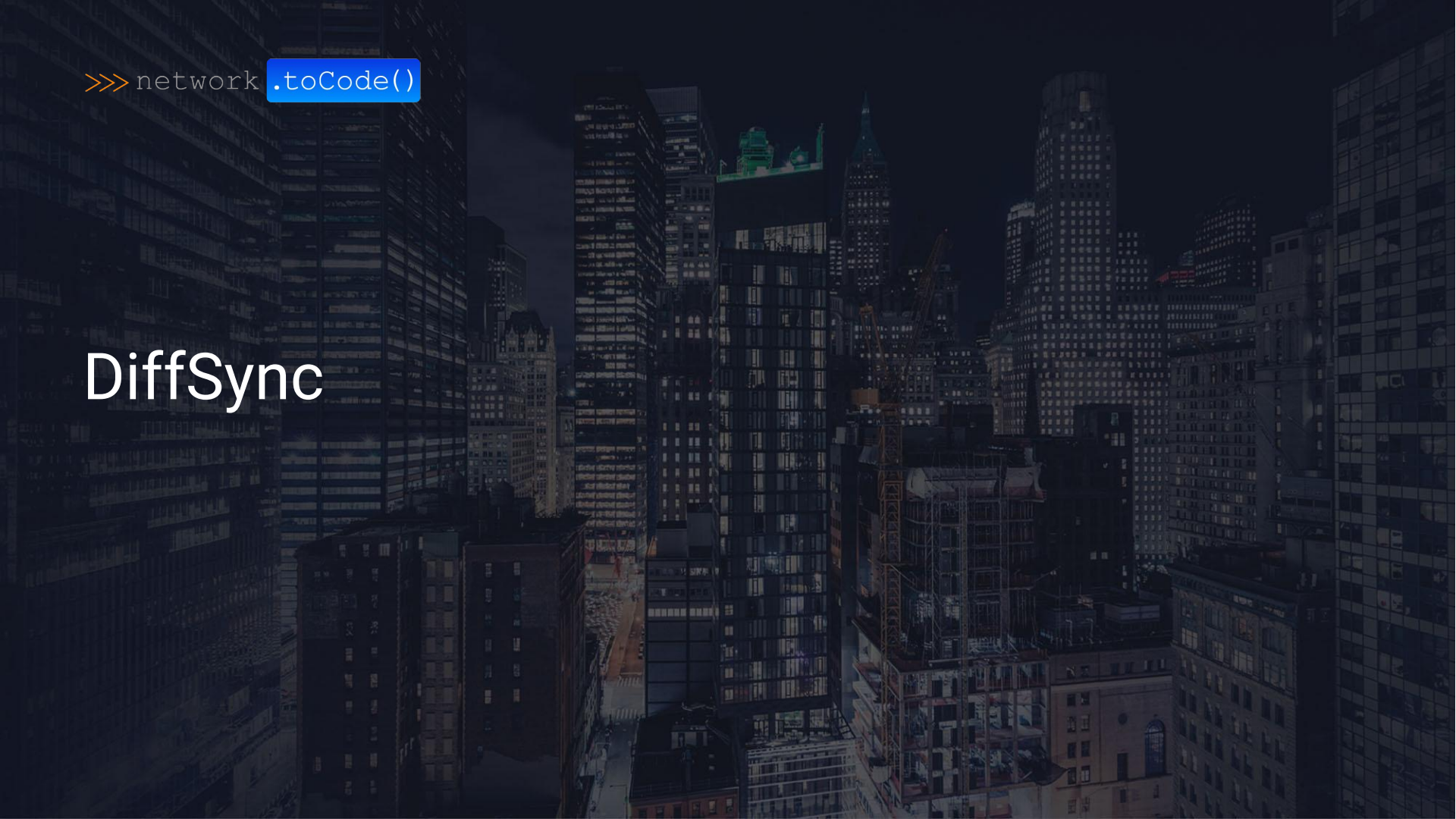

>>> network .toCode()
DiffSync

Agenda

Diffsync Overview

Components

Example (06)

Advanced Example

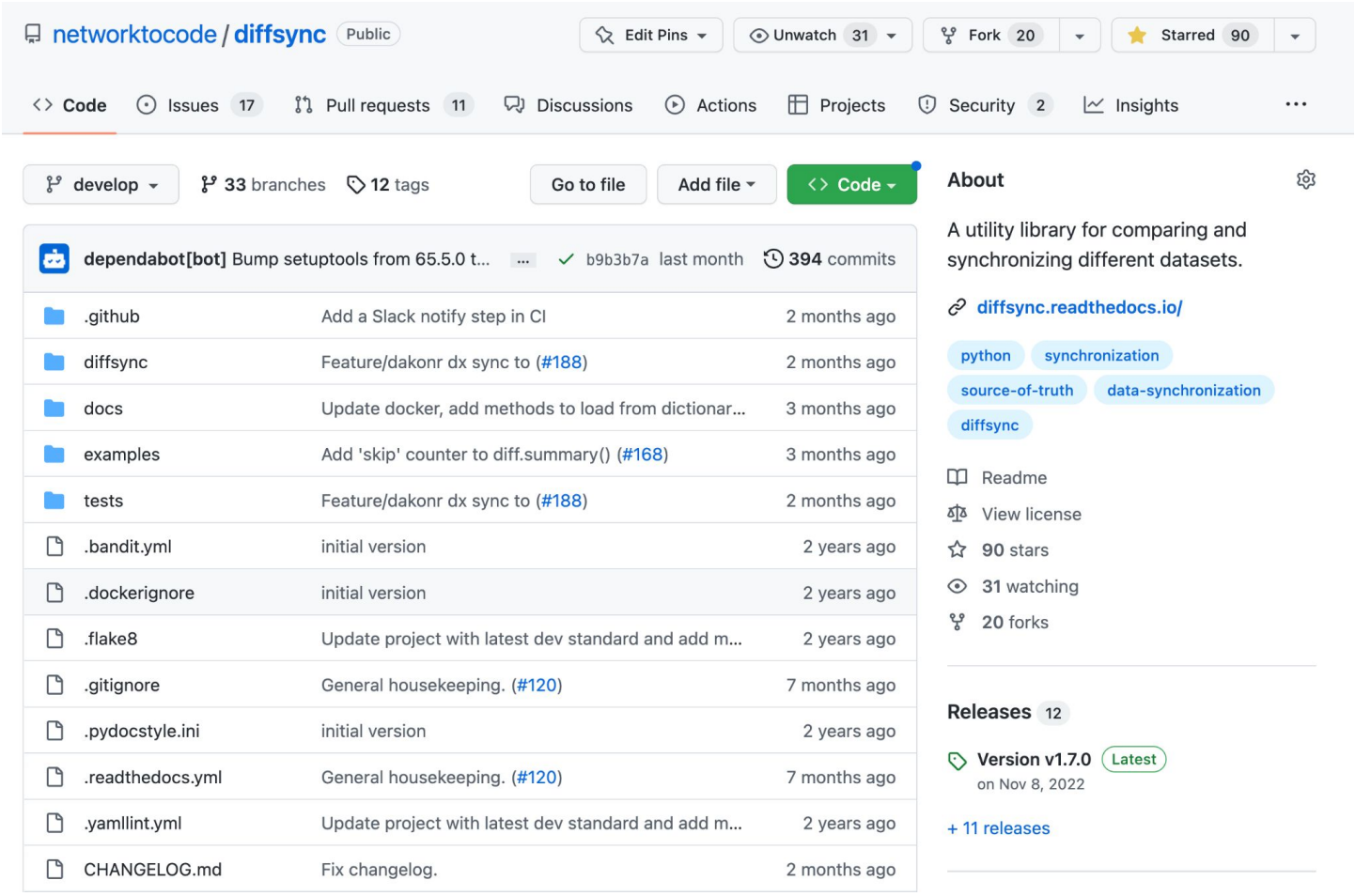
Advanced Components


DiffSync Library Overview
$ pip install diffsync

>>> Problem Statement

- Data necessary for automation is spread across disparate systems.
- Many of these systems are specialized and only contain a small subset of information we need, ie sites in ServiceNow, IP addresses in Infoblox, devices in RackTables, etc.
- Each system has their own methods of interacting, retrieving, and displaying data.
- Having to make calls to multiple systems for data increases complexity and processing time making automation less efficient and desirable.
- How do you keep Nautobot synchronized with those various Systems of Record?
- How do you know when the data is out of sync with those Systems of Record?

DiffSync


networktocode / diffsync
Public
Edit Pins
Unwatch 31
Fork 20
Starred 90
Code
Issues 17
Pull requests 11
Discussions
Actions
Projects
Security 2
Insights
develop
33 branches
12 tags
Go to file
Add file
Code
dependabot[bot] Bump setuptools from 65.5.0 t... b9b3b7a last month 394 commits
.github Add a Slack notify step in CI 2 months ago
diffsync Feature/dakonr dx sync to (#188) 2 months ago
docs Update docker, add methods to load from dictionar... 3 months ago
examples Add 'skip' counter to diff.summary() (#168) 3 months ago
tests Feature/dakonr dx sync to (#188) 2 months ago
.bandit.yml initial version 2 years ago
.dockerignore initial version 2 years ago
.flake8 Update project with latest dev standard and add m... 2 years ago
.gitignore General housekeeping. (#120) 7 months ago
.pydocstyle.ini initial version 2 years ago
.readthedocs.yml General housekeeping. (#120) 7 months ago
.yamllint.yml Update project with latest dev standard and add m... 2 years ago
CHANGELOG.md Fix changelog. 2 months ago
About
A utility library for comparing and synchronizing different datasets.
diffsync.readthedocs.io/
python
synchronization
source-of-truth
data-synchronization
diffsync
Readme
View license
90 stars
31 watching
20 forks
Releases 12
Version v1.7.0 Latest
on Nov 8, 2022
+ 11 releases

>>> How can Diffsync help you?

SUBNET
IPAM A

cidr	192.0.2.0/24
family	4
vrf	vrf-blue
vlan	VLAN123
customer_id	abc

PREFIX
IPAM B

network	192.0.2.0
prefix_length	24
vrf	vrf-blue
vlan_id	123
tenant	abc

>>> How can Diffsync help you?

SUBNET
IPAM A

cidr 192.0.2.0/24
family 4
vrf vrf-blue
vlan VLAN123
customer_id abc

How can we load the data?

