
>>> network .toCode()
Django ORM
Building Blocks Of Nautobot

Agenda

What is Django ORM?

Models & Fields

Django Queries

CRUD Operations

Nautobot Enhancements

What is Django ORM?

And how does it fit into a MVC pattern?

>>> What is Django ORM?

And how does it fit into a MVC pattern?

id (int)	name (char)	slug (char)	description (char)	location (char)	tz (char)
1	Object 1	object-1	Some description	NYC	EST (GMT-5)
2	Object 2	object-2	Another desc	HOU	CST (GMT-6)

>>> What is Django ORM?

ORM - Object Relational Mapper

Defines database schema as Python

Database schema held in stateful files

Django Migrations provide predictable upgrade paths in apps

One definition for *most* relational database backends

Interact with relational database without writing SQL

Object oriented interactions with related tables


Models & Fields
Pythonic SQL schema...ish

>>> Models & Fields

Pythonic SQL schema…ish

- Each Model is a table
- Each Field is a column in a table
- Each Instance is a row in a table
- Migrations are based on the Models
- Migrations are based on Models definitions

```python
from django.db import models
from timezone_field import TimeZoneField

from nautobot.extras.models import StatusModel
from nautobot.core.fields import AutoSlugField
from nautobot.core.models.generics import PrimaryModel


class Site(PrimaryModel, StatusModel):
    name = models.CharField(max_length=100)
    slug = AutoSlugField(populate_from="name")
    description = models.CharField(
        max_length=200, blank=True
    )
    region = models.ForeignKey(
        to="dcim.Region"
    )
    time_zone = TimeZoneField(blank=True)
    …Truncated for brevity…
```


Django Queries

>>> Django Object Relational Mapping

- Python object-based representation of a data model
  - **Class** (model): Database table
  - **Attribute** (field): Database column
  - **Instance**: Database row
- Allows reading and manipulation of an underlying database using Python methods and operations
- Far easier to use and maintain than raw SQL
  - Quicker to write, easier to read
  - Supports IDE integration for dependency tracking, etc.

>>> Django Models

```
class DeviceRole(Model):
    name = CharField(
        max_length=50,
        unique=True
    )
    slug = SlugField()
    color = ColorField(
        default="c0c0c0"
    )
    vm_role = BooleanField()
    description=CharField(
        max_length=200
    )
```

Column	Type
id (pk)	uuid
name	charvar(50)
slug	charvar(50)
color	charvar(6)
vm_role	boolean
description	charvar(200)

>>> nbshell

- Wraps the stock Django shell (Python interactive interpreter)
  - Automatically imports common classes

```
$ nautobot-server nbshell
 Django version 3.2.21

In [1]: Device.objects.count()
Out[1]: 320

In [2]:
```

- Three components
  - **Model** (class) - Database table representation
  - **Manager** - QuerySet generator (typically “objects”)
  - **Method(s)** - SQL operation wrapper; can be chained

```
DeviceRole.objects.filter(...).order_by(...)
```

>>> Django Querysets

Calling `repr()` on a QuerySet forces evaluation; returns a truncated list of instances

```
# Return all objects
In [1]: Platform.objects.all()
Out[1]: <RestrictedQuerySet [<Platform: Arista EOS>,
<Platform: Cisco ASA>, …]>

# Count all objects
In [1]: Platform.objects.count()
Out[1]: 7

# Return only objects matching certain filters
In [1]: Platform.objects.filter(name="Cisco IOS")
Out[1]: <RestrictedQuerySet [<Platform: Cisco IOS>]>
```

>>> Django QuerySets - Under the Hood

- Call print() on a QuerySet's query attribute to see the underlying SQL statement

```
In [1]: queryset = Platform.objects.filter(name="Cisco IOS")
In [1]: print(queryset.query)
SELECT "dcim_platform"."id", "dcim_platform"."created",
"dcim_platform"."description" FROM "dcim_platform" WHERE
"dcim_platform"."name" = Cisco IOS
ORDER BY "dcim_platform"."name" ASC
```

filter(name="Cisco IOS")

