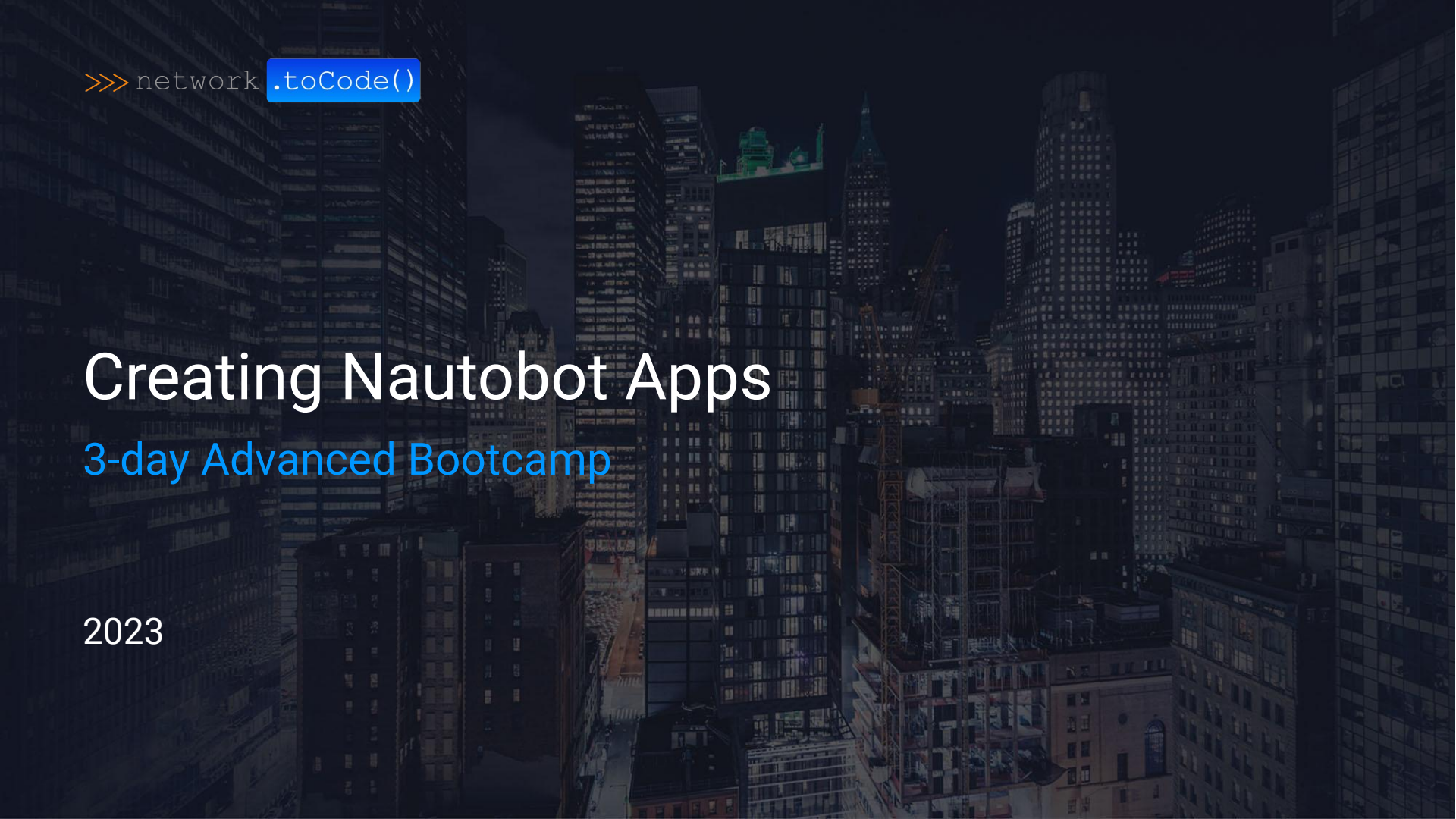

>>> network .toCode()
Creating Nautobot Apps
3-day Advanced Bootcamp
2023

Agenda

Introduction

Nautobot Overview

Introducing Apps (Plugins)

Setting up the Dev Environment

Creating Custom Models

Writing Views and Templates

Agenda, Contd.

