1 Methods

The following sections include the steps taken to create the BOINSO network applications and the way the distinct components exchange information.

1.1 BOINSO Core Web Application

1.1.1 Core Data Model

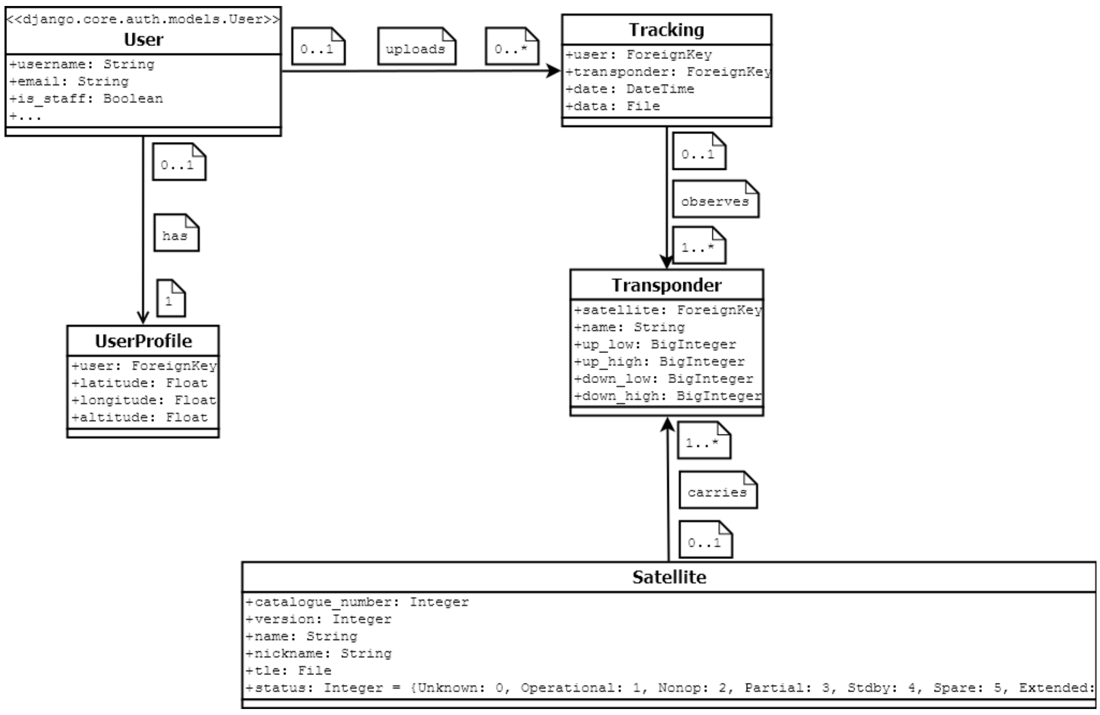


Figure 1: BOINSO Core Web Application model graph

As seen in figure 1 the model graph was modeled with only a view important entities as the initial functionality only includes passive satellite passes – meaning the tracking of a re-occurring transponder transmission. The Django core implementation of the user model was incorporated into the graph as it already included extensive integration in the authentication and permission system. The extension of the user profile was used to add **GCC!** (**GCC!**) related information which offers researches means for statistical segmentation.

1.1.2 Data model migrations

In order to implement changes to the data model (during initial development as well as the future) Django's migration system was used. The migration system created script files which allowed to make substantial changes to the underlying data source in a portable and retractable manner. A typical migration file as seen in listing 1 provides semantic naming, chronological enumeration and the possibility to transport model and data changes to all collaborating developers using the common source code repositories.

```python
# -*- coding: utf-8 -*-
from __future__ import unicode_literals

from django.db import models, migrations


class Migration(migrations.Migration):

    dependencies = [
        ('core', '0003_satellite'),
    ]

    operations = [
        migrations.AddField(
            model_name='satellite',
            name='catalogue_number',
            field=models.IntegerField(default=1),
            preserve_default=False,
        ),
    ]
```

Code 1: BOINSO Core Web Application model migration file including the addition of an integer typed catalogue number field to the satellite model.