Default ordering; manipulated with order_by()

>>> Django QuerySets - Under the Hood

- QuerySets are lazy; modification will not trigger an actual database query until needed

```
# Modifying a QuerySet is free

In [1]: queryset = Platform.objects.all()
In [1]: queryset = queryset.filter(name__startswith='Cisco')
In [1]: queryset = queryset.order_by('name')

# Forcing evaluation of a QuerySet triggers a SQL query
In [1]: queryset
Out[1]: <RestrictedQuerySet [<Platform: Cisco ASA>, <Platform:
Cisco IOS>, <Platform: Cisco IOS-XE>, <Platform: Cisco
IOS-XR>, <Platform: Cisco NX-OS>]>
```

>>> Django QuerySets - Forcing Evaluation

Operation	Example
repr()	`repr(queryset)`
len()	`len(queryset)`
bool()	`if queryset:`
list()	`list(queryset)`
Slicing	`objects = queryset[:10]`
Iteration	`for obj in queryset:`

Use .count() instead!

Use .exists() instead!

>>> Common QuerySet Methods

Method	Description
all()	Default; needed only if no others are added, except delete()
filter()	Filter results by matching on one or more attributes
exclude()	Inverse of filter()
order_by()	Order results by a particular attribute and/or invert direction
get()	Get a single object, typically by PK (or raise ObjectNotFound)
first()	Get the first matching object (or return None)
last()	Get the last matching object (or return None)
exists()	Determine whether any matching object exists
count()	Return the count of all matching objects

>>> Common QuerySet Methods

Method	Description
select_related()	Perform a SQL JOIN to pull fields from related objects
prefetch_related()	Prefetch related fields in Python (separate database queries)
create()	Create a new object directly by passing attributes as kwargs
get_or_create()	Retrieve a matching object if it exists; otherwise, create one
bulk_create()	Create a set of objects by passing a list of new instances
update()	Set the specified attributes on all matching objects
update_or_create()	Update a matching object if it exists; otherwise, create one
bulk_update()	Update a set of objects by passing a list of existing instances
delete()	Delete all matching objects (must come after filter() or all())

>>> QuerySet Field Lookups

- Django provides many stock field lookups to tweak filtering
  - These map to SQL transformations
- Lookups are specified by concatenating the field name with a double underscore (“dunder”) and the lookup name

```
DeviceRole.objects.filter(name__startswith='A')
```

>>> QuerySet Field Lookups

```
In [1]: queryset =
Platform.objects.filter(name__startswith='Cisco')

In [1]: print(queryset.query)
SELECT "dcim_platform"."id", "dcim_platform"."created",
"dcim_platform"."last_updated", "dcim_platform"."description"
FROM "dcim_platform"
WHERE "dcim_platform"."name"::text LIKE Cisco%
ORDER BY "dcim_platform"."name" ASC

In [1]:
```

name__startswith='Cisco'

>>> Common Field Lookups

Method	Description
gt(), gte(), lt(), lte()	Greater/less than (or equal to)
startswith(), endswith()	Begins/ends with the specified string*
contains()	Contains the specified string*
in()	Value exists within the specified list of values
date(), time()	Match a specific date or time
year(), month(), day()	Match a particular date component (but not the whole thing)
regex()	Apply an arbitrary regular expression*
isnull()	Matches only if the specified field is NULL

>>> Spanning Object Relationships

- Related object fields can be filtered using an instance directly

```
# Retrieve the site we're interested in

In [1]: platform = Platform.objects.first()

# Retrieve all devices assigned to that platform
In [1]: Device.objects.filter(platform=platform)
Out[1]: <ConfigContextModelQuerySet [<Device: ams01-edge-01>,
<Device: ams01-edge-02>, <Device: atl01-leaf-01>,'...(remaining
elements truncated)...']>

# Alternate way
In [1]: Device.objects.filter(platform_id=platform.pk)
Out[1]: <ConfigContextModelQuerySet [<Device: ams01-edge-01>,
```