What is the difference?

How could we compare vlan name and vlan id?

How can we synchronize the data?

PREFIX
IPAM B

network 192.0.2.0
prefix_length 24
vrf vrf-blue
vlan_id 123
tenant abc

>>> DiffSync, by example

PREFIX DIFFSYNC

prefix	192.0.2.0/24
vrf	vrf-blue
vlan_id	123
tenant	abc

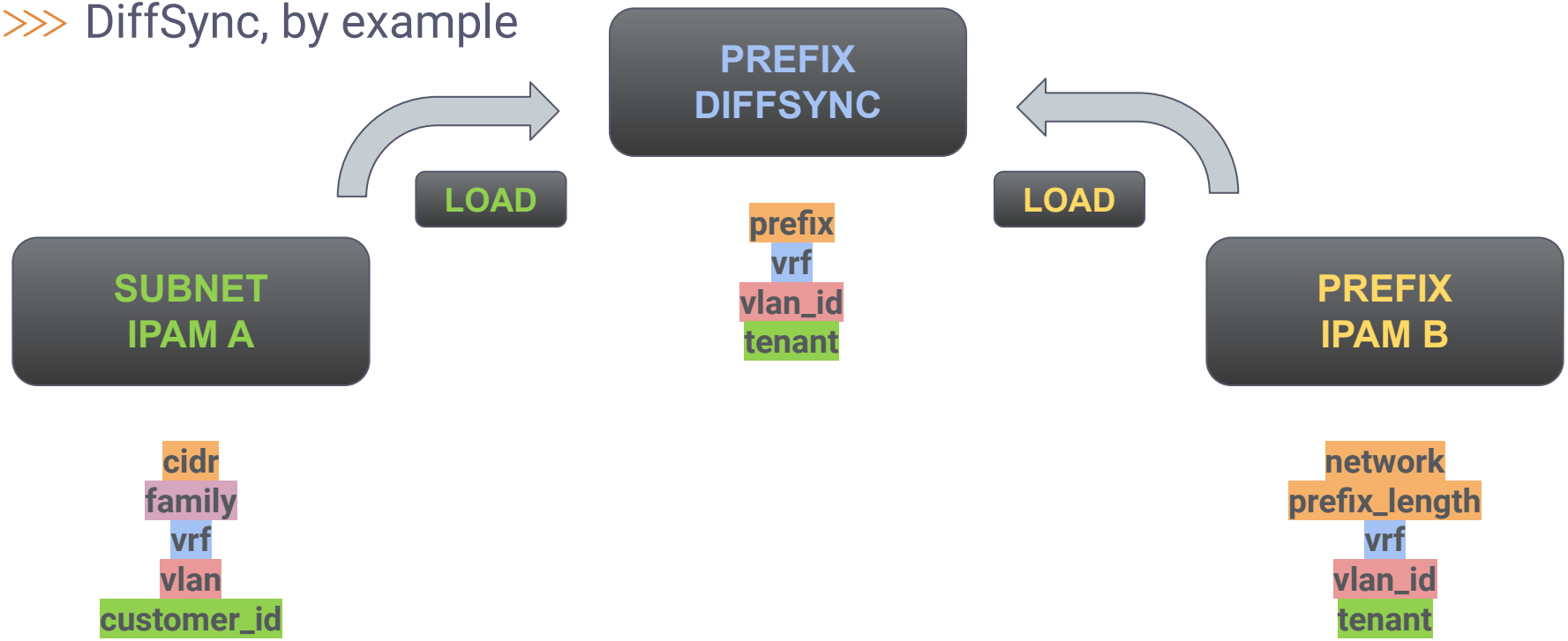
SUBNET IPAM A

cidr	192.0.2.0/24
family	4
vrf	vrf-blue
vlan	VLAN123
customer_id	abc

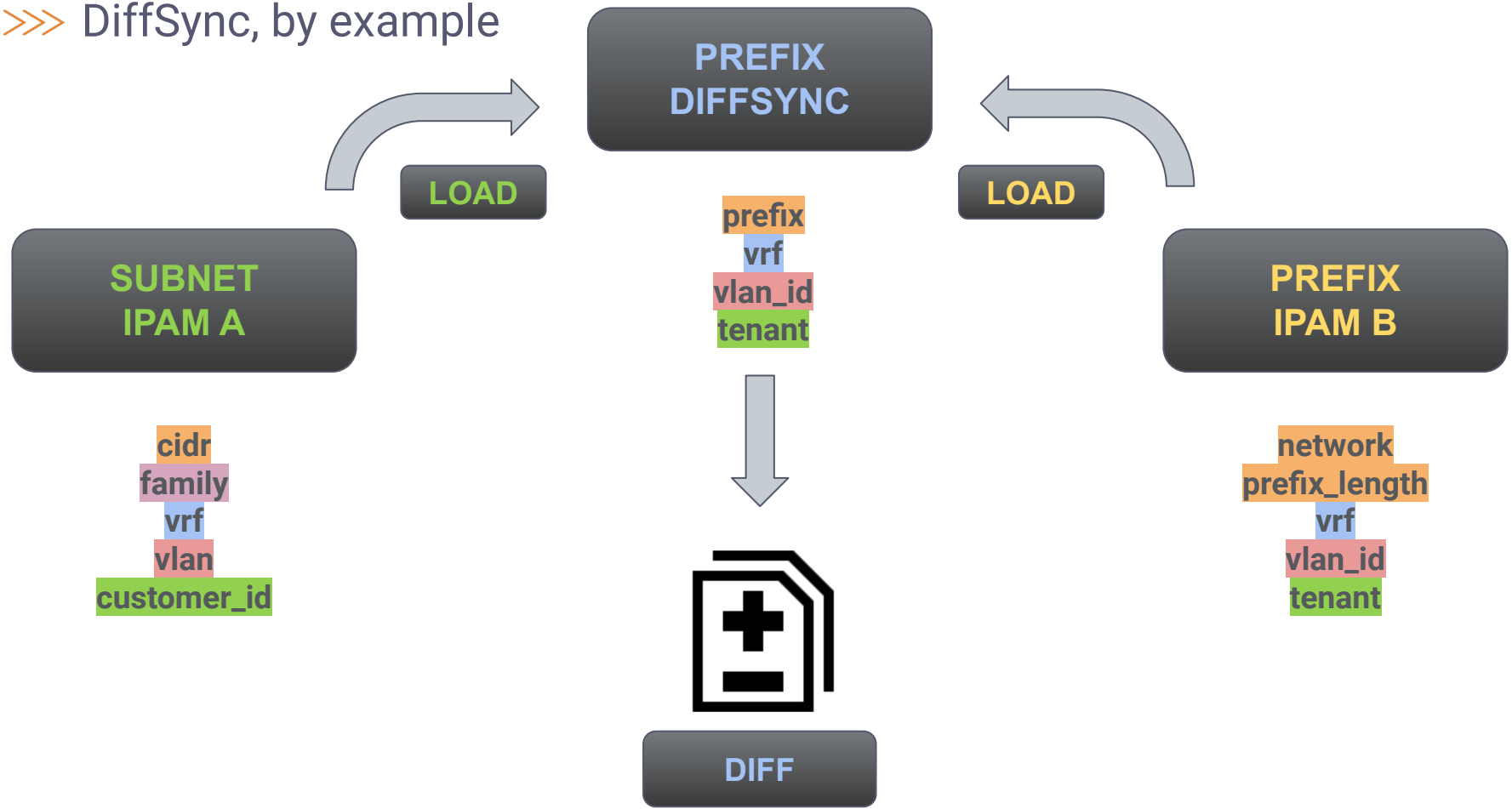
PREFIX IPAM B

network	192.0.2.0
prefix_length	24
vrf	vrf-blue
vlan_id	123
tenant	abc

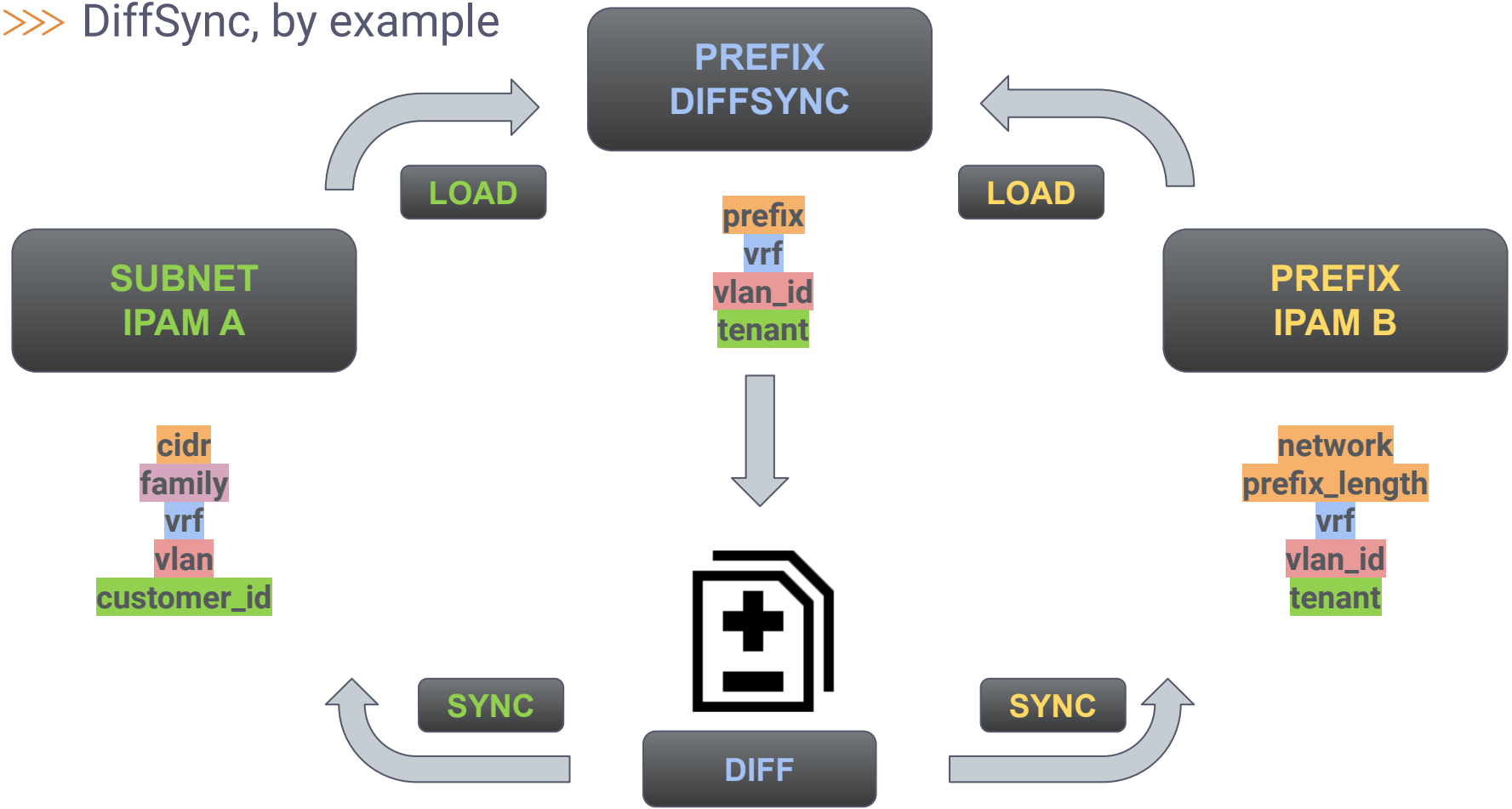
>>> DiffSync, by example



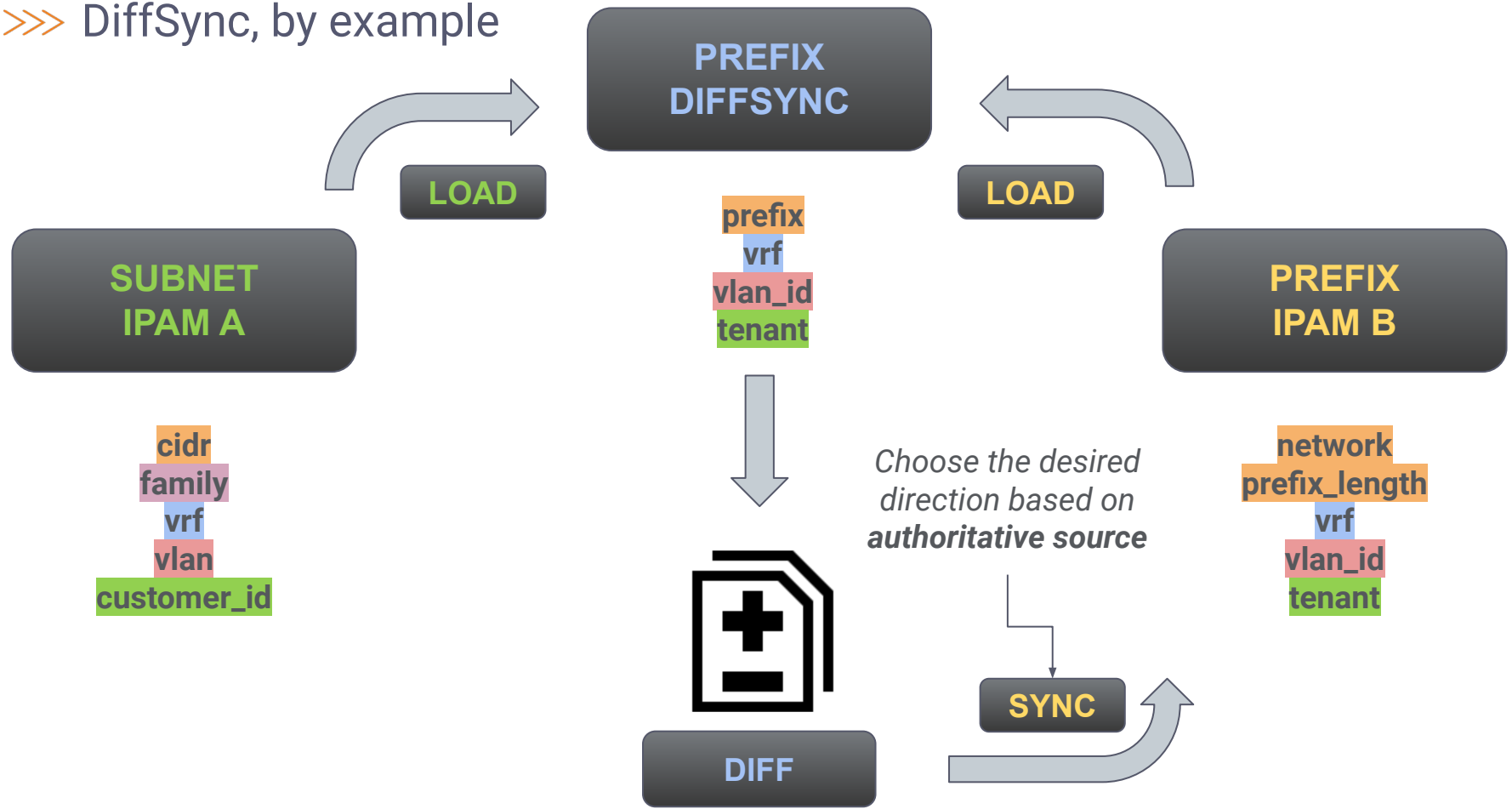
>>> DiffSync, by example



>>> DiffSync, by example



>>> DiffSync, by example



Diffsync Components

“Step by step and the thing is done.” - Charles Atlas

>>> Define DiffSync Model(s)

What do you want to synchronize?

```python
class Room(DiffSyncModel):
    """Device42 Room model."""

    _modelname = "room"
    _identifiers = ("name", "building")
    _shortname = ("name",)
    _attributes = ("notes", "custom_fields")
    _children = {"rack": "racks"}
    name: str
    building: str
    notes: Optional[str]
    racks: List["Rack"] = list()
    custom_fields: Optional[List[dict]]
```

```python
class Rack(DiffSyncModel):
    """Device42 Rack model."""

    _modelname = "rack"
    _identifiers = ("name", "building", "room")
    _shortname = ("name",)
    _attributes = ("height", "numbering_start_from_bottom", "tags", "custom_fields")
    _children = {}
    name: str
    building: str
    room: str
    height: int
    numbering_start_from_bottom: str
    tags: Optional[List[str]]
    custom_fields: Optional[List[dict]]
```

_modelname*: Defines the type of the model; used to identify common models between different systems.
_identifiers*: List of instance field names used as primary keys for this object.
_shortname: List of instance field names to use for a shorter name. Unnecessary if matches _identifiers.
_attributes: List of non-identifier instance field names for this object; used to identify the fields in common between data models for different systems.
_children: Dict of {<model_name>: <field_name>} indicating which fields store references to child data model instances. It is advised to keep these relationships in a linear tree pattern.
NOTE: Attributes are defined in Pydantic syntax. Optional must be used if the attribute can be null. * denotes mandatory fields.

>>> Define DiffSync Model(s)

Define your CRUD methods

```python
@classmethod
def create(cls, diffsync, ids, attrs):
    """Create RackGroup object in Nautobot."""
    new_rg = NautobotRackGroup(
        name=ids["name"],
        slug=slugify(ids["name"]),
        site=NautobotSite.objects.get(name=ids["building"]),
        description=attrs["notes"] if attrs.get("notes") else "",