Using Forms

Customizing the Navigation Menu

Extending the REST API

Creating Jobs

Populating Extensibility Features

Nautobot-hosted Documentation

Syncing with External Sources

Introduction

Network to Code, Training & Enablement, Instructors, Course Overview

Who is Network to Code?

Network Automation Solutions Provider

Founded in 2014, we help companies transform the way their networks are deployed, managed, and consumed using network automation and DevOps technologies.

A Diverse Team, with Deep Expertise

Engineers and developers in network automation, software and security, with leadership from vendors, integrators, and top tier consulting firms - all drive value to our clients.

Vendor Neutral Community

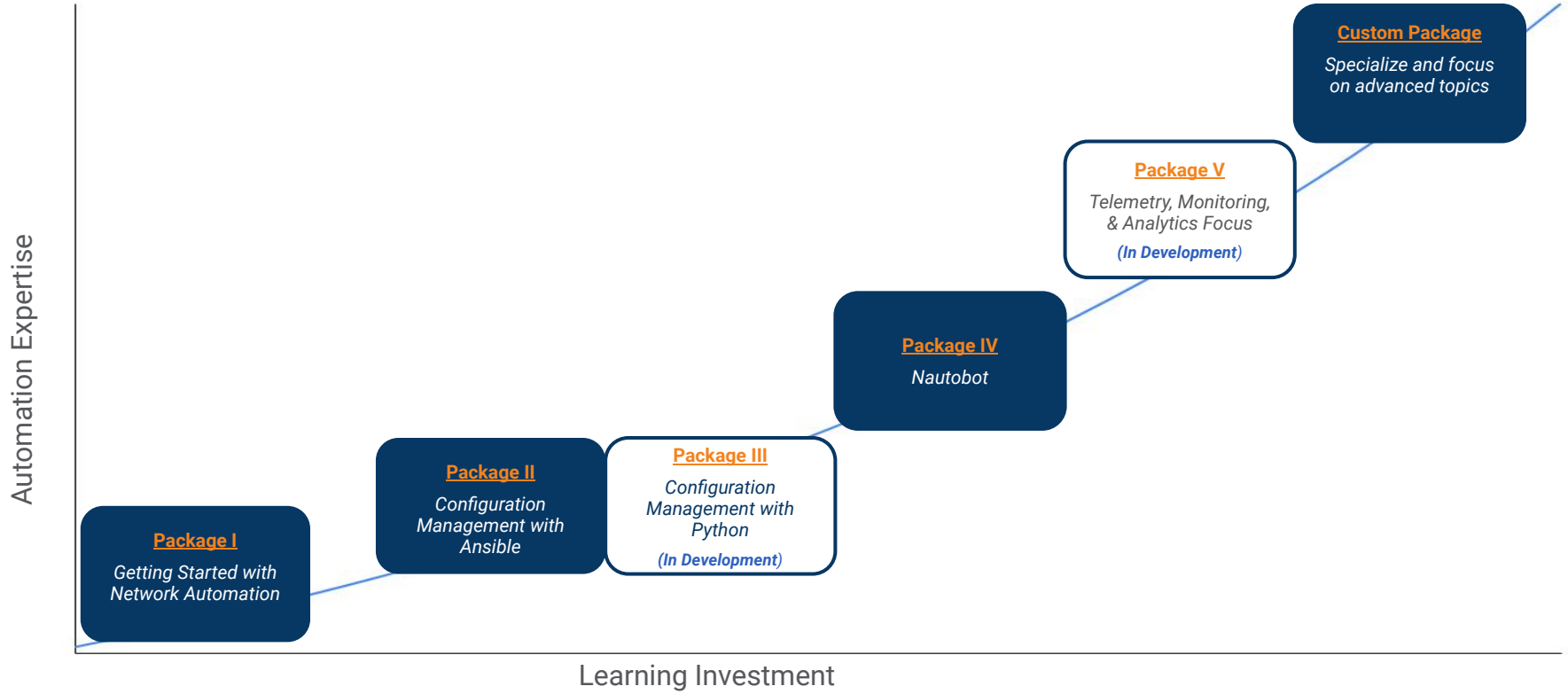
Partner with all OEMs, develop solutions with commercial and open source components. Host 19,000+ members and 300+ channels at slack.networktocode.com

Industry Recognized Thought Leaders

Working with clients across all industries and geographies, we promote a vendor and tool neutral approach, making automation a reality for any network.