>>> Spanning Object Relationships

- Dunders can also be used to follow relationships across models
  - Allows filtering of one model by the properties of a related model
  - Ex: Find all devices within a site named “Company HQ”

```
Device.objects.filter(site__name='Company HQ')
```

Field

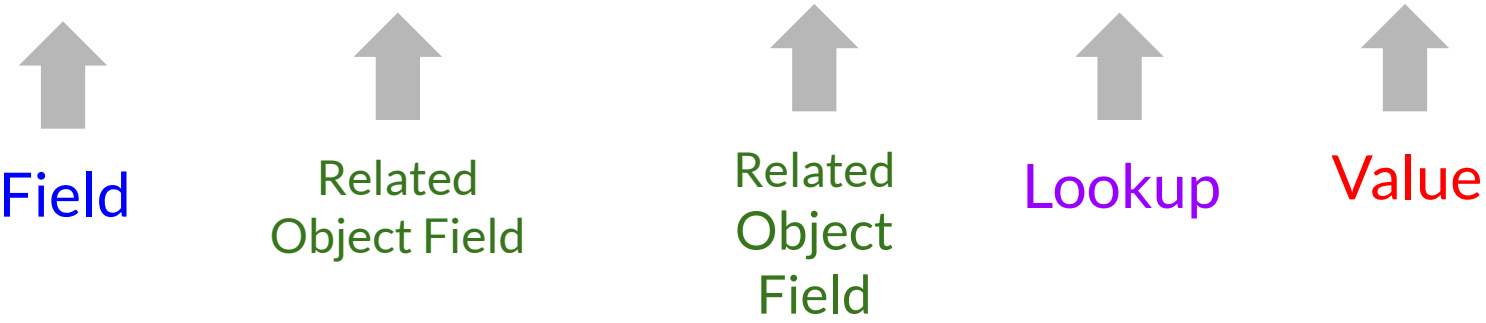
Related Object Field

Value

>>> Spanning Object Relationships

- Lookups can also be used on related fields
  - Ex: Retrieve all devices within a site assigned to a region whose name begins with “E”

```
Device.objects.filter(
  platform__manufacturer__name__startswith="Cis")
```



>>> Spanning Object Relationships

- Use with Caution! Spanning relationships incurs a performance penalty at query time

```
In [1]: print(queryset.query)
SELECT      "dcim_device"."id",
            "dcim_device"."secrets_group_id"
            ......
FROM        "dcim_device
inner join "dcim_platf
ON          (
                        "dcim_device"."platform_id" = "dcim_platform"."id")
inner join "dcim_manufacturer"
ON          (
                        "dcim_platform"."manufacturer_id" =
"dcim_manufacturer"."id")
WHERE       "dcim_manufacturer"."name"::text LIKE cis%
ORDER BY    "dcim_device"."_name" ASC
```

Spanning two relationships adds two INNER JOINs

>>> QuerySet Filtering - AND vs. OR

- Filters AND by default (multiple terms or chained methods)
- Use Q objects to perform a logical OR of multiple terms

```
# Find Devices that are active AND Cisco IOS
In [1]: cisco_ios = Platform.objects.get(name="Cisco IOS")
In [1]: Device.objects.filter(status__name="Active", platform=cisco_ios)

# Same as previous
In [1]: Device.objects.filter(status__name="Active").filter(platform=cisco_ios)

# Find sites that are active OR in US
In [1]: from django.db.models import Q
In [1]: Device.objects.filter(Q(status__name="Active") | Q(platform=cisco_ios))
```

