GeoDjango and LADM II: from Conceptual Model to Implementation

Jan VAN BENNEKOM-MINNEMA, The Netherlands

Keywords: LADM, LADM Implementation, Model Driven Architecture, Django, GeoDjango, ORM

SUMMARY

The Land Administration Domain Model (LADM) is a platform independent, conceptual information model describing the classes, attributes and associations related to administration of rights, responsibilities and restrictions affecting land and their geometrical (geospatial) properties. The (automatic) conversion and implementation of this conceptual model into a platform specific model, i.e. a relational database, is faced with many challenges. The LADM is currently under development as a multipart International Standard (i.e. LADM Edition II), and one of its parts is proposed to be dedicated to the implementation of the LADM.

An experiment with the conversion of the LADM to an implementation in the open source database PostgreSQL has been executed with Django and its extension GeoDjango. Django is an open-source web development framework with an Object-Relational Mapper (ORM) which has been utilised for this implementation. The primary goal of an ORM is to transmit data between the object-oriented platform independent model and the underlaying database. GeoDjango extends the ORM regarding querying and manipulating *spatial* data.

With the GeoDjango ORM, a substantial part of the LADM could, relatively straightforward, be implemented in the database, with support for most of the LADM classes, attributes, associations and constraints. An operational web framework was automatically generated, as a basis for future web application development, in which the data manipulation (create, retrieve, update, delete) is fully handled by the ORM based on LADM similar classes. As part of this framework, a web-based graphical user interface can be generated to support user interaction with the data.

GeoDjango and LADM II: from Conceptual Model to Implementation

Jan VAN BENNEKOM-MINNEMA, The Netherlands

1. INTRODUCTION

The Land Administration Domain Model (LADM, ISO 19152:2012) is a conceptual and platform independent information model in the Unified Modeling Language (UML), describing classes, associations and constraints related to land administration. One of the goals of LADM is to *provide an extensible basis for the development and refinement of efficient and effective land administration systems, based on a Model Driven Architecture (MDA)*. Countries and organisations considering the design and development of a land information system based on the LADM will need to convert and implement this conceptual model into a platform specific model, such as a relational database management system. The LADM is currently under development as a multipart International Standard [Lemmen, et al., 2020], referred to as LADM Edition II, and Part 6 is proposed to be dedicated to the implementation of the LADM.

Object-Relational Mapping (ORM) plays a role in a Model Driven Architecture in separating business and application logic from underlying platform technology [OMG, URL 6]. An ORM facilitates the model-driven conversion of conceptual classes into implementation objects, such as tables in a relational database. Research has been done regarding the automatic model driven conversion of the LADM into implementation [Van Bennekom-Minnema, 2008; Hespanha et al., 2008; Kalogianni et al., 2017, Alattas et al., 2018], identifying many challenges related to the models and the tools used for modelling, conversion and implementation. This is the case for the classes, attributes and associations, and even more for the constraints documented in the LADM.

This paper describes an experiment with the implementation of LADM objects, based on an ORM, facilitating the automatic implementation of classes, associations and constraints. The components of the IT architecture, used for the experiment, are described in the next section 2. Section 3 provides an overview of how the object-relational mapping process was configured and used: the package structure, classes, attributes, associations, required changes in LADM, constraints and testing with example instances. Then, in section 4, the configuration of the model-driven graphical interface is discussed, followed by conclusions and recommendations in section 5. Internet addresses (URLs), referring to the tools used, are provided after the references.

2. IT ARCHITECTURE COMPONENTS

There are many ORM tools available, e.g. Hibernate, SQLAlchemy, Sequelize, Entity Framework. For this LADM implementation, the ORM as part of the Django Web Framework was deployed. The other components are the PostgreSQL/PostGIS (database), Nginx (web application server), and Docker (packaging and deployment of development, test and production environment.

Django is an open-source web framework [Django, URL 1], based on the Python programming language for building web applications [Python, URL 8]. Django is based on a model-template-view software design pattern where the *model* component handles the data through the ORM, the *template* component handles the presentation (i.e. the graphical user interface of the application) and the *view* component handles the business logic and interaction between model and template. Django's ORM can handle geometry through **GeoDjango**, a geographic Web framework [GeoDjango, URL 4], which uses opensource spatial libraries such as GEOS, supporting the OpenGIS Simple Features for SQL, PROJ.4, supporting cartographic projections, and GDAL, providing spatial functions. GeoDjango's support for 3D data has its limitations (e.g. regarding some 3D datatypes), in this implementation the focus has been on two-dimensional data.

Django can operate with a number of relational databases, for example MySQL, Oracle, SQLite; for this experiment the open source database **PostgreSQL** with extension PostGIS for spatial data has been chosen, [PostGIS, URL 7].

Django requires a web application server to host the Django web application (including the ORM) which is realised based on **Nginx**, open source software for web servers [Nginx, URL 5]. The deployment of all components into an operational web application has been realised with open source software **Docker**, which facilitates and automates packaging and deploying applications based on *containers*. Containers are standardized units of software, isolated from and with few dependencies to its hosting environment/server [Docker, URL 2]).