    )
    if attrs.get("custom_fields"):
        for _cf in attrs["custom_fields"]:
            _cf_dict = {
                "name": slugify(_cf["key"]),
                "type": CustomFieldTypeChoices.TYPE_TEXT,
                "label": _cf["key"],
            }
            field, _ = CustomField.objects.get_or_create(name=slugify(_cf_dict["name"]), defaults=_cf_dict)
            field.content_types.add(ContentType.objects.get_for_model(NautobotRackGroup).id)
            new_rg.custom_field_data.update({_cf_dict["name"]: _cf["value"]})
    new_rg.validated_save()
    return super().create(ids=ids, diffsync=diffsync, attrs=attrs)
```

```python
def update(self, attrs):
    """Update RackGroup object in Nautobot."""
    _rg = NautobotRackGroup.objects.get(name=self.name, site__name=self.building)
    if attrs.get("notes"):
        _rg.description = attrs["notes"]
    if attrs.get("custom_fields"):
        for _cf in attrs["custom_fields"]:
            _cf_dict = {
                "name": slugify(_cf["key"]),
                "type": CustomFieldTypeChoices.TYPE_TEXT,
                "label": _cf["key"],
            }
            field, _ = CustomField.objects.get_or_create(name=slugify(_cf_dict["name"]), defaults=_cf_dict)
            field.content_types.add(ContentType.objects.get_for_model(NautobotRackGroup).id)
            _rg.custom_field_data.update({_cf_dict["name"]: _cf["value"]})
    _rg.validated_save()
    return super().update(attrs)
```

```python
def delete(self):
    """Delete RackGroup object from Nautobot.

    Because RackGroup has a direct relationship to Rack objects it can't be deleted before any Racks.
    The self.diffsync._objects_to_delete dictionary stores all objects for deletion and removes them fr
Nautobot
    in the correct order. This is used in the Nautobot adapter sync_complete function.
    """
    self.diffsync.job.log_warning(f"RackGroup {self.name} will be deleted.")
    super().delete()
    rackgroup = NautobotRackGroup.objects.get(**self.get_identifiers())
    self.diffsync._objects_to_delete["rackgroup"].append(rackgroup)  # pylint: disable=protected-access
    return self
```

```python
def delete(self):
    """Delete IPAddress object from Nautobot."""
    self.diffsync.job.log_warning(f"IP Address {self.address} will be deleted.")
    ipaddr = NautobotIPAddress.objects.get(**self.get_identifiers())
    ipaddr.delete()
    super().delete()
    return self
```

The Create, Update, and Delete methods define how you take the data from the model identifiers and attributes and manipulate the object in the desired DataTarget. Depending upon model dependencies, you might need to be careful handling deletion of some objects. This can be done by using the `sync_complete` method on the DataTarget adapter.