>>> Finding Fields and Filters

Using Django Model's "_meta" methods, you can find the fields and filters

```
# Find fields for model "Prefix"
In [1]: Prefix._meta.get_fields()
Out[1]:
(<ManyToManyRel: ipam.vrf>,
 <nautobot.ipam.fields.VarbinaryIPField: network>,
 <nautobot.ipam.fields.VarbinaryIPField: broadcast>,
 <django.db.models.fields.IntegerField: prefix_length>,
 <django.db.models.fields.CharField: type>, …)

# Find filters for model "Prefix" field "network"
In [1]: Prefix._meta.get_field('network').get_lookups()
Out[1]:
{ 'gt': django.db.models.lookups.GreaterThan,
 'net_contained': nautobot.ipam.lookups.NetContained,
 'net_contained_or_equal': nautobot.ipam.lookups.NetContainedOrEqual,
 'net_contains': nautobot.ipam.lookups.NetContains,
}
```

Field name

Filter name

In [71]:

Lab 1

>>> Lab 1 - Django Queries

Build a query that:

- Counts the number of devices in a site
- Counts the number of interfaces in a site
- Returns the prefixes that are within another prefix
- Return the devices that have a who's site startswith XXX
- Return the prefixes that do not have any vlan associated with them

CRUD Operations

Create Retrieve Update & Delete

>>> CRUD Operations

Create Retrieve Update & Delete

Create an instance of the class

Created in memory and must be saved to store in database

Create an instance of Django Model via Object Manger

Calling create will return an instance and save to the database

Get or Create an instance in one method

```
obj = Site(name="HOU", slug="hou")
obj.save()
```

```
obj = Site.objects.create(
    name="HOU",
    slug="hou",
    description="Houston, Texas COLO",
    region=<Region Instance>,
    time_zone="America/Chicago"
)
```

```
obj, created = Site.objects.get_or_create(
    name="HOU",
    slug="hou",
    description="Houston, Texas COLO",
    region=<Region Instance>,
    time_zone="America/Chicago"
)
```

>>> CRUD Operations

Create Retrieve Update & Delete

Get one instance of an object

```
obj = Site.objects.get(name="HOU")
```

Retrieve all instances of an object

```
obj = Site.objects.all()
```

Retrieve a subset matching a value

```
obj = Site.objects.filter(time_zone="America/Chicago")
```

Retrieve a subset excluding a value

```
obj = Site.objects.exclude(time_zone="America/Chicago")
```

Retrieve with lookup expressions

```
obj = Site.objects.exclude(
    time_zone__icontains="america"
)
```

>>> CRUD Operations

Create Retrieve Update & Delete

Update a queryset

```
Site.objects.all().update(
    description="Airport Site Codes"
)
```

Must specify *all filter* or *exclude*

```
objs = Site.objects.filter(
    time_zone="America/Chicago"
)
objs.update(
    description="America TZ Sites"
)
```

Calling save is not required

Update a single instance

```
obj = Site.objects.get(name="HOU")
obj.description = "Houston Global DC"
obj.save()
```

Must call *save* to update database

>>> CRUD Operations

Create Retrieve Update & Delete

Delete a queryset

```
Site.objects.all().delete()
```

Must specify *all filter* or *exclude*

```
objs = Site.objects.filter(
    time_zone="America/Chicago"
)
objs.delete()
```