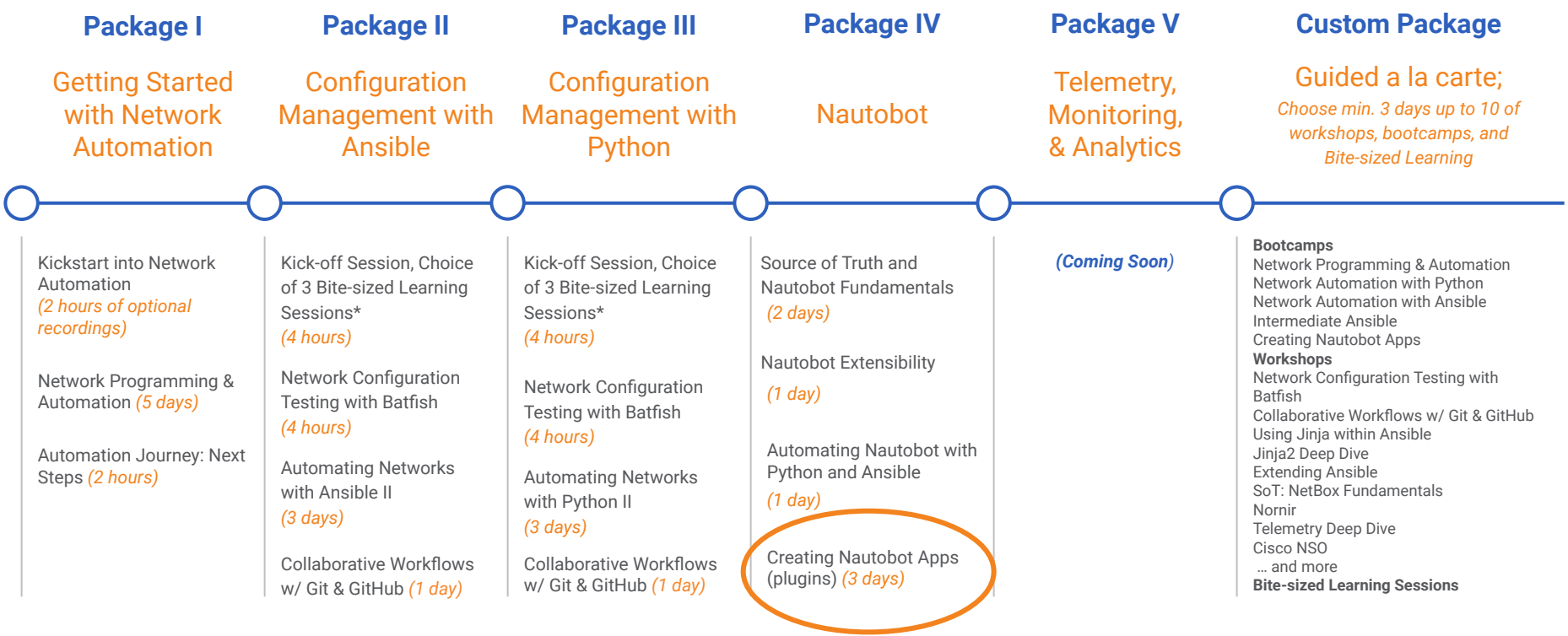
>>> The Network Automation Journey: Enablement Packages

Pre-Built Packages from Start to Finish



>>> The Network Automation Journey: Enablement Milestones

Pre-Built Packages from Start to Finish



>>> Instructor Introductions

Adam Byczkowski

Instructor &

Network Automation Engineer

Greg Mueller

Instructor &

Network Automation Engineer

>>> Intended Audience

- Nautobot power users who want to extend its functionality
- Software developers who need to integrate with Nautobot