3. OBJECT RELATIONAL MAPPING

An Object Relational Mapping tool specifies the relationship (i.e. the mapping) between objects in the conceptual model, also referred to as the *application model*, and the implemented objects in a relational database. Based on this mapping, the ORM will automatically generate and execute the DDL (Data Definition Language) to define and create the database objects (e.g. tables, primary, unique and foreign key constraints). Once the model is implemented in the database, any change to the application model (new or changed classes, attributes or associations) will lead to changes to the database, in Django referred to as *migrations*. Dependent on the nature of changes, and the existing data in the database, these migrations can be implemented fully *automatically*.

Interaction with the data in the database is based on elements of the application model; for example a developer of a land information system based on an ORM will interact with objects/instances of classes of the application model, and the ORM takes care of automatically generating the DML (Data Manipulation Language), i.e. the SQL to create, retrieve, update and delete data instances. Typically, an ORM supports different relational databases, which makes it relatively easy to switch the relational database; since the application functionality is based on the conceptual model, with the ORM 'translating' this to the database.

The LADM classes and associations are captured in the UML tool Enterprise Architect (EA, URL 3, Figure 1), which need to be provided in a Django application model. The Django model is documented in text files, and while in principle it is possible to use the model transformation possibilities in EA to generate these Django model text files, in this experiment they have been created and entered manually.

3.1. Implement LADM Package Structure

The first step in this manual entry of LADM classes into Django model classes is to implement the LADM packages, which have been structured into Django file system folders (also referred to as "apps"):

ladm_administrative_pkg:
Administrative Package with basic administrative unit and rights.

ladm_party_pkg:
Party Package with parties in those rights.

ladm_spatial_pkg:
Spatial Unit Package with the sub package Surveying and Spatial Representation.

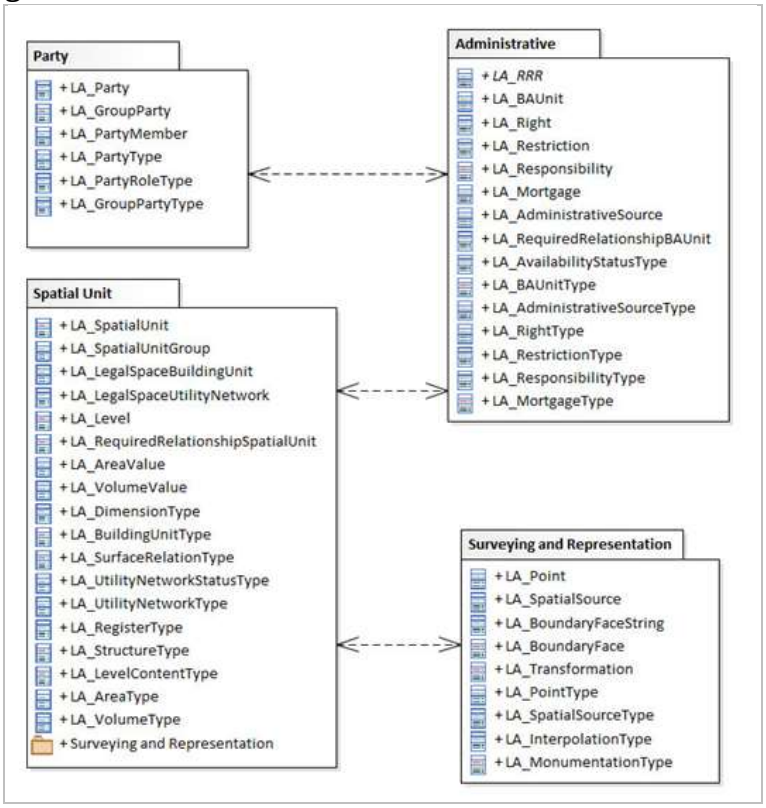


Figure 1 LADM packages and classes

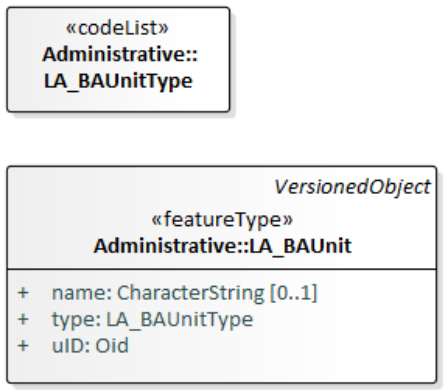
ladm_config_pkg
The LADM code lists have been implemented in Django folder *ladm_config_pkg* (code lists and other configuration classes).

The implementation of the Django model in the PostgreSQL database operates with prefixes in the table name indicating the origin in the model, resp. "a_", "p_", "s_", "c_" for 1) administrative, 2) party, 3) spatial unit, surveying and representation, and 4) configuration and settings.

For example: the LADM class *LA_BAUnit* is part of the LADM Administrative Package, which is manually described as class *BAUnit* in the Django app *ladm_administrative_pkg* and the text file "models.py" (See Figure 2). Django will automatically generate a table name for a class, unless class meta parameter *db_table* is specified, in this case table ***a**_ba_unit*.

