisation of mechanical and electrical engineers at ETH Zürich, called AMIV. Working for AMIV we realized that the current IT-infrastructure is in great need of improvement.
As a group project in the third bachelor year, we redesigned the general architecture of the system and defined and implemented an API interface as the central element of the new design.

2 Development in the past and current Situation

Over the years the IT-infrastructure of AMIV has grown with the rest of the organisation to fit many different needs:: e.g. news articles, event organization and registration, member management and lots more.
Along the way many tools have been written and integrated into the infrastructue as they were needed, since the active members of the organisation came and went, as they did not have that much time besides their studies or their time at ETH came to an end.

All of this lead to different problems which we are now facing:
Our current infrastructure is built around the Content Management System (CMS) Drupal which is extended by many self-written modules. The content is fragmented and our homepage the opposite of user friendly. The single modules in the CMS can not communicate which each otherThis leads to a lot of additional work, as one needs to put the same data into different modules to create e.g. an event that requires a website article, registration and a calendar entry, which have to be created seperately.
Moreover a single bug which may lead to a blackout in the whole infrastrucute could be in any of our (sadly) insufficiently documented modules.

Because of this maintaining the website requires both extensive knowledge of the CMS and an overview of all the different modules.

3 Concept and Goal

- Division of the infrastructure:
  - one database server: Here we store information about members, events, app-logins, permissions for the homepage, registrations

for the pay-system, signups for events
  - Homepage on basis of a Content-Management-System (e.g. Wordpress): Here we provide static contents and can show data from the database-server (events, signups, ...)
  - Applications like a beer-machine, where members can get beer with their student-card, or our mobile applications will talk to the database-server to get the data they need

- As a first step we defined interfaces of the components. This defines also the function, each component has.Our questions were:
  - Which function will this component provide?
  - Which data-elements will be accessed by whom?
  - In which format is the data stored and provided?
  - Who has right for which kind of access?

- From the client point of view, there are additional questions:
  - How can one verify their identity and right to access e.g. member data?
  - At which point authentication is neccesary?

- Implementation of the interface: We implemented the interface and the backend functions like relations or authorization and wrote a lot of tests to be sure that the interface works the way we defined.
  We did not already integrate it with the existing database or did any kind of migration.

- We documented the interface itself, the way to access it (including code examples in python). We also wrote documentation for admins and developers.
  Documentation is elemantary to conserve our knowledge and ideas behind the project for future students using this api, also in times we already left ETH.

4 Solution

On top of the function the frameworks provided and we only had to define our specific needs, we actively worked on the frameworks (first pull request already accepted) and implemented further functions:

Data Validation The Cerberus framework comes along with a basic data validation, e.g. for strings or numbers and also relationships. However, we wanted to make sure that the data a client POSTs to the api is semantically correct. So we check for example that an end-time of an event comes after the start-time.

Token based authentication Every login session is identified with a token. On login with his username and password a user receives a random generated token, which he can use to authentificate himself in subsequent requests. To achieve this we wrote a custom subclass of Eve's authentification system, which uses our SQLAlchemy data model to store currently active sessions and handle login requests.

Role based authorization Thinking about a simple model to define permissions for our api, we came along with a role based model for every big resource-category. A user can be assigned to a role that allows him to manage events and their signups, or allows him to manage email-lists.

Thinking about a simple model to define permissions for our api, we came along with a role based model for every big resource-category. A user can be assigned to a role that allows him to manage events and their signups, or allows him to manage email-lists.

We also make sure that a user can only see and manipulate his/her own eventsignups or userdata by defining one or more owners for every item in the database.

Different Languages supported for all text fields For master students in our organisation we wanted to be able to provide english translations as well. After researching "best practise" cases for this issue we implemented an additional endpoint that accepts translated content in (basically) unlimited languages.

Anonymous user support with email-confirmation For different special events like info days for high school students we need people to use our signup tool without having any account. In this case they use an email-address as identity and we authenticate request by a token the API send to the given email-address.

Application Tokens Applications like a website which displays events can authenticate themselves with application-tokens.

File management Eve uses MongoDB as default filestorage. We wanted to save files in our servers filesystem and were able to implement an extended MediaStorage Class that can be used by the framework for this exact purpose.

5 Outlook

With our group project we build the basis for a new IT system of the AMIV. The definition and documentation of the API allows us to motivate people

who want to program small applications. It is no longer necessary for everybody to learn Drupal. They just need to understand the API and can write applications in a language they prefer.
A new webpage CMS will now be reduced to providing static content or data accessed via the API. This makes content administration and design for active members simpler.
By splitting the IT system up into small pieces we expect to get a robust system that will serve the AMIV for many years.
Our plan is to migrate the whole infrastructure within one year.