>>> Define DiffSync Adapters

Fill DiffSyncModel(s) with data from DataSource/DataTarget

```python
class Device42Adapter(DiffSync):
    """DiffSync adapter using requests to communicate to Device42 server."""

    building = dcim.Building
    room = dcim.Room
    rack = dcim.Rack
    vendor = dcim.Vendor
    hardware = dcim.Hardware
    cluster = dcim.Cluster
    device = dcim.Device
    port = dcim.Port
    vrf = ipam.VRFGroup
    subnet = ipam.Subnet
    ipaddr = ipam.IPAddress
    vlan = ipam.VLAN
    conn = dcim.Connection
    provider = circuits.Provider
    circuit = circuits.Circuit

    top_level = [
        "building",
        "vendor",
        "hardware",
        "vrf",
        "subnet",
        "vlan",
        "cluster",
        "device",
        "ipaddr",
        "provider",
        "circuit",
        "conn",
    ]
```

```python
class NautobotAdapter(DiffSync):
    """Nautobot adapter for DiffSync."""

    building = dcim.Building
    room = dcim.Room
    rack = dcim.Rack
    vendor = dcim.Vendor
    hardware = dcim.Hardware
    cluster = dcim.Cluster
    device = dcim.Device
    port = dcim.Port
    vrf = ipam.VRFGroup
    subnet = ipam.Subnet
    ipaddr = ipam.IPAddress
    vlan = ipam.VLAN
    conn = dcim.Connection
    provider = circuits.Provider
    circuit = circuits.Circuit

    top_level = [
        "building",
        "vendor",
        "hardware",
        "vrf",
        "subnet",
        "vlan",
        "cluster",
        "device",
        "ipaddr",
        "provider",
        "circuit",
        "conn",
    ]
```

```python
    def load_buildings(self):
        """Load Device42 buildings."""
        for record in self._device42.api_call(path="api/1.0/buildings")["buildings"]:
            if PLUGIN_CFG.get("verbose_debug"):
                self.job.log_info(f"Loading {record['name']} building from Device42.")
            _tags = record["tags"] if record.get("tags") else []
            if len(_tags) > 1:
                _tags.sort()
            building = self.building(
                name=record["name"],
                address=sanitize_string(record["address"]) if record.get("address") else "",
                latitude=round(Decimal(record["latitude"] if record["latitude"] else 0.0), 6),
                longitude=round(Decimal(record["longitude"] if record["longitude"] else 0.0), 6),
                contact_name=record["contact_name"] if record.get("contact_name") else "",
                contact_phone=record["contact_phone"] if record.get("contact_phone") else "",
                rooms=record["rooms"] if record.get("rooms") else [],
                custom_fields=record["custom_fields"],
                tags=_tags,
            )
            _facility = get_facility(diffsync=self, tags=_tags)
            if _facility:
                self.building_map[_facility.upper()] = record["name"]
            try:
                self.add(building)
            except ObjectAlreadyExists as err:
                if PLUGIN_CFG.get("verbose_debug"):
                    self.job.log_warning(f"{record['name']} is already loaded. {err}")
```

The DiffSync adapter classes that you create will pull data from the respective DataSource or DataTarget and fill in the DiffSyncModel(s) that you created in the previous step. The `top_level` list defines the order of the “top level”, ie non-children or models that you wish to be treated as not a child, will be processed by the adapter.

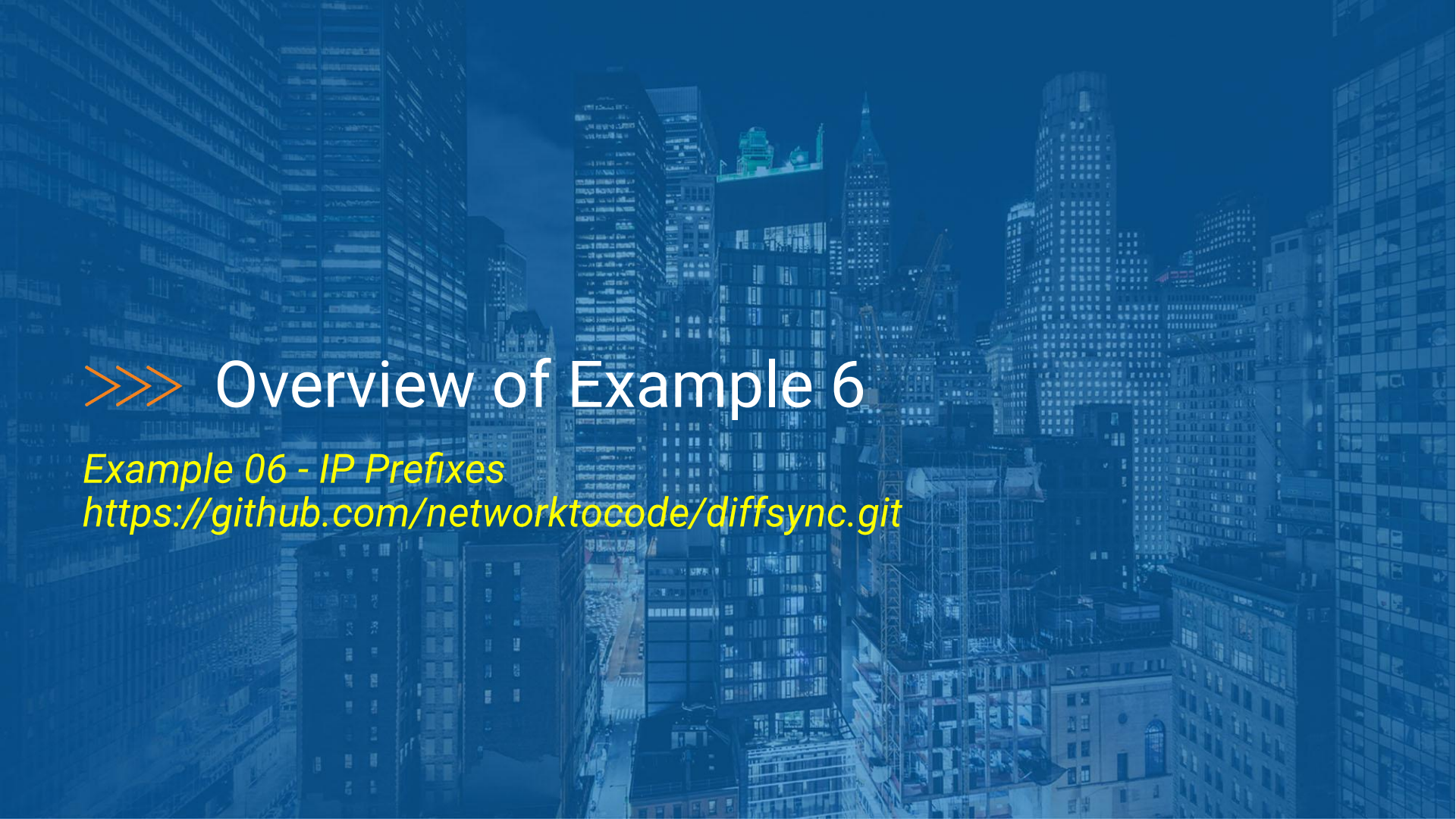
>>> Write your Nautobot Job

Execute the SSoT synchronization

```python
class Device42DataSource(DataSource, Job):
    """Device42 SSoT Data Source."""

    debug = BooleanVar(description="Enable for more verbose debug logging")

    class Meta:
        """Meta data for Device42."""

        name = "Device42"
        data_source = "Device42"
        data_source_icon = static("nautobot_device42_sync/d42_logo.png")
        description = "Sync information from Device42 to Nautobot"

    @classmethod
    def config_information(cls):
        """Dictionary describing the configuration of this DataSource."""
        return {
            "Device42 Host": PLUGIN_CFG.get("device42_host"),
            "Username": PLUGIN_CFG.get("device42_username"),
            "Verify SSL": str(PLUGIN_CFG.get("verify_ssl")),
        }

    @classmethod
    def data_mappings(cls):
        """List describing the data mappings involved in this DataSource."""
        return (
            DataMapping("building", None, "Sites", reverse("dcim:site_list")),
            DataMapping("room", None, "Rack-Groups", reverse("dcim:rackgroup_list")),
            DataMapping("vendor", None, "Manufacturers", reverse("dcim:manufacturer_list")),
            DataMapping("hardware", None, "Device Types", reverse("dcim:devicetype_list")),
            DataMapping("device", None, "Devices", reverse("dcim:device_list")),
        )
```

```python
    def sync_data(self):
        """Device42 Sync."""
        self.log_info(message="Connecting to Device42...")
        d42_adapter = Device42Adapter(job=self, sync=self.sync)
        self.log_info(message="Loading data from Device42...")
        d42_adapter.load()
        self.log_info(message="Connecting to Nautobot...")
        nb_adapter = NautobotAdapter(job=self, sync=self.sync)
        self.log_info(message="Loading data from Nautobot...")
        nb_adapter.load()
        self.log_info(message="Performing diff of data between Device42 and Nautobot.")
        diff = nb_adapter.diff_from(d42_adapter, flags=DiffSyncFlags.CONTINUE_ON_FAILURE)
        self.sync.diff = diff.dict()
        self.sync.save()
        self.log_info(message=diff.summary())
        if not self.kwargs["dry_run"]:
            self.log_info(message="Performing data synchronization from Device42.")
            try:
                nb_adapter.sync_from(d42_adapter, flags=DiffSyncFlags.CONTINUE_ON_FAILURE)
            except HTTPError as err:
                self.log_failure(message="Sync failed.")
                raise err
            except ObjectNotCreated as err:
                self.log_debug(f"Unable to create object. {err}")
            self.log_success(message="Sync complete.")


jobs = [Device42DataSource]
```

The Nautobot Job handles the execution of the DiffSync synchronization. The actual loading of the models and the sync is done in the `sync_data` method. You utilize the `sync_from` or `sync_to` methods on the respective adapter to perform the pushing of data from the DataSource to the DataTarget. You can also utilize the `config_information` and `data_mappings` methods to add information for the user to the Nautobot SSoT GUI.