The code list *LA_BAUnitType* is specified as Django class *BAUnitType* and implemented as database table ***c**_ba_unit_type*.

LADM UML Class



Django Model Class (*models.py*)

```
class BAUnit(VersionedObject):
  name = CharField(max_length=50, null=True)
  type = ForeignKey(BAUnitType)
  spatial_units = ManyToManyField(SpatialUnit)
  class Meta:
    db_table = 'a_ba_unit'
```

PostgreSQL Table DDL

```
CREATE TABLE public.a_ba_unit
( id integer NOT NULL
  DEFAULT nextval('a_ba_unit_id_seq'::regclass)
, name character varying(50)
, type_id integer NOT NULL
, CONSTRAINT a_ba_unit_pkey PRIMARY KEY (id)
, CONSTRAINT a_ba_unit_type_id_fk
   FOREIGN KEY (type_id)
        REFERENCES public.c_ba_unit_type
          (ladmtype_ptr_id)
)
```

Figure 2 Implementation class LA_BAUnit

With the Django package/application structure in place, the classes can now all be specified in their *models.py* text files.

3.2. Implement LADM Classes and Attributes

LADM classes and Django application classes appear to be quite similar i.e. object-oriented. Django supports inheritance of **super classes**, e.g. abstract super class *VersionedObject* and *LA_RRR*. For example, the attributes *RRR.share & description* will be inherited by sub classes Right, Restriction, and Responsibility.

Attributes can be specified with a range of **datatypes** (e.g. *CharField, IntegerField, DecimalField, DateField, TimeField, BooleanField, BinaryField*). The optional *([0..1]*) LA_BAUnit.name is modelled in the Django model as a text field BAUnit.name: CharField(max_length=50, null=True).

If a unique identification of a class instance is not specified, Django will automatically generate a **primary key** based on column "id" with a positive integer datatype, which is not explicitly modelled in the application model.

3.3. Implement LADM Associations

The LADM contains many attributes referring to *codelists*, for example: the datatype for the attribute LA_BAUnit.type is in essence is a "*one-to-many*" relationship from LA_BAUnit to **codelist** class LA_BAUnitType, and will therefore be modelled as *datatype* **foreign key:** BAUnit.type: ForeignKey(BAUnitType), see Figure 2.

LADM also contains quite some "*many-to-many*" associations, which in Django can be modelled as *datatype* **ManyToManyField**. For example: the association between LA_BAUnit and LA_SpatialUnit is modelled with the attribute: BAUnit.spatial_units: ManyToManyField(SpatialUnit), see Figure 2. This association is implemented in the database as a table *a_ba_unit_spatial_units* (based on class meta parameter *db_table*). The association is specified on the side of BAUnit and will, in the ORM, automatically be available as an attribute on the class on the other side: SpatialUnit.ba_units. If a "many-to-many" association has attributes, like for example: the attribute LA_PartyMember.share, the association will be modelled as datatype ManyToManyField *and* a class (Figure 3).

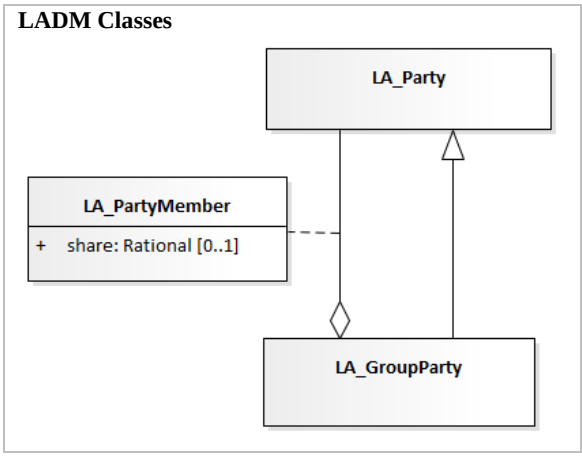


Figure 3 Implementation class LA_PartyMember, LA_Party, LA_GroupParty

3.4. Changes to LADM

During the manual entry of classes in the Django model, a few issues where found. For example: Django does not allow **inheritance of attributes** with the same name as an existing attribute, as can be found in the cases of LA_Party and subclass LA_GroupParty (attribute: *type*) or LA_Restriction and sub class LA_Mortgage. The chosen workaround for this issue has been to rename the attributes of the subclass, e.g. the attribute *LA_GroupParty.type* is implemented as *GroupParty.group_type*.

Another issue was related to the invariant that applies to LA_RRR which states "*share must be specified, unless this is meaningless for the specific type (of right)*" which seems to contradict with *share_check* being an attribute of class LA_RRR; in this implementation, **share_check** is an attribute of classes RightType, RestrictionType, and ResponsibilityType and constraints where made to only check the share if e.g. the RightType.share_check requires so (see next section on constraints, Figure 5, row 4: "`filter(share_check=True)`").

New classes were specified in the Django model, partly inspired by experiences from previous implementations: a class *Person* with subclasses *NaturalPerson* and *NonNaturalPerson*, and an optional attribute *Party.person* (i.e. a foreign key referring to the new class Person), a codelist SpatialUnitGroupHierarchyType (to assist with LA_SpatialUnitGroup.hierarchyLevel, see example in Figure 11), and codelists GenderType, CurrencyType, NonNaturalPersonType.