Part II
Developer Guide

6 General

6.1 Used Frameworks

AMIV API uses the python-eve[1] Framework which is a collection of libraries around Flask[2] and SQLAlchemy[3]. The best source for information during development is the EVE Source Code at Eve Github Repository SQL Alchemy Branch[4].

The main links for research about the used technologies are:

- Flask[5]
- SQL Alchemy[6]
- Flask-SQL Alchemy[7]
- Werkzeug[8]
- Eve[9]

6.2 Development status

Eve is still in early development and changing a lot. That means it might be possible that we can improve our codebase as more features move into Eve's core. We are currently using a patched version of eve-sqlalchemy and eve-docs, which are forked on github here:

- eve-sqlalchemy fork by Leonidaz0r[10]
- eve-docs fork by hermannsblum[11]

[1] http://python-eve.org/
[2] http://flask.pocoo.org/
[3] http://www.sqlalchemy.org/
[4] https://github.com/nicolaiarocci/eve/tree/sqlalchemy
[5] http://flask.pocoo.org/docs/0.10/api/
[6] http://docs.sqlalchemy.org/en/rel_0_9/
[7] https://pythonhosted.org/Flask-SQLAlchemy/
[8] http://werkzeug.pocoo.org/
[9] http://python-eve.org/
[10] https://github.com/Leonidaz0r/eve-sqlalchemy
[11] https://github.com/hermannsblum/eve-docs

6.3 Installation

To setup a development environment of the API we recommend using a virtual environment with the pip python package manager. Furthermore you need git.

The following command works on Archlinux based systems, other distributions should provide a similar package:

```
sudo pacman -S python2-pip git
```

After installing pip create a working environment. First create a folder:

```
mkdir amivapi
cd amivapi
```

Now create a virtualenv which will have the python package inside and activate it:

```
virtualenv venv
. venv/bin/activate
```

Now get the source:

```
git clone https://github.com/amiv-eth/amivapi.git
cd amivapi
```

Install requirements:

```
pip install --allow-external python-mysql-connector -r
    requirements.txt
```

6.4 Configuration

Create a configuration:

```
python2 manage.py create_config
```

The tests will create their own database. If you configure a MySQL Server you will be asked whether the tests should also be run there. If you don't activate that they will create temporary databases on the fly in temporary files. Note that even if they run on a MySQL server they will create their own database, so you need to have the permissions for CREATE DATABASE.

6.5 Running the tests

To run the tests you need to install tox:

```
pip install tox
```

Create a config(see above). To run all tests enter

```
tox
```

To test just one environment use -e with py27, py34, pypi or flake8

```
tox -e py27
```

To run only some tests specify them in the following way(substitute your test class):

```
tox -- amivapi.tests.forwards
```

6.6 Debugging server

To play around with the API start a debug server:

```
python2 run.py
```

When the debug server is running it will be available at http://localhost:5000 and show all messages printed using the logger functions, print functions or exceptions thrown.

7 Architecture

The main-directory lists following files:

- authentification.py: Everything about who somebody is. Tokens are mapped to sessions and logins are handled. Also author fields are set.
- authorization.py: Everything about what somebody can do. Permissions are implemented here.
- bootstrap.py: The Eve-App gets created here. All blueprints and event-hooks are registered in the bootstrap.

- confirm.py: Blueprint and event-hooks regarding the confirmation of unregistered users.
- cron.py: Jobs run on a regular basis (sending mail about expiring permissions, cleanup)
- documentation.py: Loads additional documentation for the blueptrints.
- forwards.py: Hooks to implement the email-functionality of forwards and assignments to forwards.
- localization.py: Localization of content-fields.
- media.py: File Storage. Handles uploaded files and serves them to the user.
- models.py: The Data-Model. As a basis of the API, in the Data-Model the different Data-Classes and their relations get defined.
- schemas.py: Creates the basic validation-schema out of the data-model and applies custom changes.
- settings.py: Constants which should not be changed by the admin, but can be changed by some developer
- utils.py: General helping functions.
- validation.py: Every validation that extends the basic Cerberus-schema-definition and Hooks for special semantic checks, e.g. whether an end-time comes after a start-time.

For understanding the structure of the api, the data-model in models.py is the Point to start.