Delete a single instance

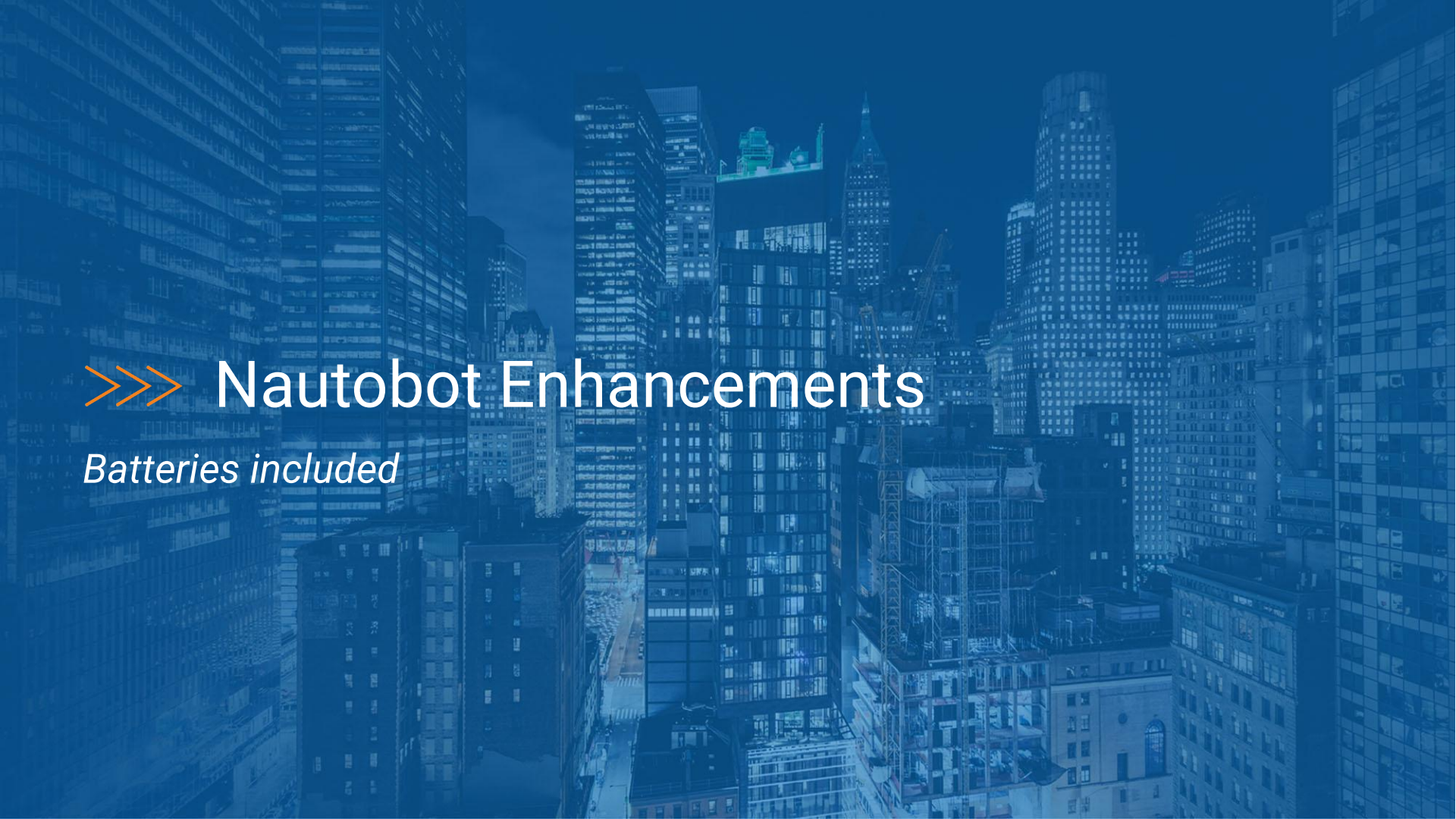
```
obj = Site.objects.get(name="DFW")
obj.delete()
```

Lab 2

>>> Lab 2 - Crud Operations

Build a query that:

- Create a Site object
- Create a Device object
- Update the status for all Arista Devices