Geometry in the LADM is handled in the Spatial Unit Package, for example through classes *LA_BoundaryFace* (MultiSurface), *LA_BoundaryFaceString* (MultiCurve) and *LA_Point*, for storing original measurements and transformed or adjusted final data. While these classes can be successfully implemented for 2D data with GeoDjango, a **new geometry attribute** has been added to SpatialUnit; this is also envisioned as a change in the LADM Edition II.

3.5. Implement LADM Constraints

As shown in Figure 2, **attribute constraints** like for example datatype, mandatory or optional, and maximum length can be implemented as *database* constraints. Constraints concerning one or more attributes of the same instance, also called tuple constraints, can be specified in the application model and implemented as database constraints as well. For example: a check constraint on the association class LA_PartyMember.share (Figure 4, row 1), which is specified in a Django specific syntax resulting in database objects (*`gte=0 ~ greater-than-or-equal to 0, lte=1 ~ lower-than-or-equal to 1`*). Row 2 shows the implementation of a unique key, in this case to avoid the same party being added more than once to a specific group.

Note that tuple constraints, defined on *abstract* classes, will not be implemented automatically by the ORM, for example: check constraints regarding the attributes *Oid*, *beginLifespanVersion & endLifespanVersion* of class VersionedObject, or regarding attributes of the class *LA_RRR*. These will need to be specified as part of their sub-classes in the application model (*for the classes LA_RRR: LA_Right, LA_Restriction, LA_Responsibility*), or alternatively, the abstract classes can be specified as concrete (non-abstract), after which Django will implement these as tables with check constraints.

No	Description	Django

1	**Check constraint on LA_PartyMember.share (between 0 and 1)**	`class Meta:`
		`  db_table = 'p_group_party_member'`
		`  constraints = [CheckConstraint(check =`
		`  Q(share__gte=0) | Q(share__lte=1 )]`

2	**Unique constraint on LA_PartyMembers**	`class Meta:`
		`  db_table = 'p_group_party_member'`
		`  unique_together = ['group', 'party']`

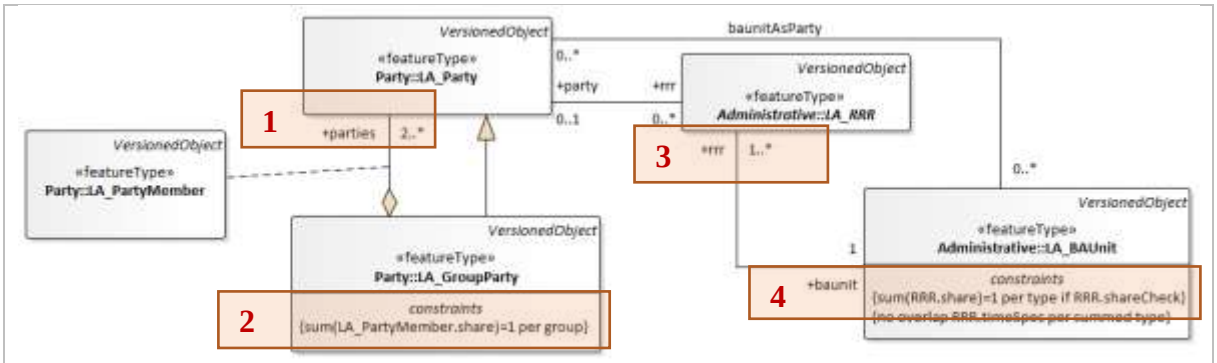
Figure 4 Implementation of LADM constraints in database

Constraints in the LADM are defined in the Object Constraint Language (OCL), and some of these are assertions that concern multiple tuple/instances of the same class, or instances of other classes, also referred to as **cross-row and multi-table check constraints**. This type of constraint requires the execution of database queries, sometimes into the same table that is being updated, which in many relational databases is not possible. A full implementation of this constraint in the database would require custom development (programming) e.g. based on

database triggers. An alternative to full database implementation of these assertions would be to implement these (only) on the application side, in the Django model.

Consider for example Figure 5, row 1, "**a Group must at least have two member parties**". This is implemented as attributes *ladm_constraints* (list) and *check_ladm_constraints* (method) for the class VersionedObject, which are inherited by the classes LA_Party and LA_GroupParty. For the class LA_GroupParty, a ladm_constraint "check_two_party_members" is added (`GroupParty.party_members.count() >= 2`). The attribute *GroupParty.check_ladm_constraints* will be empty if all constraints pass or will provide the list of violated constraints. While it is possible in the database to have a Group with less than 2 members, violating the constraint, it would be easy to select these Group objects through the application model.

Another constraint "**Sum of Group member shares must be 1**" is implemented as "check_total_member_shares" for GroupParty, see Figure 5, row 2. The constraint "**a BAUnit must have at least one RRR**" is depicted on row 3.
Row 4 shows a constraint which requires some more coding based on ORM model classes: "**Sum of BAUnit shares must be 1 per Right Type**", which is implemented in the same structure based on attributes ladm_constraints and check_ladm_constraints.