8 Security

Checking whether a request should be allowed consists of two steps, authentification and authorization. Authentification is the process of determining the user which is performing the action. Authorization is the process of determining which actions should be allowed for the authentificated user.

Authentification will provide the ID of the authentificated user in g.logged_in_user

Authorization will provide whether the user has an admin role in g.resource_admin

Requests which are handled by eve will automatically perform authentification and authorization. If you implement a custom endpoint you have to call them yourself. However authorization really depends on what is about to happen, so you might have to do it yourself. To get an idea of what to do

look at the authorization hooks(pre_xxx_permission_filter()). You can quite certainly reuse that code somehow.

Perform authentification(will abort the request for anonymous users):

```
if app.auth and not app.auth.authorized([], <resource>, request.
    method):
    return app.auth.authenticate()
```

Replace with the respective resource name.

8.1 Authentification

File: authentification.py

The process of authentification is straight forward. A user is identified by his username and his password. He can sent those to the /sessions resource and obtain a token which can prove authenticity of subsequent requests. This login process is done by the process_login function. Sessions do not time out, but can be deleted.

When a user sends a request with a token eve will create a TokenAuth object and call its check_auth function. That function will check the token against the database and set the global variable g.logged_in_user(g is the Flask g object) to the ID of the owner of the token.

8.2 Authorization

File: authorization.py

A request might be authorized based on different things. These are defined by the following properties of the model:

```
__public_methods__ = [<methods>]
__registered_methods__ = [<methods>]
__owner_methods__ = [<methods>]
__owner__ = [<fields>]
```

The **xx_methods** properties define methods which can be accessed. Also a list of fields can be set, which make somebody an owner of that object if he has the user ID corresponding to the fields content. For example a ForwardUser object has the list

```
__owner__ = ['user_id', 'forward.owner_id']
```

This defines the user referenced by the field user_id as well as the user referenced by owner_id of the corresponding Forward as the owner of this ForwardUser object.

I recommend to look over the common_authorization() function. The rules created by it are the following, in that order:

1. If the user has ID 0 allow the request.
2. If the user has a role which allows admin access to the endpoint allow the request
3. If the endpoint is public allow the request.
4. If the endpoint is open to registered users allow the request.
5. If the endpoint is open to object owners, perform owner checks(see below)
6. Abort(403)

The function will also set the variable g.resource_admin depending on whether the user has an admin role(or is root).

One thing to note is that users which are not logged in are already aborted by eve when authentification fails for resources which are not public, therefore this is not checked anymore in step 4.

9 Roles

Roles can be defined in permission_matrix.py. A role can give a user the right to perform any action on an endpoint. If permission is granted based on a role no further filters are applied, hence it is refered to as admin access and g.resource_admin is set.

10 Owner checks

If the authorization check arrives at step 5 and the requested resource has owner fields, then those will be used to determine the results. This is the case for example when a registered user without an admin role performs a GET request on the ForwardUser resource. He can perform that query, however he is supposed to only see entries which forward to him or where he is the listowner.

This is solved by two functions. When extracting data we need to create additional lookup filters. Those are inserted by the apply_lookup_filters() function which is called by the hooks below it. When inserting new data or changing data it gets more complicated. First we need to make sure that the object which is manipulated belongs to the user, that is achieved using the previously described function. In addition we need to make sure that the object afterwards still belongs to him. We do not want people

moving EventSignups or ForwardUsers to other users. All this is done in the will_be_owner() function which is used by the hooks as needed. However to achieve this the function needs to figure out what would happen if the request was executed. This is currently done by the resolve_future_field() function, which tries to resolve relationships using SQLAlchemy meta attributes for the data which is not yet inserted.
If this checks out ok, the hooks return, if not the request is aborted.

To check ownership inside your own function for an existing object, you can use get_owner(resource, _id) from utils. It will return a list of user-ids who are owners of the item. A common owner check looks like this:

```
1 if g.logged_in_user is in utils.get_owner(<resource>, <_id>):
```

10.1 API Keys

Instead of a token an API key can be sent. These are generated by manage.py and are stored in the config file. If an API key is sent, the user ID will be –1(the anonymous user) and all actions will be authorized based on the settings for that key in the config.

For implementation see common_authorization() and TokenAuth.check_auth()

11 Localization

The api includes support for several languages in four fields which contain language_dependant content, these are:

- joboffers.title
- joboffers.description
- events.title
- events.desription

The general idea is: for every instance of each field we want an unique ID that can be used to provide translated content.