Overview of Example 6

Example 06 - IP Prefixes
https://github.com/networktocode/diffsync.git

>>> Example files in 06-ip-prefixes

```
$ tree
.
├── README.md           -> Example to follow
├── adapter_ipam_a.py   -> Nautobot DiffSync Adapter for IPAM A
├── adapter_ipam_b.py   -> Nautobot DiffSync Adapter for IPAM B
├── data
│   ├── ipam_a.yml      -> Backend representation of IPAM A
│   └── ipam_b.yml      -> Backend representation of IPAM B
├── main.py
├── models.py           -> DiffSync Prefix model
└── requirements.txt    -> Python dependencies to run
```

>>> Data sources

data/ipam_a.yml

```yaml
---
- cidr: "10.10.10.10/24"
  family: 4
  vrf: "data"
  vlan: "VLAN10"
  customer_id: null
- cidr: "10.20.20.20/24"
  family: 4
  vrf: "voice"
  vlan: "VLAN20"
  customer_id: "ABC corp"
- cidr: "172.18.0.0/16"
  family: 4
  vrf: null
  vlan: "VLAN18"
  customer_id: null
```

Different schemas

data/ipam_b.yml

```yaml
---
- network: "10.10.10.10"
  prefix_length: 24
  tenant: null
  vlan_id: 123
  vrf: "voice"
- network: "2001:DB8::"
  prefix_length: 32
  tenant: "XYZ Corporation"
  vlan_id: 10
  vrf: "data"
```

>>> Up and Running with Diffsync

1 - Git Clone DiffSync from Github

```
$ git clone
https://github.com/networktocode/diffsync.git
```

2 - Set up environment

```
$ cd diffsync/examples/06-ip-prefixes
$ python3 -m venv my-virtual-env
$ source my-virtual-env/bin/activate
$ pip3 install -r requirements.txt
```

3 - Enter Python interpreter mode

```
$ python
>>>
```

>>> Let's Try it Out!

1 - Import DiffSync Adapters

```
>>> from adapter_ipam_a import IpamA
>>> from adapter_ipam_b import IpamB
```

2 - Initialize and load IPAM B

```
>>> ipam_b = IpamB()
>>> ipam_b.load()
```

>>> Let's Try it Out!

1 - Import DiffSync Adapters

```
>>> from adapter_ipam_a import IpamA
>>> from adapter_ipam_b import IpamB
```

2 - Initialize and load IPAM B

```
>>> ipam_b = IpamB()
>>> ipam_b.load()
```

3 - Check the loaded data

```
>>> ipam_b.dict()
{
    'prefix': {
        '10.10.10.10/24': {
            'prefix': '10.10.10.10/24',
            'vrf': 'voice',
            'vlan_id': 123
        },
        '2001:DB8::/32': {
            'prefix': '2001:DB8::/32',
            'vrf': 'data',
            'vlan_id': 10,
            'tenant': 'XYZ Corporation'
        }
    }
}
```

>>> Let's Try it Out!

1 - Import DiffSync Adapters

```
>>> from adapter_ipam_a import IpamA
>>> from adapter_ipam_b import IpamB
```

2 - Initialize and load IPAM B

```
>>> ipam_b = IpamB()
>>> ipam_b.load()
```

3 - Check the loaded data

```
>>> ipam_b.dict()
{
    'prefix': {
        '10.10.10.10/24': {
            'prefix': '10.10.10.10/24',
            'vrf': 'voice',
            'vlan_id': 123
        },
        '2001:DB8::/32': {
            'prefix': '2001:DB8::/32',
            'vrf': 'data',
            'vlan_id': 10,
            'tenant': 'XYZ Corporation'
        }
    }
}
```

data/ipam_b.yml

```yaml
---
- network: "10.10.10.10"
  prefix_length: 24
  tenant: null
  vlan_id: 123
  vrf: "voice"
- network: "2001:DB8::"
  prefix_length: 32
  tenant: "XYZ Corporation"
  vlan_id: 10
  vrf: "data"
```

>>> Let's Try it Out!

1 - Import DiffSync Adapters

```
>>> from adapter_ipam_a import IpamA
>>> from adapter_ipam_b import IpamB
```

2 - Initialize and load IPAM B

```
>>> ipam_b = IpamB()
>>> ipam_b.load()
```

3 - Check the loaded data

```
>>> ipam_b.dict()
{
    'prefix': {
        '10.10.10.10/24': {
            'prefix': '10.10.10.10/24',
            'vrf': 'voice',
            'vlan_id': 123
        },
        '2001:DB8::/32': {
            'prefix': '2001:DB8::/32',
            'vrf': 'data',
            'vlan_id': 10,
            'tenant': 'XYZ Corporation'
        }
    }
}
```

data/ipam_b.yml

```yaml
---
- network: "10.10.10.10"
  prefix_length: 24
  tenant: null
  vlan_id: 123
  vrf: "voice"
- network: "2001:DB8::"
  prefix_length: 32
  tenant: "XYZ Corporation"
  vlan_id: 10
  vrf: "data"
```

>>> Let's Try it Out!

4 - Initialize and load IPAM A

5 - Check the loaded data

```
>>> ipam_a = IpamA()
>>> ipam_a.load()

>>> ipam_a.dict()
{
    'prefix': {
        '10.10.10.10/24': {
            'prefix': '10.10.10.10/24',
            'vrf': 'data',
            'vlan_id': 10
        },
        '10.20.20.20/24': {
            'prefix': '10.20.20.20/24',
            'vrf': 'voice',
            'vlan_id': 20,
            'tenant': 'ABC corp'
        },
        '172.18.0.0/16': {
            'prefix': '172.18.0.0/16',
            'vlan_id': 18
        }
    }
}
```

>>> Let's Try it Out!

4 - Initialize and load IPAM A

5 - Check the loaded data

```
>>> ipam_a = IpamA()
>>> ipam_a.load()

>>> ipam_a.dict()
{
    'prefix': {
        '10.10.10.10/24': {
            'prefix': '10.10.10.10/24',
            'vrf': 'data',
            'vlan_id': 10
        },
        '10.20.20.20/24': {
            'prefix': '10.20.20.20/24',
            'vrf': 'voice',
            'vlan_id': 20,
            'tenant': 'ABC corp'
        },
        '172.18.0.0/16': {
            'prefix': '172.18.0.0/16',
            'vlan_id': 18
        }
    }
}
```

data/ipam_a.yml

```yaml
---
- cidr: "10.10.10.10/24"
  family: 4
  vrf: "data"
  vlan: "VLAN10"
  customer_id: null
- cidr: "10.20.20.20/24"
  family: 4
  vrf: "voice"
  vlan: "VLAN20"
  customer_id: "ABC corp"
- cidr: "172.18.0.0/16"
  family: 4
  vrf: null
  vlan: "VLAN18"
  customer_id: null
```

>>> Let's Try it Out!

4 - Initialize and load IPAM A

5 - Check the loaded data

```
>>> ipam_a = IpamA()
>>> ipam_a.load()

>>> ipam_a.dict()
{
    'prefix': {
        '10.10.10.10/24': {
            'prefix': '10.10.10.10/24',
            'vrf': 'data',
            'vlan_id': 10
        },
        '10.20.20.20/24': {
            'prefix': '10.20.20.20/24',
            'vrf': 'voice',
            'vlan_id': 20,
            'tenant': 'ABC corp'
        },
        '172.18.0.0/16': {
            'prefix': '172.18.0.0/16',
            'vlan_id': 18
        }
    }
}
```

data/ipam_a.yml

```
---
- cidr: "10.10.10.10/24"
  family: 4
  vrf: "data"
  vlan: "VLAN10"
  customer_id: null
- cidr: "10.20.20.20/24"
  family: 4
  vrf: "voice"
  vlan: "VLAN20"
  customer_id: "ABC corp"
- cidr: "172.18.0.0/16"
  family: 4
  vrf: null
  vlan: "VLAN18"
  customer_id: null
```

>>> Let's Try it Out!

6 - Let's diff

```
>>> diff = ipam_a.diff_from(ipam_b)
>>> diff.summary()
{'create': 2, 'update': 1, 'delete': 1, 'no-change': 0}
```

>>> Let's Try it Out!

6 - Let's diff

```
>>> diff = ipam_a.diff_from(ipam_b)
>>> diff.summary()
{'create': 2, 'update': 1, 'delete': 1, 'no-change': 0}
```

7 - Check the Diff

```
>>> diff.dict()
{
    'prefix': {
        '10.10.10.10/24': {
            '-': {
                'vrf': 'voice',
                'vlan_id': 123},
            '+': {
                'vrf': 'data',
                'vlan_id': 10
            }
         },
         '10.20.20.20/24': {
            '+': {'vrf': 'voice', 'vlan_id': 20, 'tenant': 'ABC corp'}
         },
         '172.18.0.0/16': {
            '+': {'vrf': None, 'vlan_id': 18, 'tenant': None}
        },
        '2001:DB8::/32': {
            '-': {'vrf': 'data', 'vlan_id': 10, 'tenant': 'XYZ Corporation'}
        }
    }
}
```

>>> Let's Try it Out!

6 - Let's diff

7 - Check the Diff

```
>>> diff = ipam_a.diff_from(ipam_b)
>>> diff.summary()
{'create': 2, 'update': 1, 'delete': 1, 'no-change': 0}


>>> diff.dict()
{
    'prefix': {
        10.10.10.10/24': {
            '-': {
                'vrf': 'voice',
                'vlan_id': 123},
            '+': {
                'vrf': 'data',
                'vlan_id': 10
            }
        },
        '10.20.20.20/24': {
            '+': {'vrf': 'voice', 'vlan_id': 20, 'tenant': 'ABC corp'}
        },
        '172.18.0.0/16': {
            '+': {'vrf': None, 'vlan_id': 18, 'tenant': None}
        },
        '2001:DB8::/32': {
            '-': {'vrf': 'data', 'vlan_id': 10, 'tenant': 'XYZ Corporation'}
        }
    }
}
```

>>> Let's Try it Out!

8 - Check the Diff, with flags

```
>>> from diffsync.enum import DiffSyncFlags
>>> diff = ipam_a.diff_from(
...     ipam_b,
...     flags=DiffSyncFlags.SKIP_UNMATCHED_DST
... )
>>> diff.dict()
{
    'prefix': {
        '10.10.10.10/24': {
            '-': {
                'vrf': 'voice',
                'vlan_id': 123},
            '+': {
                'vrf': 'data',
                'vlan_id': 10
            }
         },
         '10.20.20.20/24': {
            '+': {'vrf': 'voice', 'vlan_id': 20, 'tenant': 'ABC corp'}
         },
         '172.18.0.0/16': {
            '+': {'vrf': None, 'vlan_id': 18, 'tenant': None}
        },
    }
}
```

>>> Let's Try it Out!