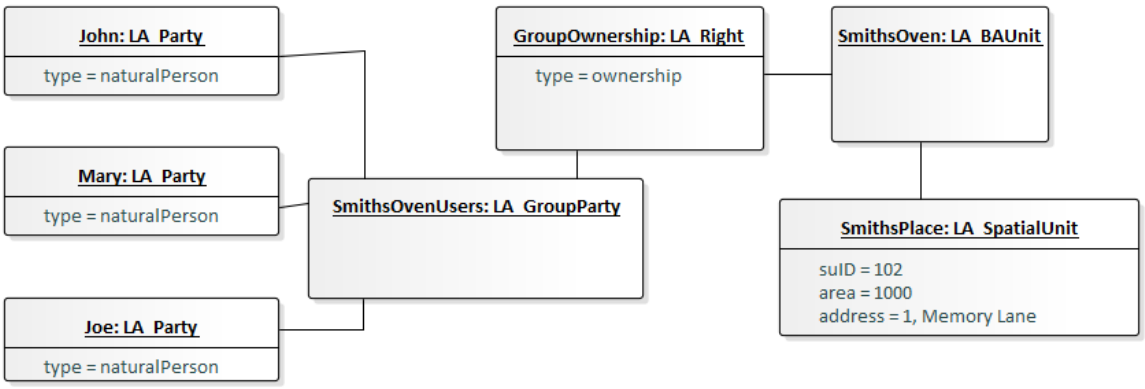


No	Description	Django
1	**Group must have at least two member Parties [2..*]**	*self=GroupParty -> check_two_party_members* **self.party_members.count() >= 2**
2	**Sum of Group member shares must be 1**	*self=GroupParty -> check_total_member_shares* **GroupPartyMember.objects.filter(group=self)** **.aggregate(Sum('share'))['share__sum'] = 1**
3	**A BAUnit must have at least one RRR [1..*]**	*self=BAUnit -> check_one_rrr* **self.rights.count()+self.restrictions.count()** **+self.responsibilities.count() >= 1**
4	**The sum of BAUnit shares must be 1 per Right Type** *(unless this is meaningless for the specific type)*	*self=BAUnit -> check_share_one_for_right_type* **# make a distinct list of right types** *right_types =* *RightType.objects.filter(right__ba_unit=self)* *distinct().filter(**share_check**=True)* *for right_type in right_types:* **# loop through right types and calculate the sum of shares for each** *share_rights = self.rights.filter(type=right_type)* *.aggregate(Sum('share'))['share__sum']* *# validate the constraint* *if share_rrrs != 1:* *check_passed = False*

Figure 5 Implementation of LADM constraints in application model

3.6. Interact with LADM Instances

Annex C of ISO 19152 shows examples of instances which have been used to test data manipulation via the application model (as opposed to SQL queries on the database). Consider example C.5 Group Party (Figure 6, row 1-6):



LADM C.5 Group Party

No	Description	Django
1	*smiths_oven*	`smiths_oven = BAUnit(name="Smith's Oven", type="basic_property_unit")`

2	*smiths_place*	`smiths_place = SpatialUnit(label="Smith's Place", address="1, Memory Lane")`
		`area = AreaValue(size=1000, type="surveyed_area" )`
		`smiths_place.areas.add(area)`

3	associate *smiths_oven with smiths_place*	`smiths_oven.spatial_units.add(smiths_place)`

4	*john, mary, joe smiths_oven_users*	`john = Party(type="natural_person", name="John")`
		`mary = Party(type="natural_person", name="Mary")`
		`joe = Party(type="natural_person", name="Joe")`
		`smiths_oven_users = GroupParty(`
		`name="Smith's Oven Users",`
		`type="group",`
		`group_type="partnership")`

5	*Associate john, mary, joe with smiths_oven_users*	`smiths_oven_users.party_members.add(john, mary, joe)`

6	*Create right for smiths_oven_users to smiths_oven*	`group_ownership=Right(type="ownership"),`
		`ba_unit=smiths_oven,`
		`party=smiths_oven_users)`

7	*Query all BAUnit instances*	`all_baunits = BAUnit.objects.all()` `all_baunits.count() -> 52`

8	*Query BAUnit by primary key and get its spatial_units*	`baunit = BAUnit.objects.get(pk=11)`
		`print(baunit) -> Smith's Oven [11]`
		`baunit.spatial_units.count() -> 1`
		`baunit.spatial_units.all() ->`
		<QuerySet [<SpatialUnit: Smith's Place>]>

9	*Query BAUnit instances where name starts with "Gaia"*	`baunits = BAUnit.objects.filter(name__startswith="Gaia")`
		`baunits.count() -> 6`
		`print(baunits) ->`
		<QuerySet [<BAUnit: Gaia Grazing Grounds [10]>, <BAUnit: Gaia Lodge [15]>, <BAUnit: Gaia Social Centre [16]>, <BAUnit: Gaia Subnet [23]>, <BAUnit: Gaia Fuel Station Protection Area [34]>, <BAUnit: Gaia Fuel Station [35]>]>

Figure 6 Test application model with ISO 19152 Annex example

All instantiations are done based on classes of the application model: for example: create BAUnit "Smith's Oven" and corresponding SpatialUnit "Smith's Place"; create a GroupParty "Smith's Oven Users" consisting of Party John, Mary and Joe; and create an "ownership" Right for this group.