This is solved using two new resources:

1. translationmappings
   This resource is internal (see schemas.py), which means that it can only be accessed by eve internally.
   To ensure that this works with eve and our modifications (like _author fields) we are not using SQLAlchemy relationship configurations to create this field.

Instead the hook "insert_localization_ids" is called whenever events and joboffers are created. It posts internally to languagemappings to create the ids which are then added to the data of post.
The relationship in models.py ensures that all entries in the mapping table are deleted with the event/joboffer

2. translations
   This resource contains the actual translated data and works pretty straightforward:
   Given a localization id entries can be added

How is the content added when fetching the resource?

The insert_localized_fields hook check the language relevant fields and has to query the database to retrieve the language content for the given localization id.

Then it uses flasks request.accept_languages.best_match() function to get the best fitting option. (it compares the Accept Language header to the given languages)

When none is matching it tries to return content in default language as specified in settings.py (Idea behind this: There will nearly always be german content). If this it not available, it uses an empty string)

The field (title or description) is then added to the response

11.1 Note: Testing

Both events and joboffers have the exact language fields, but job_offers have less other required fields.
Therefore testing is done with job_offers - if there are any problems with language fields in events, ensure that the tests work AND that all language fields in events are configured EXACLY like in joboffers

11.2 Note: Automatization

Since there are only four language fields (with title and description for both events and joboffers, which is convenient) all hooks and schema updates are done manually. Should a update of the api be intended which includes several more language fields automating this should be considered.

For every language field the following is necessary:

- Internal post to languagemappings to create the id (locatization.py, hook)
- Retrieving the content when fetching the resource (locatization.py, hook)

- Adding a id (foreignkey) and relationship to translationmappings (models.py)
- Removing id from the schema to prohibit manually setting it (schemas.py)

12 Files

For files we wrote our own MediaStorage class as used by Eve by extending the template[12] . The files need a folder which is created in the process of "create_config".

Maybe in future releases of Eve there will be an official implementation of file system storage. Maybe it would be useful to use this instead of our implementation instead in this case.

How Eve uses the MediaStorage Class can be found here[13]

To serve the information specified in EXTENDED_MEDIA_INFO the file "media.py"
contains the class "ExtFile" which contains the file as well as the additional information Eve needs.

As EXTENDED_MEDIA_INFO we use file name, size and a URL to the file. The URL can be accessed over a custom endpoint specified in "file_endpoint.py", using flask methods.

13 Validation

Luckily the cerberus validator is easily extensible, so we could implement many custom rules. Those are found in validator.py and are not very complex.

More information on cerberus and its merits can be found in the Cerberus Documentation[14]

14 Cron

There are some tasks which are done on a regular basis. This includes removing expired permissions and unused sessions. Users who's permissions expire should be warned prior to this by mail. This is all done by a cronjob. The cronjob runs cron.py.

[12] https://github.com/nicolaiarocci/eve/blob/develop/eve/io/media.py
[13] http://python-eve.org/features.html#file-storage
[14] https://cerberus.readthedocs.org/en/latest/

Part III
User Guide

15 General

[TOC]

15.1 About this document

This document should help somebody who wants to develop a client for the AMIV API. It focuses on manipulating data via the public interface. For in depth information see Developer Guide, for reference visit the [API Reference](https://<base_url>/docs).

15.2 Portability

The API is designed and all clients should be designed to be useable outside of AMIV. Although we will use api.amiv.ethz.ch as the base URL in this document this is not necessary and a client should provide a config entry for that.

15.3 Encryption

The API is only accessible via SSL and will never be made public via an unencrypted channel, as should all your apps.

15.4 Date format, string format

Date and time is always UTC in ISO format with time and without microseconds. Any different time format will result in 400 Bad Request.

```
1 %Y-%m-%DT%H:%M:%SZ
```

All strings are UTF–8.

15.5 About REST

AMIV API is a REST API[15]. REST is a stateless protocoll modelled after HTTP. The API consists of resources, which have objects. For example the resource /users provides access to the member database. Every resource has the methods GET, POST, PUT, PATCH, DELETE. These are the known regular HTTP methods, GET and POST being the most well known. The

[15] https://de.wikipedia.org/wiki/Representational_State_Transfer

API is based on the python-eve[16] framework, so you can refer to eve for detailed information as well.

There are many clients available to use REST and there are libraries for all kind of programming languages. Many HTTP libraries will also be able to communicate with a REST API.