8 - Check the Diff, with flags

```
>>> from diffsync.enum import DiffSyncFlags
>>> diff = ipam_a.diff_from(
...     ipam_b,
...     flags=DiffSyncFlags.SKIP_UNMATCHED_DST
... )
>>> diff.dict()
{
    'prefix': {
        '10.10.10.10/24': {
            '-': {
                'vrf': 'voice',
                'vlan_id': 123},
            '+': {
                'vrf': 'data',
                'vlan_id': 10
            }
        },
        '10.20.20.20/24': {
            '+': {'vrf': 'voice', 'vlan_id': 20, 'tenant': 'ABC corp'}
        },
        '172.18.0.0/16': {
            '+': {'vrf': None, 'vlan_id': 18, 'tenant': None}
        },
    }
}
```

Supported Global Flags

Name
CONTINUE_ON_FAILURE
SKIP_UNMATCHED_SRC
SKIP_UNMATCHED_DST
SKIP_UNMATCHED_BOTH
LOG_UNCHANGED_RECORDS

```
'2001:DB8::/32': {
    '-': {'vrf': 'data', 'vlan_id': 10, 'tenant': 'XYZ Corporation'}
}
```

>>> Let's Try it Out!

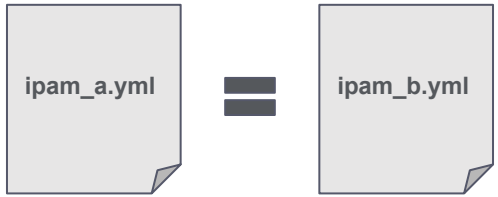
9 - Enforce synchronization

```
>>> ipam_a.sync_to(
...     ipam_b,
...     flags=DiffSyncFlags.SKIP_UNMATCHED_DST
... )
```

>>> Let's Try it Out!

9 - Enforce synchronization

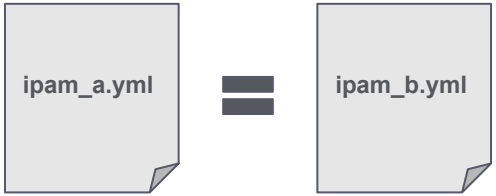
```
>>> ipam_a.sync_to(
...     ipam_b,
...     flags=DiffSyncFlags.SKIP_UNMATCHED_DST
... )
```



>>> Let's Try it Out!

9 - Enforce synchronization

```
>>> ipam_a.sync_to(
...     ipam_b,
...     flags=DiffSyncFlags.SKIP_UNMATCHED_DST
... )
```



10 - Let's confirm it's in sync

```
>>> new_ipam_b = IpamB()
>>> new_ipam_b.load()
>>> diff = ipam_a.diff_to(new_ipam_b)
>>> diff.summary()
{'create': 0, 'update': 0, 'delete': 0, 'no-change': 3}
```

>>> DiffSyncModel

```python
class Prefix(DiffSyncModel):
    """Example model of a Prefix."""

    _modelname = "prefix"
    _identifiers = ("prefix",)
    _attributes = ("vrf", "vlan_id", "tenant")


    prefix: str
    vrf: Optional[str]
    vlan_id: Optional[int]
    tenant: Optional[str]
```

>>> DiffSyncModel

```
class DiffSyncModel(pydantic.BaseModel)
```

```
class Prefix(DiffSyncModel):
    """Example model of a Prefix."""

    _modelname = "prefix"
    _identifiers = ("prefix",)
    _attributes = ("vrf", "vlan_id", "tenant")



    prefix: str
    vrf: Optional[str]
    vlan_id: Optional[int]
    tenant: Optional[str]
```

>>> DiffSyncModel

DiffSyncModel metadata

- ***_modelname***: String, type of the model
- ***_identifiers***: List of instance field names used as primary keys
- ***_attributes***: List of non-identifier instance field names, used to identify the fields in common

```
class DiffSyncModel(pydantic.BaseModel)
```

```
class Prefix(DiffSyncModel):
    """Example model of a Prefix."""

    _modelname = "prefix"
    _identifiers = ("prefix",)
    _attributes = ("vrf", "vlan_id", "tenant")
```

```
    prefix: str
    vrf: Optional[str]
    vlan_id: Optional[int]
    tenant: Optional[str]
```

>>> DiffSyncModel

DiffSyncModel metadata

- ***_modelname***: String, type of the model
- ***_identifiers***: List of instance field names used as primary keys
- ***_attributes***: List of non-identifier instance field names, used to identify the fields in common

```
class DiffSyncModel(pydantic.BaseModel)
```

```
class Prefix(DiffSyncModel):
    """Example model of a Prefix."""

    _modelname = "prefix"
    _identifiers = ("prefix",)
    _attributes = ("vrf", "vlan_id", "tenant")
```

Attributes included in the model comparison process, as _attributes or _children

```
    prefix: str
    vrf: Optional[str]
    vlan_id: Optional[int]
    tenant: Optional[str]
```

>>> DiffSyncModel

```
class DiffSyncModel(pydantic.BaseModel)
```

DiffSyncModel metadata

- ***_modelname***: String, type of the model
- ***_identifiers***: List of instance field names used as primary keys
- ***_attributes***: List of non-identifier instance field names, used to identify the fields in common

```
class Prefix(DiffSyncModel):
    """Example model of a Prefix."""

    _modelname = "prefix"
    _identifiers = ("prefix",)
    _attributes = ("vrf", "vlan_id", "tenant")
```

Attributes included in the model comparison process, as _attributes or _children

```
    prefix: str
    vrf: Optional[str]
    vlan_id: Optional[int]
    tenant: Optional[str]
```

There is the option to create model relationships with **_childrens**

>>> Loading data from Nautobot

Use `data` from the YAML file, but could be via API or ORM, for example

Instantiate a diffsync object **prefix** and add it to the store. Doing this we could add any custom manipulation needed for the later comparison (convert from VLAN name to identifier)

```
class IpamA(DiffSync):
    def load(self):
        """Load prefixes from IPAM A."""
        for subnet in self.data:
            prefix = self.prefix(
                prefix=subnet["cidr"],
                vrf=subnet["vrf"],
                vlan_id=int(
                    subnet["vlan"].lstrip("VLAN")
                ),
                tenant=subnet["customer_id"],
            )
            self.add(prefix)
```

>>> Loading data from YAML

```python
class IpamA(DiffSync):
    def load(self):
        """Load prefixes from IPAM A."""
        for subnet in self.data:
            prefix = self.prefix(
                prefix=subnet["cidr"],
                vrf=subnet["vrf"],
                vlan_id=int(
                    subnet["vlan"].lstrip("VLAN")
                ),
                tenant=subnet["customer_id"],
            )
            self.add(prefix)
```

>>> Loading data from YAML

Use `data` from the YAML file but could be via API or ORM, for example

```
class IpamA(DiffSync):
    def load(self):
        """Load prefixes from IPAM A."""
        for subnet in self.data:
            prefix = self.prefix(
                prefix=subnet["cidr"],
                vrf=subnet["vrf"],
                vlan_id=int(
                    subnet["vlan"].lstrip("VLAN")
                ),
                tenant=subnet["customer_id"],
            )
            self.add(prefix)
```

>>> Loading data from YAML

Use `data` from the YAML file but could be via API or ORM, for example

Instantiate a diffsync object **prefix** and add it to the store. Doing this we could add any custom manipulation needed for the later comparison (convert from VLAN name to identifier)

```python
class IpamA(DiffSync):
    def load(self):
        """Load prefixes from IPAM A."""
        for subnet in self.data:
            prefix = self.prefix(
                prefix=subnet["cidr"],
                vrf=subnet["vrf"],
                vlan_id=int(
                    subnet["vlan"].lstrip("VLAN")
                ),
                tenant=subnet["customer_id"],
            )
            self.add(prefix)
```

>>> *DiffSyncModel* CRUD operations

Define the create operation in the IPAM A Adapter. How would we create objects when required

Transform data from ids and attrs to use to create the remote object data

```
class Prefix(DiffSyncModel)

class IpamAPrefix(Prefix):

    @classmethod
    def create(cls, diffsync, ids, attrs):
        """Create a Prefix record in IPAM A."""
        diffsync.data.append(
            {
                "cidr": ids["prefix"],
                "family": ipaddress.ip_address(
                    ids["prefix"].split("/")[0]
                ).version,
                "vrf": attrs["vrf"],
                "vlan": f'VLAN{attrs["vlan_id"]}',
                "customer_id": (
                    attrs["tenant"]
                    if attrs["tenant"]
                    else None,
                )
            }
        )

        return super().create(diffsync, ids=ids, attrs=attrs)
```

>>> *DiffSyncModel* CRUD operations

```python
class IpamAPrefix(Prefix):

    @classmethod
    def create(cls, diffsync, ids, attrs):
        """Create a Prefix record in IPAM A."""
        diffsync.data.append(
            {
                "cidr": ids["prefix"],
                "family": ipaddress.ip_address(
                    ids["prefix"].split("/")[0]
                ).version,
                "vrf": attrs["vrf"],
                "vlan": f'VLAN{attrs["vlan_id"]}',
                "customer_id": (
                    attrs["tenant"]
                    if attrs["tenant"]
                    else None,
                )
            }
        )

        return super().create(diffsync, ids=ids, attrs=attrs)
```

>>> *DiffSyncModel* CRUD operations

```
class Prefix(DiffSyncModel)

class IpamAPrefix(Prefix):

    @classmethod
    def create(cls, diffsync, ids, attrs):
        """Create a Prefix record in IPAM A."""
        diffsync.data.append(
            {
                "cidr": ids["prefix"],
                "family": ipaddress.ip_address(
                    ids["prefix"].split("/")[0]
                ).version,
                "vrf": attrs["vrf"],
                "vlan": f'VLAN{attrs["vlan_id"]}',
                "customer_id": (
                    attrs["tenant"]
                    if attrs["tenant"]
                    else None,
                )
            }
        )

        return super().create(diffsync, ids=ids, attrs=attrs)
```

>>> *DiffSyncModel* CRUD operations

Define the create operation in the IPAM A Adapter. How would we create objects when required