After instantiating the Annex C examples, the records can be queried through the application model classes, see some example in Figure 8, row 7-9).

4. MODEL DRIVEN GRAPHICAL USER INTERFACE

Once the Django application model is defined in the "models.py" text files, the ORM will take care of managing the objects in the relational database, as wells as the data manipulation (as shown in Figure 6). Based on the model, Django can also generate a web-based graphical user interface (GUI), with user authorisation and authentication, to maintain a choice of the LADM classes, for example the code lists.

By adding one line per class in the "admin.py" text file; simple web pages are created for viewing and maintaining the data. See the GUI generated for codelist class LA_GroupPartyType in Figure 7.

Text file "admin.py":

```
admin.site.register(GroupPartyType)
```

Django Administration GUI:

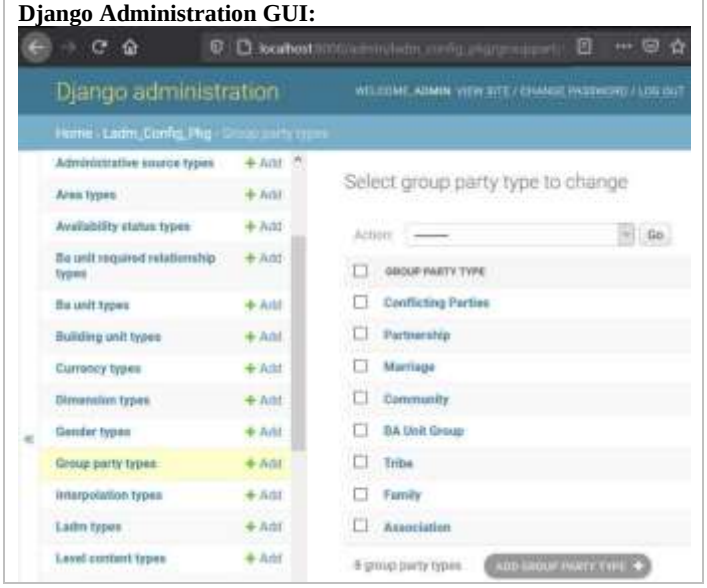


Figure 7 Example of generated GUI for GroupPartyType

More complex GUIs can be created with a few more lines in "admin.py", for example to build a *master-detail* page for a BAUnit and its Rights. Figure 8 shows the content for the admin.py file:

1. first the *detail* section for **Right** and its attributes (fields) is defined,
2. then the *master* section for **BAUnit**, followed by
3. a call to the section for rights (*RightInline*), and then finally,
4. informing Django that this should be part of the Admin GUI (`admin.site.register`).

```
1)  class RightInline(admin.TabularInline):
        model = Right
        readonly_fields = ('oid', 'begin_lifespan', 'end_lifespan')
        fieldsets = [
            (None, {'fields': ['begin_lifespan', 'end_lifespan']}),
            (None, {'fields': [('type', 'party','share')]}), ]

2)  class BAUnitAdmin(admin.ModelAdmin):
        readonly_fields = ('oid', 'begin_lifespan', 'end_lifespan',)
        search_fields = ['name']
        formfield_overrides = {
            models.TextField: {'widget': forms.Textarea(attrs={'rows': 2}},
        }
        fieldsets = [
            (None, {'fields': [('oid', 'begin_lifespan', 'end_lifespan')]}),
            (None   , {'fields': [('type', 'name'), 'remark']}),
            #(None   , {'fields': ['remark']}) ]
        3) inlines = [RightInline]

4)  admin.site.register(BAUnit, BAUnitAdmin)
```

Figure 8 Example of settings in admin.py text file for BAUnit and Rights

Figure 9 shows the GUI, generated based on the settings in the *admin.py in* Figure 8, in this case showing the rights in the example in Annex C.31 of ISO 19152.