>>> Course Prerequisites

- General familiarity with Nautobot
- Intermediate-level Python experience
- Experience writing Django/Jinja2 templates
- Basic understanding of REST API fundamentals
- Experience developing Django applications
  - Creating models
  - Writing views & templates
  - Using forms and validation

>>> Course Outline

Day 1

0. Nautobot Overview
1. Introducing Apps (Plugins)
2. Setting up the Development Environment
3. Creating Custom Models

Day 2

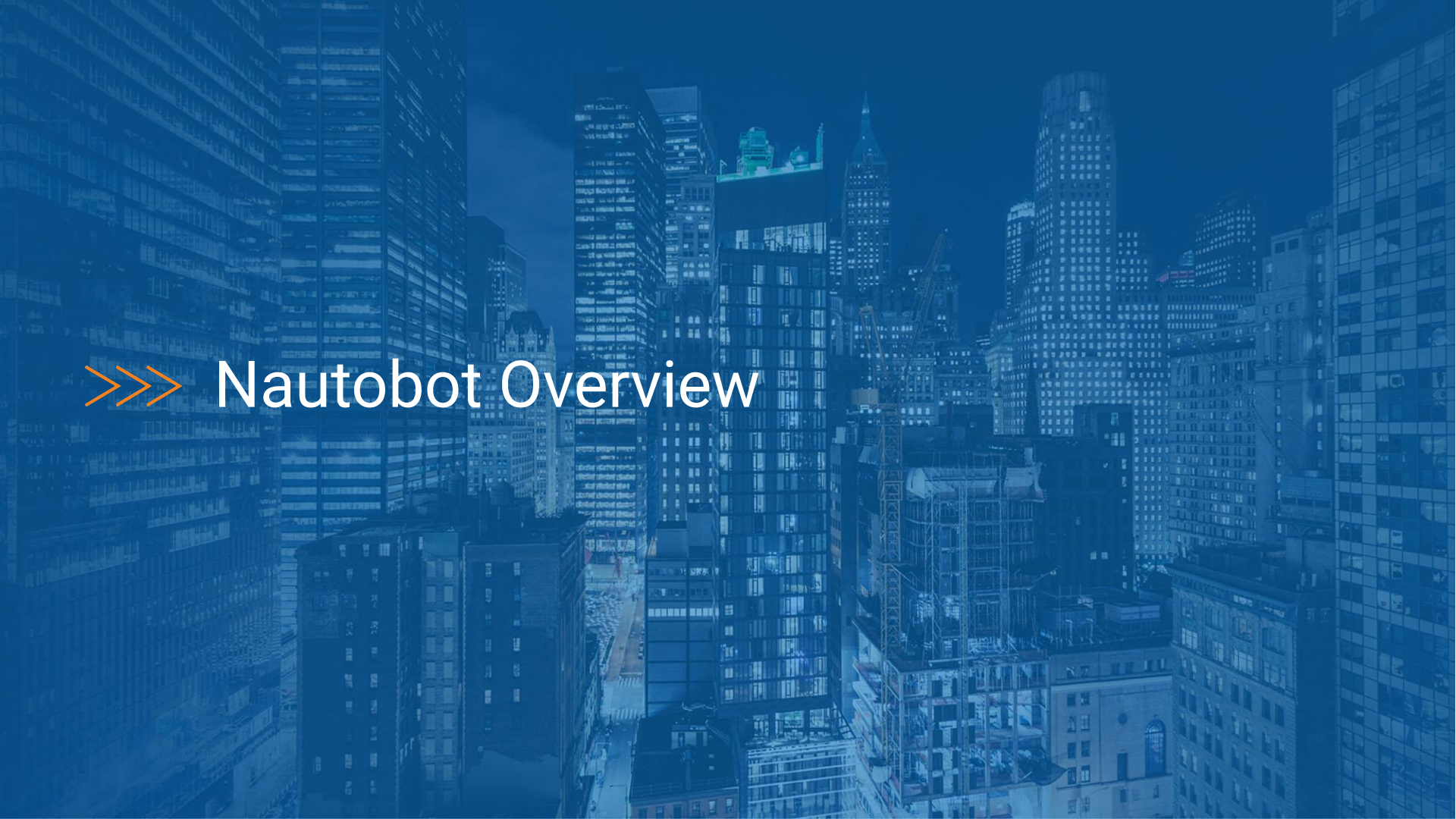
Bonus Demo: Exploring the ORM

4. Writing Views and Templates
5. Using Forms
6. Customizing the Navigation Menu

Day 3

Challenge Demo: Adding a Banner

7. Extending the REST API
8. Creating Jobs
9. Populating Extensibility Features



Nautobot Overview

>>> About Nautobot

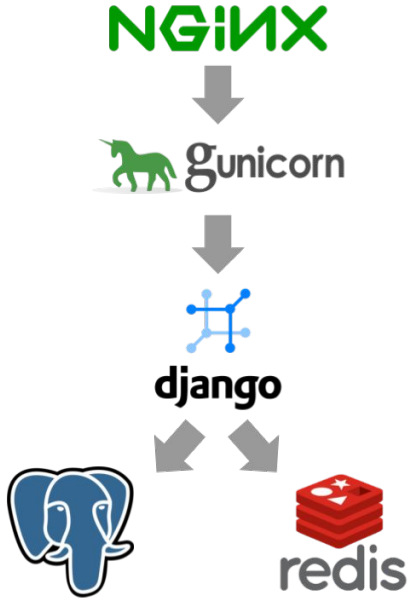
- Web application which functions as the source of truth for a network
  - IPAM, DCIM, circuits, tenancy, secrets, etc.
- Highly structured data model with myriad integration points
  - REST API, webhooks, plugins, export templates, CLI shell, etc.
- Built on the Django Python framework
- Open source (Apache 2 license) since 2021

>>> Design Philosophy

- Replicate the real world
  - Data model is tightly coupled to real-world constraints
- Function as a source of truth for the network
  - Use Nautobot to provision devices, not vice versa
- Keep things simple
  - High value and ease of maintenance are preferred over 100% complete solutions

>>> Application Architecture

HTTP daemon	**NGINX** Handles HTTP requests
WSGI	**uWSGI or Gunicorn** WSGI middleware
Application	**Django/Python** Framework for application logic
Database	**PostgreSQL or MySQL** Relational database
Queuing/caching	**Redis/django-cacheops** In-memory key-value store



>>> Application Architecture