Nautobot Enhancements

Batteries included

>>> Nautobot Enhancements

Batteries included

Base abstract models

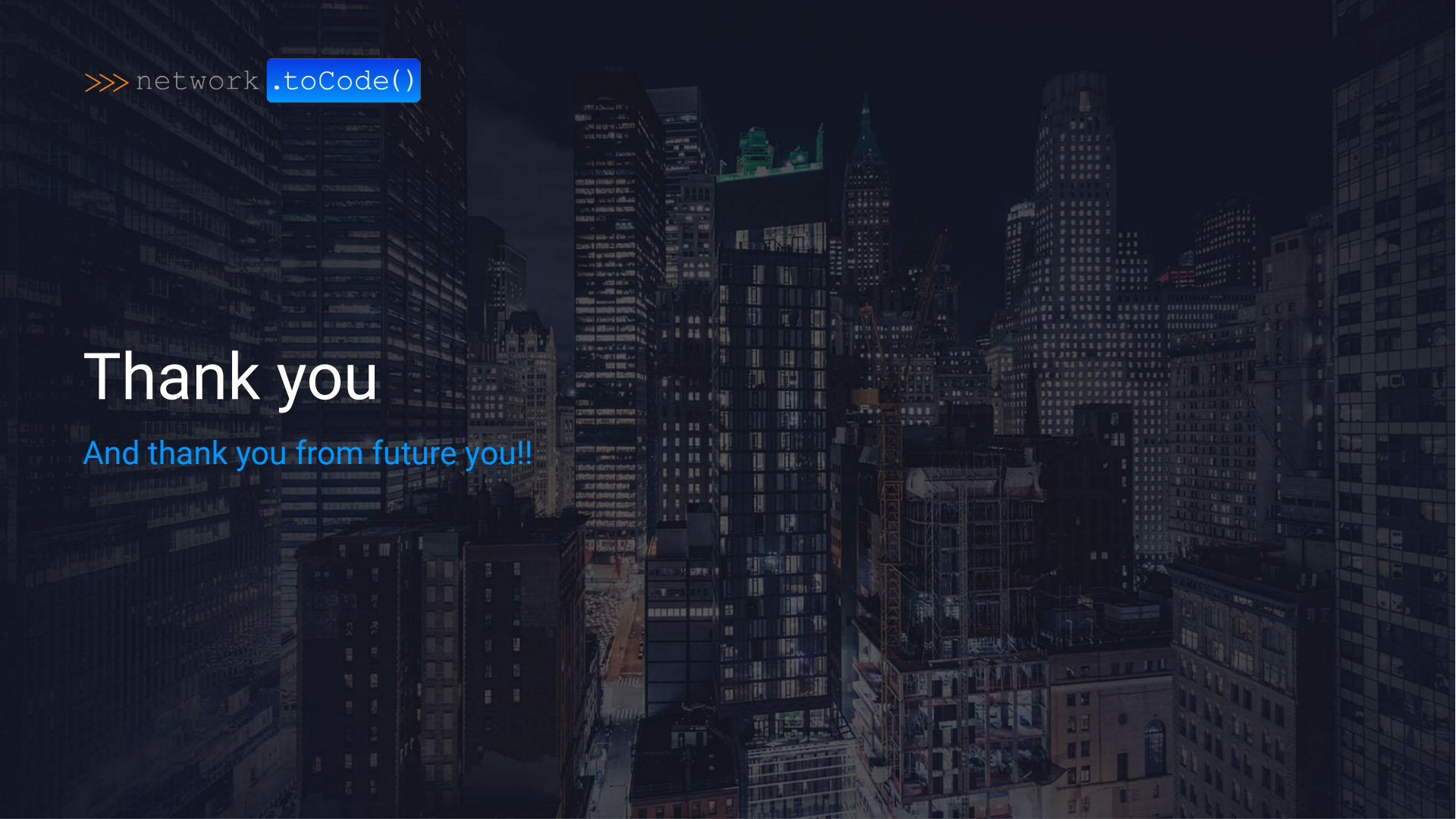
UUID, Tags, Object Change Log, and much more

Validated save, calls full clean

Base set of tools *(filtersets, tables, etc)*

GraphQL with on minimal effort *(extra features & following patterns)*

Namespaced related objects *(Tags, Status, Custom Fields, etc)*


>>> network .toCode()
Thank you
And thank you from future you!!