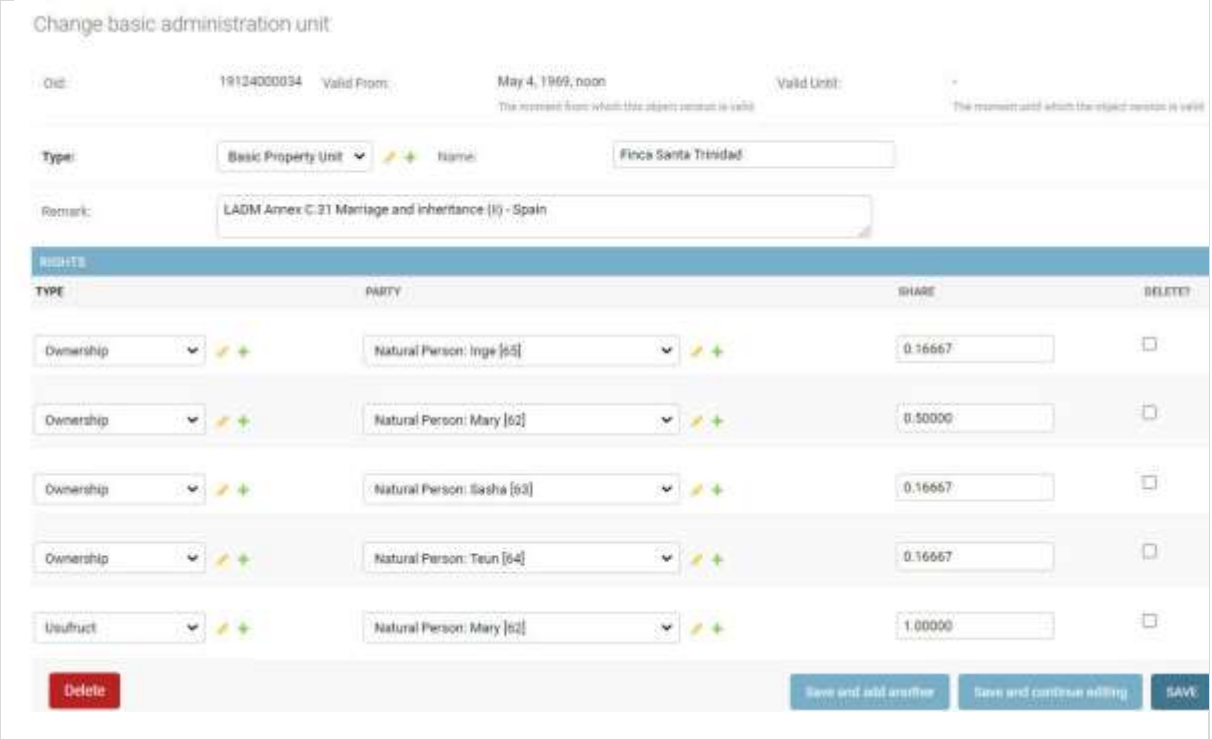


Figure 9 Example of generated GUI for BAUnit and Rights

Before, in Figure 6, row 9, a query is shown, executed through the ORM python interface, filtering all BAUnit instances where its name starts with "*Gaia*". The file admin.py (in Figure 8) has a setting `search_fields=[name]`, which will generate a search field, allowing to execute a similar query through the GUI (Figure 10).

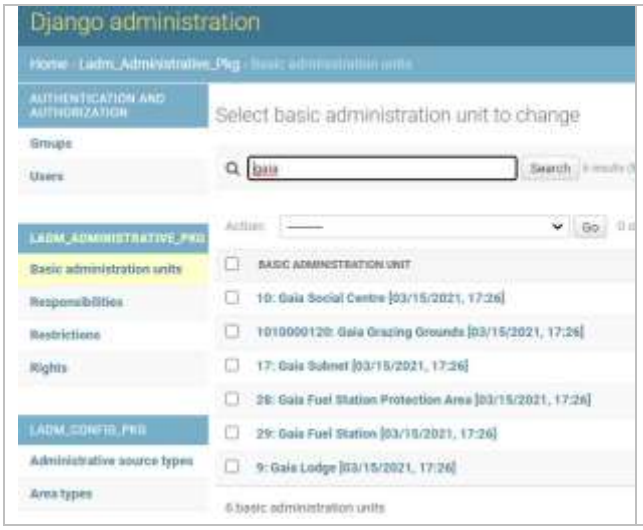


Figure 10 Query BAUnit instances in the GUI where name starts with "Gaia

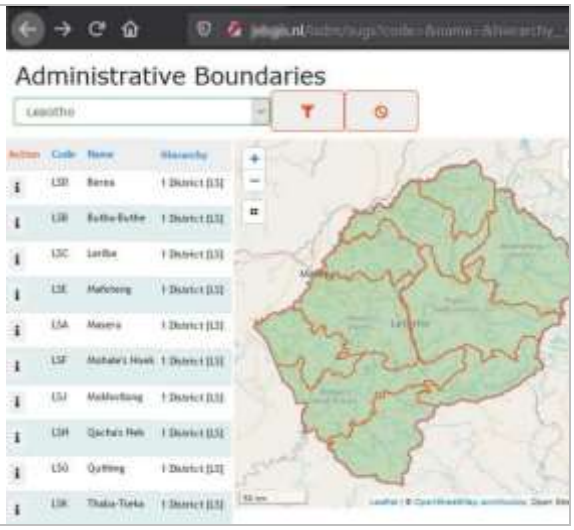


Figure 11 Example of custom developed Django Web application page for LADM class LA_SpatialUnitGroup

Note that Django's Admin is recommended for use as an organization's *internal* management tool for class instances and is not intended for building the entire *front-end web application* available to all users. See Figure 11 for an example of a custom-developed Django Web application page, based on GeoDjango application model and Leaflet (open-source library for creating interactive web maps), to list and maintain the class LA_SpatialUnitGroup, including geometry.