- Nautobot is divided into Django “apps” corresponding to major functions
  - circuits: Data circuits & providers
  - dcim: Sites, racks, device types, devices, cables, etc.
  - extras: Extensions & integrations (e.g. webhooks, export templates)
  - ipam: Prefixes, IP addresses, VLANs, VRFs, etc.
  - secrets: Credentials storage
  - tenancy: Tenants and groups
  - virtualization: Clusters and virtual machines

>>> Integration Mechanisms

- REST API
  - Machine-focused CRUD operations; parallel to the web UI
- Webhooks
  - Automatically inform remote systems of changes to Nautobot data
- Export templates
  - Customized export of Nautobot objects
- Plugins
  - Python-level extensions to Nautobot

Section 1: Introducing Apps (Plugins)

>>> What can an App do?

- Create custom Django models
  - Define new database tables
  - Generate and run SQL database migrations
- Provide custom “pages” (views) within the UI
  - Implement custom business logic
  - Use Django Template Language (DTL) or Jinja2 to render templates
  - Register custom URLs under the `/plugins` path

>>> What can an app do?

- Inject custom content within certain stock views
  - Embed custom content/buttons in prescribed locations within core object views
- Add REST API endpoints
  - Provide full CRUD support for your plugin's models
- Insert custom middleware
- Leverage the complete Python environment and Django ORM

>>> Plugin Examples

- nautobot-golden-config
  - Provide context around golden configuration
- nautobot-plugin-device-onboarding
  - Easily onboard new devices
- nautobot-plugin-ssot
  - Facilitates integration and data synchronization between various "source of truth" (SoT) systems
- nautobot-plugin-welcome-wizard
  - Assist users with the necessary initial steps in populating data within Nautobot

>>> When does a plugin make sense?