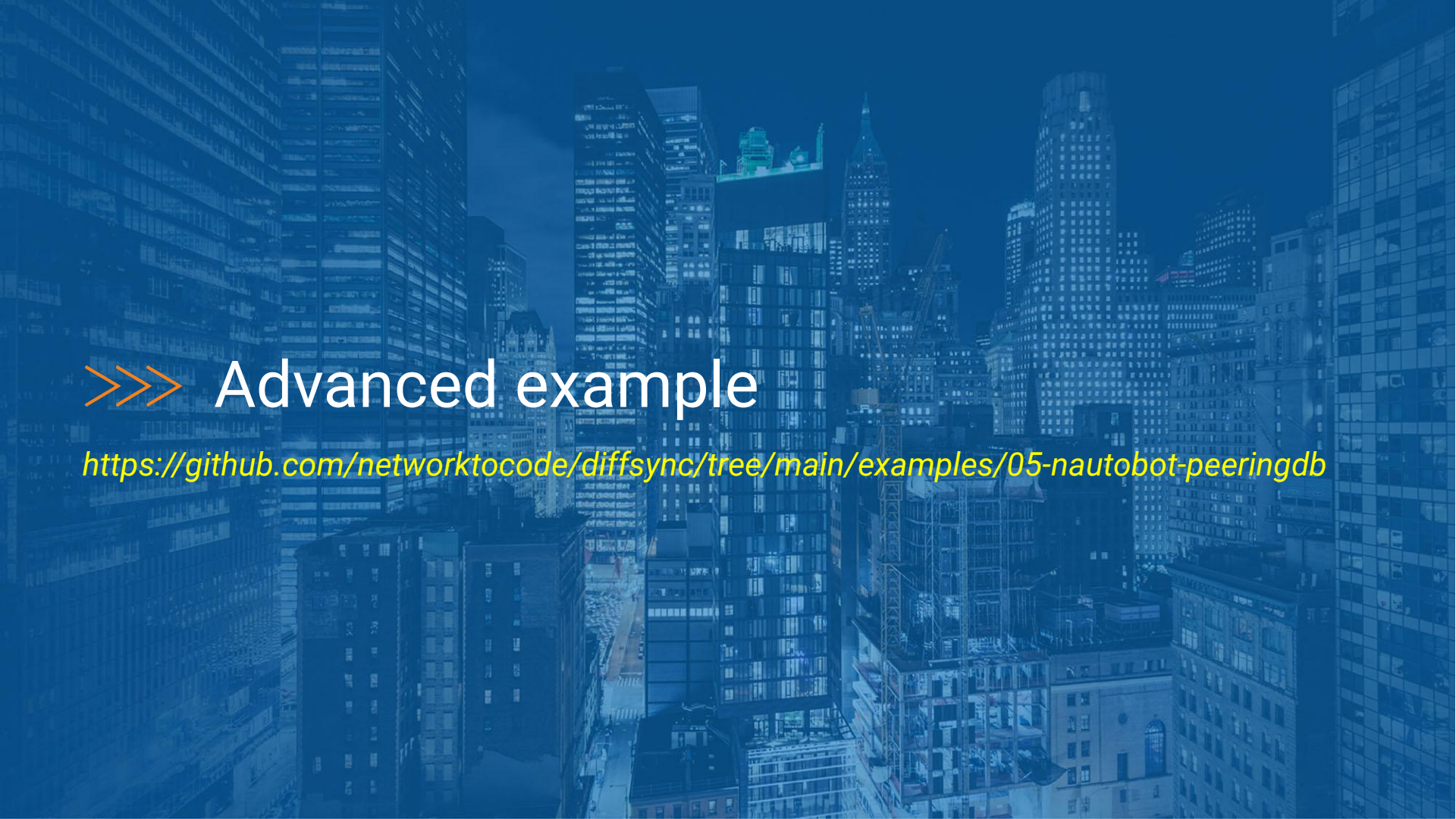
```
class Prefix(DiffSyncModel)

class IpamAPrefix(Prefix):

    @classmethod
    def create(cls, diffsync, ids, attrs):
        """Create a Prefix record in IPAM A."""
        diffsync.data.append(
            {
                "cidr": ids["prefix"],
                "family": ipaddress.ip_address(
                    ids["prefix"].split("/")[0]
                ).version,
                "vrf": attrs["vrf"],
                "vlan": f'VLAN{attrs["vlan_id"]}',
                "customer_id": (
                    attrs["tenant"]
                    if attrs["tenant"]
                    else None,
                )
            }
        )

        return super().create(diffsync, ids=ids, attrs=attrs)
```

>>> *DiffSyncModel* CRUD operations

Define the create operation in the IPAM A Adapter. How would we create objects when required

Transform data from ids and attrs to use to create the remote object data

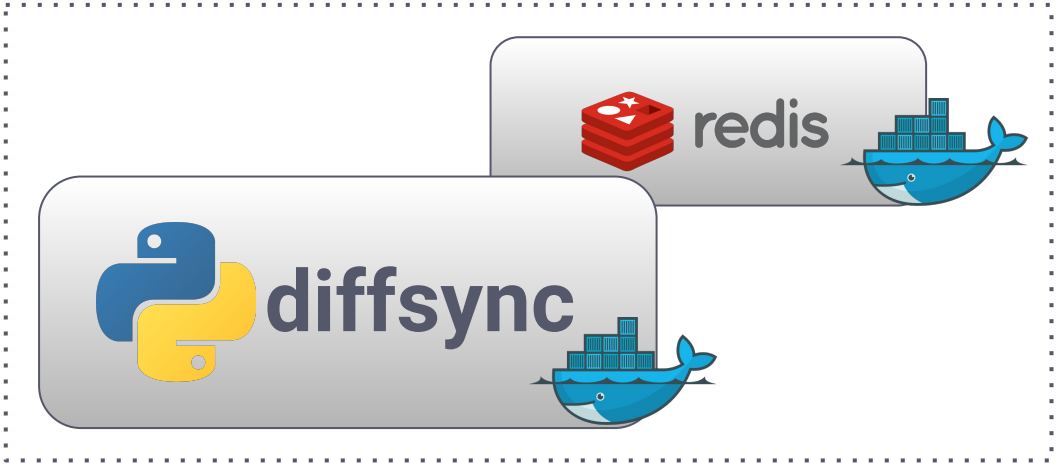
```
class Prefix(DiffSyncModel)

class IpamAPrefix(Prefix):

    @classmethod
    def create(cls, diffsync, ids, attrs):
        """Create a Prefix record in IPAM A."""
        diffsync.data.append(
            {
                "cidr": ids["prefix"],
                "family": ipaddress.ip_address(
                    ids["prefix"].split("/")[0]
                ).version,
                "vrf": attrs["vrf"],
                "vlan": f'VLAN{attrs["vlan_id"]}',
                "customer_id": (
                    attrs["tenant"]
                    if attrs["tenant"]
                    else None,
                )
            }
        )

        return super().create(diffsync, ids=ids, attrs=attrs)
```

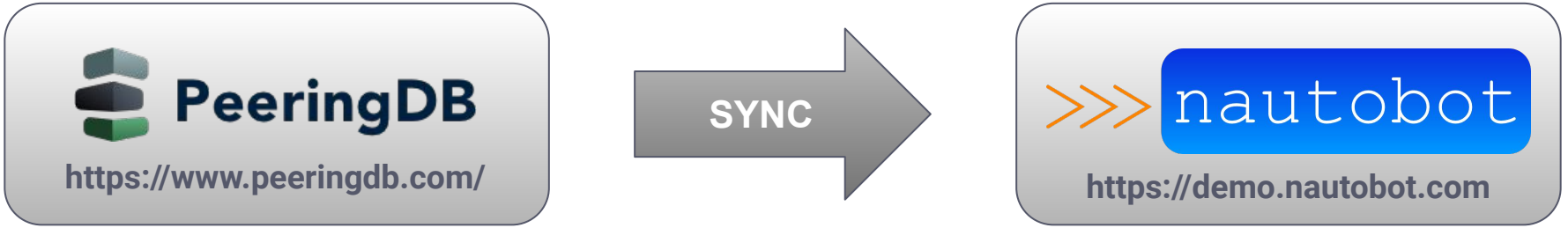


Advanced example

https://github.com/networktocode/diffsync/tree/main/examples/05-nautobot-peeringdb