1.1.3 Web API

Table 1 shows the different **API!** (**API!**) endpoints which were implemented to interact with a **MCC!** (**MCC!**) in a programmatic way using **HTTP!** (**HTTP!**) verbs. As almost every programming language has a certain amount of networking capabilities being able to make **HTTP!** calls it was seen as the most reasonable approach to make the client development as open as possible. In order to make the **API!** usable for open scripted clients as well as closed compiled clients an implementation of the OAuth2 authorization framework as stipulated in [**?**].

In order to keep users relatively independent a user profile (once registered at the **MCC!** web application) poses as an application (using the OAuth2 vernacular) making the access to OAuth2 protected endpoints uniform on all client implementation. The OAuth2 provider used

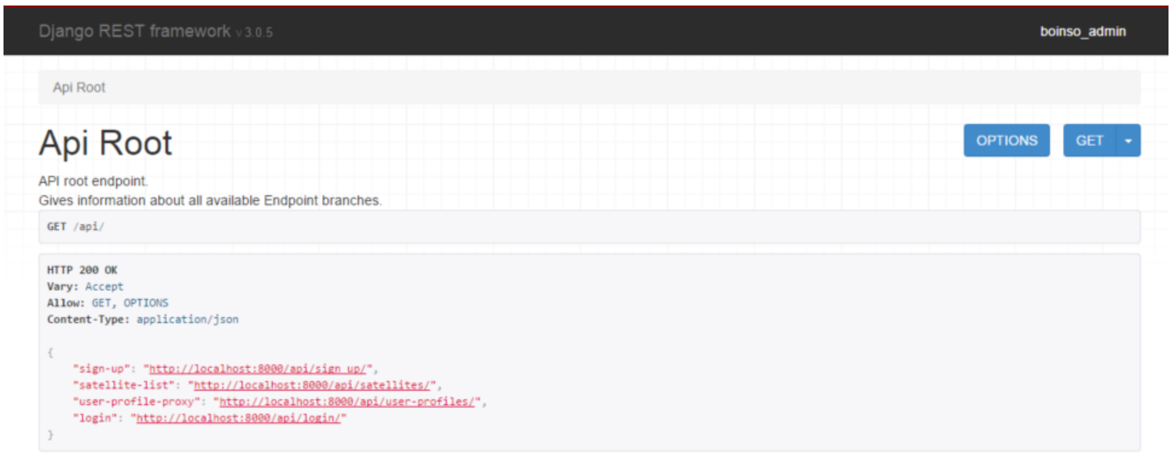
Method	**Endpoint**	**Usage**	**Returns**	**Auth**
POST	/api/sign_up/	Sign up new **GCC!**	OAuth2 credentials	None
GET	/api/login/	Retrieve registered **GCC!**	OAuth2 credentials	HTTP Basic
GET	/api/satellites/	Retrieves list of available satellites	Array of satellites	None
GET	/api/satellites/:id/	Retrieves satellite with specific ID	Satellite	None
GET	/api/transponders/:id/	Retrieves transponder with specific ID	Transponder	None
GET	/api/user-profiles/	Retrieves user related to auth token	User profile	OAuth2
GET	/api/user-profiles/:id/	Retrieves user profile with specific ID	User profile	OAuth2
PUT/PATCH	/api/user-profiles/:id/	Updates user profile with ID	User profile	OAuth2
DELETE	/api/user-profiles/:id/	Deletes user profile with specific ID	Deleted Notification	OAuth2

Table 1: Web API specifications listing method with HTTP-verbs relative endpoint **URL!** a short usage note a short description of the returned values and the used authentication type

in the Boinso Core Web Application was added using the extensive Django OAuth Toolkit distributed under the **BSD!** (**BSD!**) license. Utilizing the capabilities of the provided tools it was possible to both implement the Boinso MCC Web Client as well as the BOINSO GPredict Bridge with a web native callback driven authentication work-flow even though the initial configuration of the BOINSO Core Web Application was considerably more complex.