Plugin

- Interacts directly with data
- Extends the Nautobot UI/API
- Performs functions within the same general scope as Nautobot
- Can be written using the provided integration points
- Makes sense to deploy alongside every Nautobot instance

Standalone Application

- Moves data among systems other than Nautobot
- Function diverges from that of a network SoT
- Requires features beyond what is provided by the plugins API
- Has different deployment requirements than Nautobot

>>> Plugin Installation

1. Enter the Nautobot virtual environment
2. Install the Python package
3. Record it as a local requirement
    a. This ensures the plugin gets reinstalled during upgrades

```
$ source venv/bin/activate

(venv)$ pip install nautobot-plugin-foo

(venv)$ echo nautobot-plugin-foo >> local_requirements.txt
```

>>> Plugin Installation

4. Enable the plugin by adding it to the `PLUGINS` list in `nautobot_config.py`
5. Configure plugin:
   a. Specify any required/desired plugin-specific parameters

```
PLUGINS = [
    'nautobot_foo'
]

PLUGINS_CONFIG = {
    'nautobot_foo': {
        'foo': 'abc',
        'bar': 123,
      }
}
```

>>> Plugin Installation

6. Run `post_upgrade`
7. Restart Nautobot services

```
(venv)$ nautobot-server post_upgrade

(venv)$ sudo systemctl restart nautobot nautobot-worker
```

>>> Demo: Lab 01

1. Setup development environment
2. Connect to the lab environment
3. Setup server for development
4. Install and enable an existing Nautobot plugin

Section 2: Setting up the Dev Environment

>>> Project Structure

- *project_name/*
  - *plugin_name/*
    - templates/
      - plugin_name/
        - *.html
    - __init__.py
    - models.py
    - …
    - views.py
  - README
  - setup.py

Git root
    Plugin root

HTML templates
PluginConfig
Django models
    Other Django components
Django views
    Documentation
Distutils script

>>> Initial Setup

- Create a project directory and initialize a git repository
  - Serves as the root for our project
  - Enables revision control

```
~$ mkdir plugin-project
~$ cd plugin-project/
~/plugin-project$ git init .
Initialized empty Git repository in ~/plugin-project/.git/
```

>>> Initial Setup

- Create plugin directory structure using the `startplugin` command
- Edit the `__init__.py` file to customize our `PluginConfig`

```
~/plugin-project$ nautobot-server startplugin my_plugin
~/plugin-project$ ls my_plugin/
__init__.py  migrations  models.py  navigation.py  tests
urls.py  views.py
~/plugin-project/my_plugin/$ edit my_plugin/__init__.py
```

>>> Creating a PluginConfig

- `PluginConfig` class is provided by Nautobot
  - Analogous to Django's `AppConfig` class
  - Provides plugin metadata and indicates scope of functionality
- Subclass it to make your own configuration class
  - Expose as config in `__init__.py`

>>> Define PluginConfig in __init__.py

```
from nautobot.extras.plugins import PluginConfig

class MyPluginConfig(PluginConfig):
       name = 'my_plugin'
       verbose_name = 'My First Plugin'
       description = 'A work in progress'
       version = '0.1'
       author = 'Author Name'
       author_email = 'author@example.com'
       required_settings = []
       default_settings = {}

config = MyPluginConfig
```

>>> PluginConfig Attributes

Name	Description
`name`	Plugin name (same as the plugin directory)
`verbose_name`	Human-friendly name
`version`	Current release
`description`	Short description of what the plugin does
`author`	Your name
`author_email`	Your email address
`base_url`	URL path for UI views & REST API endpoints
`required_settings`	Mandatory configuration parameters
`default_settings`	Default values for optional configuration parameters

>>> PluginConfig Attributes

Name	Description
`min_version`	Minimum compatible version of Nautobot
`max_version`	Maximum compatible version of Nautobot
`middleware`	List of custom middleware classes provided by the plugin
`caching_config`	Plugin-specific queryset caching configuration (for django-cacheops)
`template_extensions`	Python path to template extensions module (if not default)
`menu_items`	Python path to menu items module (if not default)