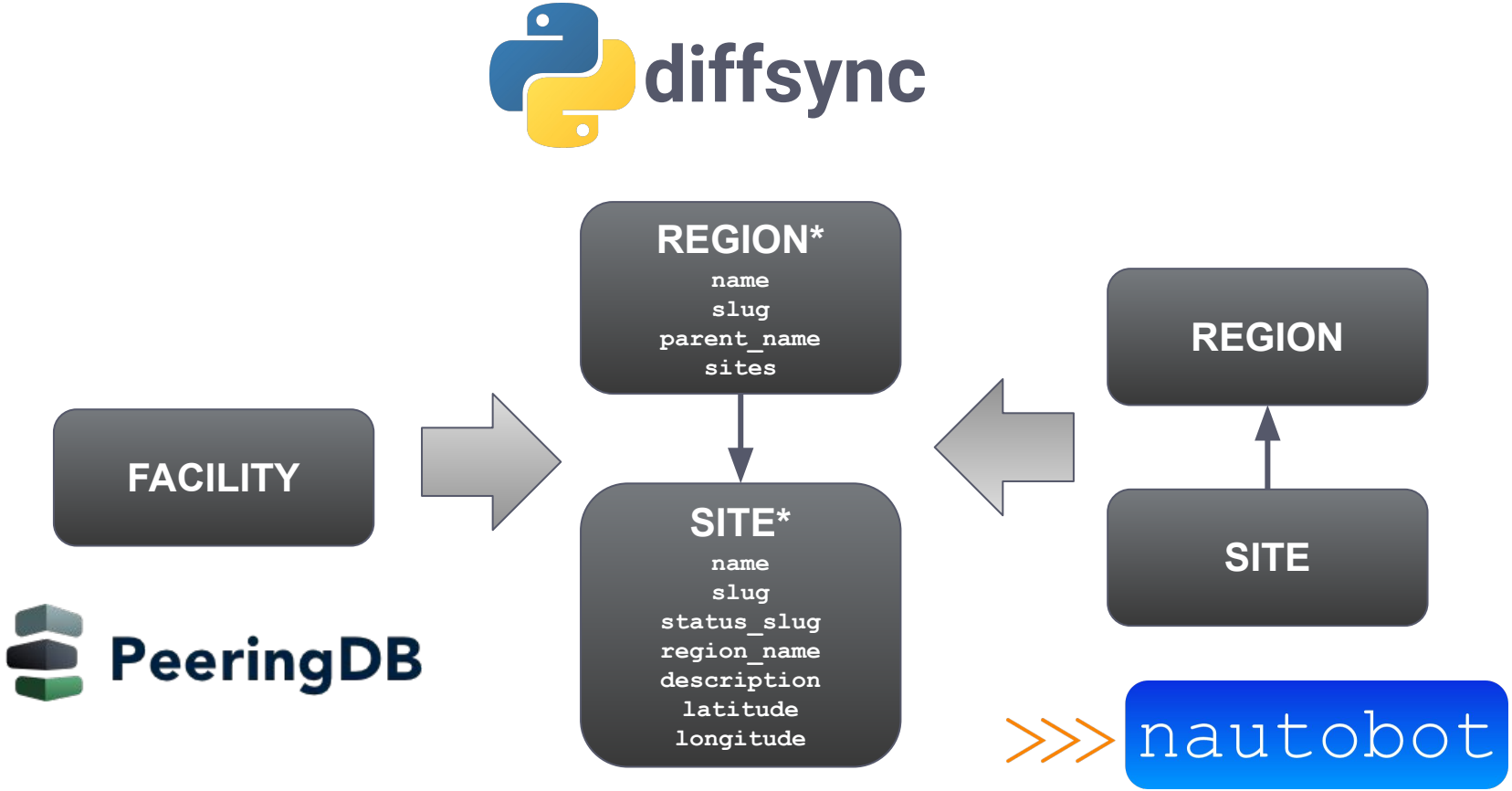
>>> Scenario

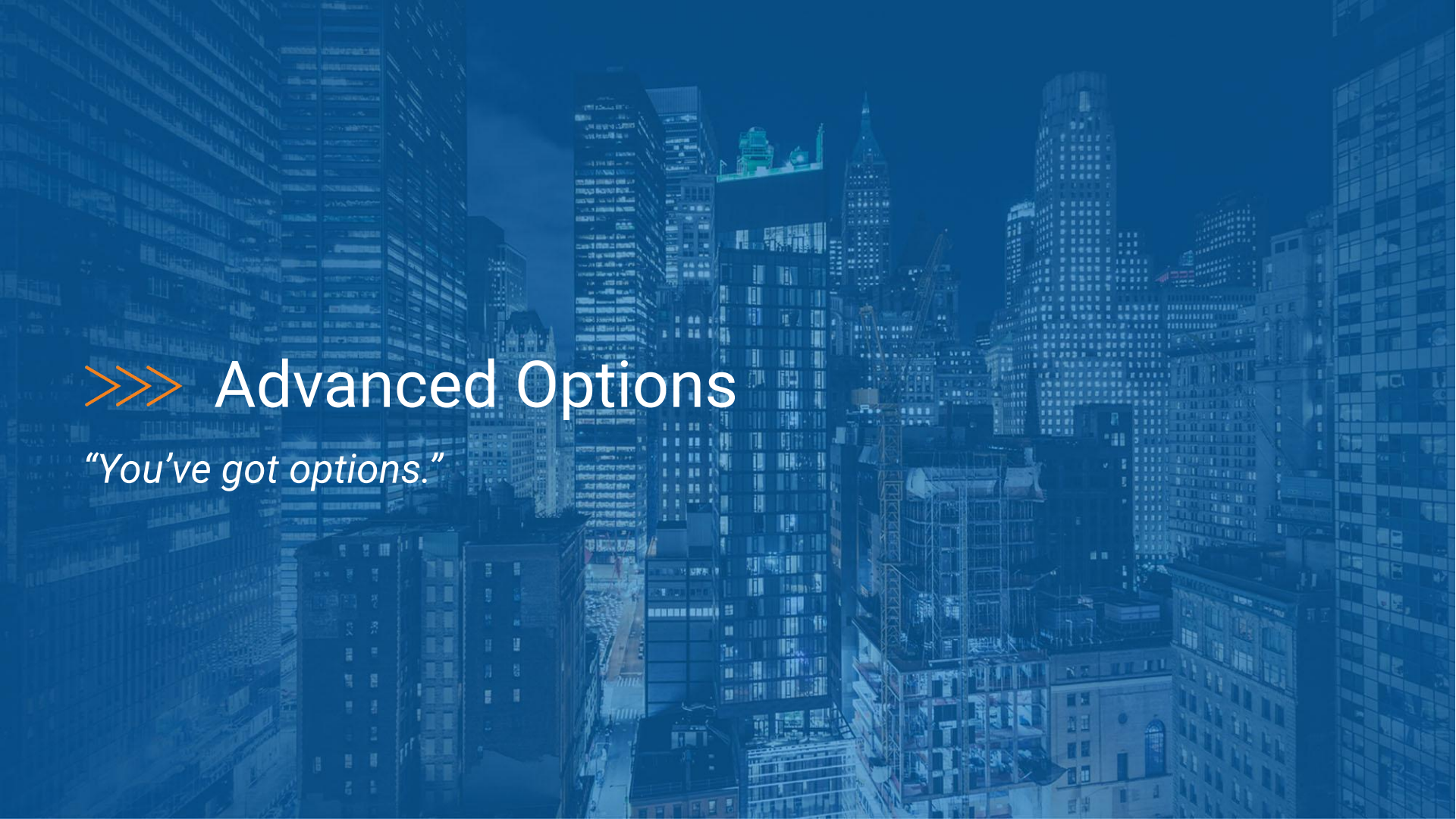




** PeeringDB is a public database of networks*

Scenario


diffsync
REGION*
name
slug
parent_name
sites
REGION
FACILITY
SITE*
name
slug
status_slug
region_name
description
latitude
longitude
SITE
PeeringDB
nautobot



Advanced Options

“You’ve got options.”

>>> DiffSync Flags

Customize DiffSync behavior with the Global and Model Flags

Global Flags: These are flags that affect how all data is handled. These are mostly focused on how to handle synchronization failures, unmatched objects, and logging.

Name	Description	Binary Value
CONTINUE_ON_FAILURE	Continue synchronizing even if failures are encountered when syncing individual models.	0b1
SKIP_UNMATCHED_SRC	Ignore objects that only exist in the source/"from" DiffSync when determining diffs and syncing. If this flag is set, no new objects will be created in the target/"to" DiffSync.	0b10
SKIP_UNMATCHED_DST	Ignore objects that only exist in the target/"to" DiffSync when determining diffs and syncing. If this flag is set, no objects will be deleted from the target/"to" DiffSync.	0b100
SKIP_UNMATCHED_BOTH	Convenience value combining both SKIP_UNMATCHED_SRC and SKIP_UNMATCHED_DST into a single flag	0b110
LOG_UNCHANGED_RECORDS	If this flag is set, a log message will be generated during synchronization for each model, even unchanged ones.	0b1000

```
from diffsync.enum import DiffSyncFlags
flags = DiffSyncFlags.CONTINUE_ON_FAILURE
diff = nautobot.diff_from(local, flags=flags)
```

>>> DiffSync Flags

Customize DiffSync behavior with the Global and Model Flags

Model Flags: These are flags that are specific to an individual object or an instance of that object.

Name	Description	Binary Value
IGNORE	Do not render diffs containing this model; do not make any changes to this model when synchronizing. Can be used to indicate a model instance that exists but should not be changed by DiffSync.	0b1
SKIP_CHILDREN_ON_DELETE	When deleting this model, do not recursively delete its children. Can be used for the case where deletion of a model results in the automatic deletion of all its children.	0b10

```python
from diffsync import DiffSync
from diffsync.enum import DiffSyncModelFlags
from model import MyDeviceModel

class MyAdapter(DiffSync):

    device = MyDeviceModel

    def load(self, data):
        """Load all devices into the adapter and add the flag IGNORE to all firewall devices."""
        for device in data.get("devices"):
            obj = self.device(name=device["name"])
            if "firewall" in device["name"]:
                obj.model_flags = DiffSyncModelFlags.IGNORE
            self.add(obj)
```

>>> Custom Diff Class

Control Behavior of DiffSync Processing and Data Handling

```
>>> from diffsync.enum import DiffSyncFlags
>>> from diff import AlphabeticalOrderDiff
>>> diff = remote_adapter.diff_from(local_adapter, diff_class=AlphabeticalOrderDiff)
>>> type(diff)
<class 'AlphabeticalOrderDiff'>
```

```
class MixedOrderingDiff(Diff):
    """Alternate diff class to list children in alphabetical order, except devices to be ordered by CRUD action."""

    @classmethod
    def order_children_default(cls, children):
        """Simple diff to return all children in alphabetical order."""
        for child_name, child in sorted(children.items()):
            yield children[child_name]

    @classmethod
    def order_children_device(cls, children):
        """Return a list of device sorted by CRUD action and alphabetically."""
        children_by_type = defaultdict(list)

        # Organize the children's name by action create, update or delete
        for child_name, child in children.items():
            action = child.action or "skip"
            children_by_type[action].append(child_name)

        # Create a global list, organized per action
        sorted_children = sorted(children_by_type["create"])
        sorted_children += sorted(children_by_type["update"])
        sorted_children += sorted(children_by_type["delete"])
        sorted_children += sorted(children_by_type["skip"])

        for name in sorted_children:
            yield children[name]
```

SSoT Overview

https://github.com/networktocode/diffsync/tree/main/examples/05-nautobot-peeringdb

>>> Extending the SSoT Plugin

Just the basics

1. Create one or more DiffSyncModel classes to define the data you wish to synchronize.
2. For each model you'll need to make a create, update, and delete method that handles taking the data from the DiffSyncModel and pushing it to your desired DataTarget.
3. Then create DiffSync adapter classes for each System of Record to load data into each of the DiffSyncModels that you created. This means you'll need to create one for Nautobot and one for your other SoR like ServiceNow.
4. Finally, write your Nautobot Job to handle the synchronization of data. It should be derived from the DataSource or DataTarget classes. In this class you'll need to write a *sync_data* method for handling the synchronization between your DataSource and DataTarget. You can also add the additional methods *config_information* and *data_mappings* to display additional information in the GUI or an *object_lookup* method to handle finding DiffSyncModel objects in Nautobot.


>>> network .toCode()
Thank you