The methods meanings:

Resource methods(use i.e. on /users) * GET - Retriving data, query information may be passed in the query string * POST - Creating a new entry, the new entry must be provided in the data section

Item methods(use i.e. on /users/4) * PATCH - Changing an entry * DELETE - Removing an entry * PUT - Replacing an entry, this is like DELETE immediately followed by POST. PUT ensures no one else can perform a transaction in between those two queries

15.6 Response format

The status code returned by the API are the standard HTTP status codes[17]. Codes starting with 2 mean the operation was successfull, starting with 3 are authentification related, 4 are client errors, 5 are server errors, 6 are global errors. Most important codes are:

- 200 - OK (Generic success)
- 201 - Created (successful POST)
- 204 - Deleted (successful DELETE)
- 400 - Bad request (This means your request has created an exception in the server and the previous state was restored, if you are sure it is not your fault file a bug report)
- 401 - Please log in
- 403 - Logged in but not allowed (This is not for you)
- 404 - No content (This can also mean you could retrive something here, but no object is visible to you because your account is that of a peasant)
- 412 - The etag or confirmation-token you provieded is wrong
- 422 - Semantic error (Your data does not make sense, e.g. dates in the past which should not be)
- 500 - Generic server error

[16] http://python-eve.org/index.html
[17] https://de.wikipedia.org/wiki/HTTP-Statuscode

- 501 - Not implemented (Should work after alpha)

All responses by the API are in the json format[18] by default. Using the Accept header output can be switched to XML, but we encourage to use json as near to no testing has been done for XML output. If you want XML support consider reading the Developer Guide and providing unit tests for XML output.

15.7 HATEOAS

The API is supposed to be human readable, meaning a human can read the responses and only knowing REST standard can perform any action available. That means it is possible to get any information about the structure of the data via the API. Starting at the root node /URL links will be provided to any object.

Check wikipedia[19] for more info.

15.8 Example: First Request

The examples will provide code in python using the requests[20] library. If you are developing a client in the python language requests might be a possible choice to query the API.

Request:

```
GET /
```

Code:

```
response = requests.get("https://api.amiv.ethz.ch/")
```

Response:

```
status: 200

{
 "_links": {
  "child": [
   {
    "href": "/files",
    "title": "files"
   },
   {
    "href": "/studydocuments",
```

[18] https://de.wikipedia.org/wiki/JavaScript_Object_Notation
[19] https://en.wikipedia.org/wiki/HATEOAS
[20] http://docs.python-requests.org/en/latest/

```
      "title": "studydocuments"
    },
    {
      "href": "/forwardusers",
      "title": "forwardusers"
    },
    {
      "href": "/forwards",
      "title": "forwards"
    },
    {
      "href": "/sessions",
      "title": "sessions"
    },
    {
      "href": "/joboffers",
      "title": "joboffers"
    },
    {
      "href": "/eventsignups",
      "title": "eventsignups"
    },
    {
      "href": "/forwardaddresses",
      "title": "forwardaddresses"
    },
    {
      "href": "/users",
      "title": "users"
    },
    {
      "href": "/events",
      "title": "events"
    },
    {
      "href": "/permissions",
      "title": "permissions"
    }
  ]
 }
}
```


16 Authentification

Most access to the API is restricted. To perform queries you have to log in and acquire a login token. The login token is a unique string identifying you during a session. Sessions are a part of the data model as any other object and can be created in the normal way. Just send a POST request to the /sessions resource:

16.1 Example: Login

Request:

```
POST /sessions?username=myuser&password=mypassword
```

Code:

```
response = requests.post("https://api.amiv.ethz.ch/sessions",
    data={"username": "myuser", "password": "mypassword"})
```

Response:

```
status = 201

{
    u'_author': -1,
    u'_created': u'2014-12-20T11:50:06Z',
    u'_etag': u'088401622fc10cbf0d549e9282072c37829a1b81',
    u'_id': 4,
    u'_links': {u'self': {u'href': u'/sessions/4', u'title': u'
    Session'}},
    u'_status': u'OK',
    u'_updated': u'2014-12-20T11:50:06Z',
    u'id': 4,
    u'token': u'eyJzadasdaswfgmjuhdjhfgjs=',
    u'user_id': 4
}
```

We will look at the details of this response later. First we only notice the token field and use that token to issue an authenticated query to find out something about our user account. A token can be passed as the HTTP Basic Auth username with an empty password. The python requests library provides this functionality as does command line curl. If you can not pass such a field you can create the Authorization header which would be generated by that parameter yourself. For that you need to base64 encode the token followed by a colon. We will see examples for both methods.