To guarantee a secure transmission of authentication data (login calls require **HTTP!** Basic authentication which is relatively easy to decipher while OAuth2 access tokens are not required to be encrypted) a **SSL!** (**SSL!**)/**TLS!** (**TLS!**) encrypted connection should be used. **MCC!**s can create their own certificates but should also consider investing in signed certificates by accredited institutions.

Besides the functionality of the web **API!** and its versatile authentication system another feature of the used components include an automatically generated browsable **API!**. It was not part of this project to change the design or branding which leads to the **API!** root in figure 2 stills showing the Django Rest Framework base style.



Api Root

API root endpoint.
Gives information about all available Endpoint branches.

```
GET /api/
```

```
HTTP 200 OK
Vary: Accept
Allow: GET, OPTIONS
Content-Type: application/json

{
    "sign-up": "http://localhost:8000/api/sign_up/",
    "satellite-list": "http://localhost:8000/api/satellites/",
    "user-profile-proxy": "http://localhost:8000/api/user-profiles/",
    "login": "http://localhost:8000/api/login/"
}
```

Figure 2: Browsable API root view. API components are interconnected by URLs allowing users both manually as well as programatically discovering API components.

1.2 BOINSO MCC Web Client

The BOINSO MCC Web Client was constructed as a sample client implementation, an easy to use and extend web application template for **MCC!**s and the first interaction point for **GCC!**s. The main focus was to keep it as simple as possible while giving users a intuitive experience making it possible for them to register with a **MCC!**, update their profile information and get informed about the **MCC!**'s satellites.

Using AngularJS, it's ngResource module and the capabilities of the Django **CORS!** (**CORS!**) Headers module (in the BOINSO Core Web Application) the application was build to request and load user data – if the authentication process resulted in positive application feedback – asynchronously and rather than interrupting user commands by alerts in cases of failure redirecting him or her to a state where the problem can be countered.

In AngularJS this behavior could be implemented by using **HTTP!** interceptor chains like seen in listing 2 making extensive use of lazy dependency injection to avoid the risk of circular constructor dependencies.

```
$httpProvider.interceptors.push(['$injector', function($injector) {
    return {
        responseError: function(rejection) {

            // I inject these services via implicitly calling the $injector
               service
            // if I don't do it like this I get a circular dependency error
            // I wish that some day I fully understand my doings
            var authService = $injector.get('authService');
            var $state = $injector.get('$state');
            var store = $injector.get('store');
            var $q = $injector.get('$q');

            if (rejection.status === 401) {
                $state.go('home');
                store.remove('auth_token');
                authService.login();
            }

            return $q.reject(rejection);
        }
    };
}]);
```

Code 2: Example for a HTTP interceptor reacting to HTTP 401 messages redirecting the user to a log-in overlay.

1.3 BOINSO GPredict Bridge

The BOINSO GPredict Bridge was build as a small functional piece of middleware connecting the BOINSO Core Web Application with the user's **GCC!** client machine. Private radio enthusiasts tend to invest a lot of time in their – quite expensive – observation set-ups. The machine controlling the hardware (radio rig, rotator controller and eventually additional peripherals) are optimized for maximal stability which makes their maintainers very skeptical of new components. Considering the module had to be easy to install, update and use as well as being very portable (supporting even Windows XP machines had to be considered) the application was written in pure Python with only few and rather common dependencies (like the formidable requests module).

As the idea behind the module was to introduce satellite tracking automation it was constructed that – if configured correctly – can be run autonomously via Unix cron jobs or Windows services. The configuration is generated in the users home directory (as for instance a vim.rc file would be) and contains the user information, the root **URL!**s of the affiliated **MCC!**s as well as related health and skip flags. The configuration files are written as valid **JSON!** (**JSON!**) files.