5. CONCLUSIONS AND RECOMMENDATIONS

With the components as described in section 2, the environment can be (re-)deployed with a simple (docker) command which then 1) creates the containers GeoDjango, PostgreSQL database and Nginx web application server, 2) migrates and links the application model classes to the PostgreSQL database objects, 3) loads the code lists (ISO 19152 Annex J) and the example data (ISO 19152 Annex C) through the object-relational mapper (ORM).

ISO 19152 Annex A describes how the conformance of an implementation of LADM classes, attributes and associations can be assessed: i.e. Level 1, for only the basic classes, Level 2, for basic and common classes, or Level 3 for all LADM classes. With the GeoDjango ORM a *Level 3 conformance* could be reached, except for specific 3D geometry that Django does not support.

Therefore, the utilisation of an ORM for this experiment with the implementation of LADM is deemed successful. After manually specifying the LADM classes as application model classes in the ORM, an *automatic implementation* of the classes as tables in the underlying database can be realised. Both the database object creation and the data manipulation (i.e. create, retrieve, update, delete) is handled fully though application classes and objects; the ORM generates the required SQL regarding the database objects and records. Based on the application model, (Geo)Django also automatically generates a web based graphical user interface to support data manipulation.

Further research is recommended and could be made on: 1) creating and testing a full implementation of LADM II with an (GeoDjango) object-relational mapper - possibly contributing to Part 6 of LADM II; 2) the generation of a simple web graphical user interface to manipulate all the LADM objects; 3) the use of generating the Django application model from the LADM UML (in the tool Enterprise Architect); 4) the approach to implementing all (OCL) constraints in LADM; 5) the implementation of abstract class VersionedObject; and 6) the 3D capabilities and limitations of GeoDjango.

REFERENCES

- Abdullah Alattas, Peter van Oosterom, Sisi Zlatanova (2018) Deriving the Technical Model for the Indoor Navigation Prototype based on the Integration of IndoorGML and LADM Conceptual Model, In: Proceedings of the 7th Land Administration Domain Model Workshop, Zagreb, pp. 24
- Van Bennekom-Minnema, J. (2008). The Land Administration Domain Model 'Survey Package' and Model Driven Architecture, Master's thesis, GIMA (TUD, UT-ITC, UU, WUR), pp. 199, 2008
- Eftychia Kalogianni, Efi Dimopoulou, Wilko Quak, Michael Germann, Lorenz Jenni, Peter van Oosterom (2017) INTERLIS Language for Modelling Legal 3D Spaces and Physical 3D Objects by Including Formalized Implementable Constraints and Meaningful Code Lists, In: ISPRS International Journal of Geo-Information, MDPI AG, 6(10), pp. 319
- Hespanha J.P, Van Bennekom-Minnema, J., Van Oosterom, P.J.M., Lemmen, C.H.J. (2008). The model driven architecture applied to the Land Administration Domain Model version 1.1 with focus on constraints specified in the object constraint language, in Proceedings from FIG Working Week - Integrating Generations, Stockholm, Sweden. Federation International des Géomètres, June 2008, pp. 1-19
- Chrit Lemmen, Peter van Oosterom, Eva-Maria Unger, Eftychia Kalogianni, Anna Shnaidman, Abdullah Kara, Abdullah Alattas, Agung Indrajit, Katherine Smyth, Aurélie Milledrogues, Rohan Bennett, Peter Oukes, Hans-Christoph Gruler, Daniel Casalprim, Golgi Alvarez, Trias Aditya, Ketut Gede Ary Sucaya, Javier Morales, Marisa Balas, Nur Amalina Zulkifli, Cornelis De Zeeuw (2020) The land administration domain model: advancement and implementation, In: Proceedings of the Annual World Bank Conference on Land and Poverty, Washington DC, pp. 28

INTERNET ADDRESSES (URLS)

The following URL's have been referred to in this paper; all URL's were checked and available at March 24, 2021.

1. Django, a Python Web framework, https://www.djangoproject.com.
2. Docker, packaging and deployment of applications, https://www.docker.com.
3. Enterprise Architect, a modelling tool for business, software and systems, https://sparxsystems.com/.
4. GeoDjango, a geographic Web Framework, https://docs.djangoproject.com/en/3.1/ref/contrib/gis.
5. Nginx, open source web server, https://www.nginx.com.
6. OMG, MDA - The Architecture of Choice for a Changing World, https://www.omg.org/mda.
7. PostGIS, a spatial database extender for PostgreSQL object-relational database, https://postgis.net.
8. Python, programming language, https://www.python.org.

BIOGRAPHICAL NOTES

Jan van Bennekom-Minnema obtained and MSc in Geographical Information Management and Applications in 2008 from University of Utrecht. He is a consultant for COWI Denmark, working on international Land Administration and Spatial Planning projects on database and web application design & development, spatial data analysis, mobile device-based field data collection.

CONTACTS

Mr Jan van Bennekom-Minnema
Land Administration, GIS and IT Specialist
COWI A/S
Parallelvej 2,
Kongens Lyngby,
2800 Denmark
Email: jvb@cowi.com
Website: www.cowi.com