16.2 Example: Retrieving user

It is possible to retrive a user using its username. Normally we would use an ID to retrive an item, but in this case it is easier this way.

Request:

```
POST /sessions "username=myuser&password=mypassword"
GET /users/myuser (+Authorization header)
```

Code with good REST library:

```
login = requests.post("http://api.amiv.ethz.ch/sessions", data={
    "username": "myuser", "password": "mypassword"})
token = login.json()['token']
response = requests.get("https://api.amiv.ethz.ch/users/myuser",
     auth=requests.auth.HTTPBasicAuth(token, ""))
```

Code with bad REST library:

```
login = requests.post("/sessions", data={"username": "myuser", "
    password": "mypassword"})
token = login.json()['token']
auth_header = b64encode(token + ":")
response = requests.get("/users/myuser", headers={"Authorization
    ": auth_header})
```

Response:

```
{
    '_author': 0,
    '_created': '2014-12-18T23:29:07Z',
    '_etag': '29023402348290348203498203948 2034',
    '_links': {
        'parent': {
            'href': '/', 'title': 'home'
        },
        'self': {
            'href': '/users',
            'title': 'users'
        }
    },
    '_updated': '2014-12-18T23:29:07Z',
    'birthday': None,
    'department': None,
    'email': 'kimjong@whitehouse.gov',
    'firstname': 'Edward',
    'gender': 'male',
    'groups': None,
    'id': 4,
    'lastname': 'Nigma',
    'ldapAddress': None,
    'legi': None,
    'membership': 'none',
    'nethz': None,
    'phone': None,
    'rfid': None,
    'username': 'myuser'
}
```

16.3 API keys

If access is not done by a user but rather by a service(cron, vending machine, info screen), user based authorization does not work. Instead an API key can be used. The API administrator can generate keys using the manage.py script and configure which endpoints can be accessed. Endpoint access via API key will give admin priviledges. The API key can be sent in the same way as a token. You can think of it as a permanent admin session for specific endpoints.

16.4 Unregistered users

Next to GET operations on public data, AMIV API currently allows unregistered users in exactly two cases: Signing up for a public event or managing email-subscribtions for public email lists. In Both cases, 'is_public' of the event or forward must be True.

Basically, an unregistered user can perform any GET, POST, PATCH or DELETE action on the supported resource within the usual rights. However, as the HTTP request comes without login, you need to confirm yourself and your email-address with a special token. After the creation of a new item with POST, the User will get an email with the Token. Your Admin might provide links in this mail to a user-friendly tool. However, here is the Workflow that always works: Just POST the token send to you to '/confirmations' in the following way:

```
POST /confirmations?token=dagrfvcihk34t8xa2dasfd
```

After this, the server knows that the given email-address is valid. Every further Action kann be performed as usually, but with a special Header:

```
{
    'Token': dagrfvcihk34t8xa2dasfd
}
```

The API will return 403 FORBIDDEN if you did forgot to provide a token and will return 412 PRECONDITION FAILED if the provided token is not valid for the requested item.

16.5 Public Events

To subscribe to a public event with an email-address you simply post to "/eventsignups":

Data:

```
{
    'event_id': 17,
    'user_id': -1,
    'email': "mymail@myprovider.ch",
}
```

You will receive a 202 Acepted. This means that the signup is not valid yet, but the server has received valid data and the user can confirm the signup by clicking on a link in an email. The User-ID '–1' stands for the anonymous user.

16.6 Email Forwards

For email-lists, we know 3 resources: '/forwards', '/forwardusers', '/forwardaddresses'. '/forwards' is used to manage lists. '/forwardusers' is used to manage entries which forward to a registered user. '/forwardaddresses' is used for anonymous entries. To create a new subscription or change an existing one for an unregistered user, you need to use '/forwardaddresses'. The procedure of confirmation is exactly the same as for events.

17 GET queries

GET queries can be customized in many ways. There is the possibility for where, projection, embedding and pagination clauses.

17.1 where clauses

Using a where clause one can specify details about the object looked for. Queries can be stated in the python syntax(as if you would write an if clause). This is some kind of experimental, if any issues occur please contact api@amiv.ethz.ch or write a report in the issue tracker on github.

An example (url-encoded) is:

```
GET /events?where=title=="Testevent"+and+spots>5
```

A more complex query would be

```
GET /events?where=(title=="Testevent"+and+spots>5)+or+title=="
    Testevent2"
```

Embedding works only for equality comparison and no recursion at the moment(to improve, commit to the eve-sqlalchemy project), for example:

```
GET /events?where=signups.user_id==5
```

This would return all events which the user with the id 5 is signed up for.

17.2 Projections

Using the projection parameter it is possible to decide which fields should be returned. For example:

```
GET /events?projection={"location":0,"signups":1}
```

This will turn of the location field, but return a list of signups. The behaviour of data relations when their projection is enabled can be configured using embedding.

17.3 Embedding

Turning embedding on and off will determine how relations are returned by the API. With embedding turned on the whole object will be returned, with embedding turned off only the ID will be returned.

```
GET /users?projection={"permissions":1}&embedded={"permissions"
    :1}
```

This will return all the permission objects embedded in the response

17.4 Sorting

Results can be sorted using the *sort* query parameter. Prepending the name with a - will sort in descending order.

```
GET /events?sort=-start_time
```

This will return the events sorted by descending start_time.

17.5 Pagination

The number of returned results for queries to resource endpoints can be controlled using the max_results and the page parameter.

```
GET /events?max_results=10&page=3
```

This will return the third page of 10 items.

18 PUT, PATCH, DELETE queries

18.1 If-Match

To manipulate an existing object you have to supply the If-Match header to prevent race conditions. When you use GET on an element you will be provided with an _etag field. The etag is a string which changes whenever the object is manipulated somehow. When issuing a PUT, PATCH or DELETE query you must supply the etag in the If-Match header to ensure that no one else changed the object in between.

If no etag is provided, you will recieve 403 FORBIDDEN. If the etag is wrong, the api returns 412 PRECONDITION FAILED.

19 Example: Use PATCH to change a password

```
GET /users/myuser (+Authorization header)
PATCH /users/myuser data: "password=newpw" headers: "If-Match:
    a23...12b"
```

Code:

```
me = requests.get("/users/myuser", auth=myauth)
etag = me.json()['_etag']
result = requests.patch("/users/myuser", data={"password":"newpw
    "}, headers={"If-Match":etag})
```

The response will be the changed user object.

20 Localization: Content in different languages

The api supports descriptions and titles for events and job offers in different languages. If you post to one of those ressources, the response will contain a title_id and description_id. Those are the unique identifiers. To add content in various languages you can now use this id to post to the /translations resource

20.1 Example: Create an event with the requests library

Code:

```
import json  # To properly encode event data

"""Usual login"""
auth = {'username': user, 'password': pw}
r = requests.post('http://api.amiv.ethz.ch/sessions', data=auth)
```

```
token = r.json().get('token')
session = requests.Session()
session.auth = (token, '')

"""Some data without language relevant content"""
data = {'time_start': '2045-01-12T12:00:00Z',
        'time_end': '2045-01-12T13:00:00Z',
        'time_register_start': '2045-01-11T12:00:00Z',
        'time_register_end': '2045-01-11T13:00:00Z',
        'location': 'AMIV Aufenthaltsraum',
        'spots': 20,
        'is_public': True}

payload = json.dumps(data)

self.session.headers['Content-Type'] = 'application/json'
response = self.session.post('http://api.amiv.ethz.ch/events',
                             data=payload).json()
del(self.session.headers['Content-Type']) # Header not needed
    anymore
```

Response:

```
{u'_author': 0,
 u'_created': u'2015-03-05T14:12:19Z',
 u'_etag': u'8a20c7c3e035eb5a03906ce8f0f7717a4300e9de',
 u'_id': 1,
 u'_links': {u'self': {u'href': u'/events/1', u'title': u'Event
    '}},
 u'_status': u'OK',
 u'_updated': u'2015-03-05T14:12:19Z',
 u'description_id': 2,
 u'id': 1,
 u'is_public': True,
 u'location': u'AMIV Aufenthaltsraum',
 u'spots': 20,
 u'time_end': u'2045-01-12T13:00:00Z',
 u'time_register_end': u'2045-01-11T13:00:00Z',
 u'time_register_start': u'2045-01-11T12:00:00Z',
 u'time_start': u'2045-01-12T12:00:00Z',
 u'title_id': 1}
```

Now extract ids to post translations
Code:

```
"""Now add some titles"""
self.session.post('http://api.amiv.ethz.ch/translations',
                  data={'localization_id': r['title_id'],
                        'language': 'de',
                        'content': 'Irgendein Event'})
self.session.post('http://api.amiv.ethz.ch/translations',
```

```
                    data={'localization_id': r['title_id'],
                          'language': 'en',
                          'content': 'A random Event'})

"""And description"""
self.session.post('http://api.amiv.ethz.ch/translations',
                    data={'localization_id': r['description_id'],
                          'language': 'de',
                          'content': 'Hier passiert was. Komm
    vorbei!'})
self.session.post('http://api.amiv.ethz.ch/translations',
                    data={'localization_id': r['description_id'],
                          'language': 'en',
                          'content': 'Something is happening. Join
     us!'})
```

If we now specify the 'Accept-Language' Header, we get the correct content!

Code:

```
self.session.headers['Accept-Language'] = 'en'

self.session.get('http://api.amiv.ethz.ch/events/%i' % response
    ['id']).json()
```

Response:

```
{u'_author': 0,
 u'_created': u'2015-03-05T14:12:19Z',
 u'_etag': u'8a20c7c3e035eb5a03906ce8f0f7717a4300e9de',
 u'_links': {u'collection': {u'href': u'/events', u'title': u'
    events'},
             u'parent': {u'href': u'/', u'title': u'home'},
             u'self': {u'href': u'/events/1', u'title': u'Event
    '}},
 u'_updated': u'2015-03-05T14:12:19Z',
 u'additional_fields': None,
 u'description': u'Something is happening. Join us!',
 u'description_id': 2,
 u'id': 1,
 u'img_1920_1080': None,
 u'img_thumbnail': None,
 u'img_web': None,
 u'is_public': True,
 u'location': u'AMIV Aufenthaltsraum',
 u'price': None,
 u'signups': [],
 u'spots': 20,
 u'time_end': u'2045-01-12T13:00:00Z',
 u'time_register_end': u'2045-01-11T13:00:00Z',
 u'time_register_start': u'2045-01-11T12:00:00Z',
```

```
23 u'time_start': u'2045-01-12T12:00:00Z',
   u'title': u'A random Event',
25 u'title_id': 1}
```

Yay! The title and description are added in english as requested.

21 Working with files

Working with files is not much different from other resources. Most resources contain the file, only study documents, which will be explained below.

21.1 Files in Events, Joboffers, etc.

Files can be uploaded using the "multipart/form-data" type. This is supported by most REST clients. Example using python library "requests" and a job offer: (More info on requests here: http://docs.python-requests.org/en/latest/)

Code:

```
1 """Usual login"""
  auth = {'username': user, 'password': pw}
3 r = requests.post('http://api.amiv.ethz.ch/sessions', data=auth)
  token = r.json().get('token')
5 session = requests.Session()
  session.auth = (token, '')
7
  """Now uploading the file"""
9 with open('somefile.pdf', 'rb') as file:
      data = {'title': 'Some Offer'}
11    files = {'pdf': file}
      session.post('http://api.amiv.ethz.ch/joboffers',
13                 data=data, files=files)
```

Response:

```
1 {'_author': 0,
   '_created': '2015-02-19T14:46:14Z',
3  '_etag': '9cd7fdf37507d2001f5902330ff38db1236bdb84',
   '_id': 1,
5  '_links': {'self': {'href': '/joboffers/1', 'title': 'Joboffer'
     }},
   '_status': 'OK',
7  '_updated': '2015-02-19T14:46:14Z',
   'id': 1,
9  'pdf': {'content_url': '/storage/somefile.jpg',
           'file': None,
11         'filename': 'somefile.jpg',
           'size': 55069},
```

```
13  'title': 'Some Offer'}
```

Note that ‘file’ in the response is None since returning as Base64 string is deactivated.

21.2 Working with study documents

Study documents are a collection of files. Using them is simple:

1. Create a study document (POST to /studydocuments)
2. Save ID of the newly created document
3. Upload files to the ‘/files’ resource as described above, using the ID

22 Common Problems

22.1 PATCH, PUT or DELETE returns 403

It is only possible to issue these methods on objects, not on resources. This will not work:

```
1 DELETE /users?where=id==3
```

Use this instead:

```
1 DELETE /users/3
```

Make sure you provided the required If-Match header. If that does not help make sure you can use GET on the item. If you are unable to request a GET then your account can not access the object. If you are able to GET the object, then your provided data is invalid. If you do not have admin priviledges for the endpoint(method on that resource) make sure your request will conserve your ownership of the object.

22.2 How can I send boolean etc to the server using python requests?

To properly encode Integer, Boolean and such you need to properly format the data to json before sending, like this:
Code:

```
import json

data_raw = {'spots': 42,
            'is_public:' True}

payload = json.dumps(data_raw)
```

The payload is now ready for sending! (Be sure to set the 'Content-Type' header to 'application/json')