>>> Packaging the Plugin for Installation

- Create `setup.py` in the project root
  - Distutils installation script
  - Provides metadata about your plugin
  - Allows us to install the plugin within our Python virtual environment
- In practice, you may opt to use an alternative packaging tool such as `poetry`

>>> Packaging the Plugin for Installation

1. Navigate to the project directory (the root directory)
2. Create `setup.py`

```
~$ cd plugin-project/
~/plugin-project$ edit setup.py
```

>>> Creating setup.py

```
from setuptools import find_packages, setup

setup(
      name='my-plugin',
      version='0.1',
      description='An example Nautobot plugin',
      url='https://github.com/my-organization/my-nautobot-plugin',
      author='Author Name',
      author_email='author@example.com',
      license='Apache 2.0',
      install_requires=[],
      packages=find_packages(),
      include_package_data=True,
)
```

>>> Install the Plugin for Development

3. Activate the Nautobot virtual environment
4. Install the plugin for development (`develop`)
   - Links to our development path (instead of copying)
   - Ensures code changes are reflected immediately

```
~/plugin-project$ source /opt/nautobot/venv/bin/activate
(venv) ~/plugin-project$ python setup.py develop
```

>>> Install the Plugin for Development

5. Enable the plugin in Nautobot's `nautobot_config.py`
6. Restart the Nautobot service (if not using the development server)

```
$ edit /opt/nautobot/nautobot_config.py


PLUGINS = [
    'my_plugin',
]


$ sudo systemctl restart nautobot nautobot-worker
```

>>> Demo: Lab 02

1. Initialize a git repo for a new plugin
2. Create `setup.py`
3. Create template code using `startplugin` command
4. Create a `PluginConfig` subclass
5. Install the plugin for development

Section 3: Creating Custom Models

>>> Django MVT Concept

- Model: Structured data stored in a relational database
  - Table rows are handled as individual Python instances
- View: The business logic employed in request/response processing
  - get() and post() functions handle HTTP requests
- Template: Rendering of data for display to the user
  - DTL/Jinja2 renders HTML templates using context provided by views
- Similar to MVC concept:

Idiomatic term (MVC)	Django term (MVT)
Model	Model
View	Template
Controller	View

>>> Django Models

- Representation of an underlying SQL table
  - Fields = columns
  - Instances = rows
- Field types determine supported values
  - Character, integer, etc.
- All provided by Django

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=100)
    birthdate = models.DateField()
    height = models.IntegerField()

    class Meta:
        ordering = ['name']
```

>>> Django Model Data Fields

Name	Description
`AutoField`	Auto-incrementing integer (do not use for primary key)
`BooleanField`	True or false (or null)
`CharField`	String up to a specified length
`DateField, DateTimeField`	Date or date and time
`FileField, ImageField`	Stores the location of a file on disk (does not write to database)
`IntegerField, DecimalField`	Integer/decimal numbers (see also other iterations of these)
`JSONField`	Stores raw JSON
`TextField`	Arbitrary blob of text
`UUIDField`	Stores a UUID (cannot be auto-generated by database)

>>> Django Model Relational Fields

Name	Description
`ForeignKey`	Integer field which points to some instance of another model
`OneToOneField`	Similar to ForeignKey but must be unique
`ManyToManyField`	Employs an intermediate table to support many-to-many mappings
`GenericForeignKey`	Employs content types to reference an instance of *any* model

>>> Django Model Meta Options

Name	Description
`abstract`	Set to true if defining a model intended to be subclassed
`db_table`	Allows customizing the database table name
`ordering`	Specifies the field(s) by which instance should be ordered
`indexes`	Database indexes to create
`unique_together`	Fields that must form a unique set (complements unique=True)
`constraints`	Arbitrary database constraints to enforce
`verbose_name`, `verbose_name_plural`	Human-friendly name for the model (*not* an instance)

>>> Schema Migrations

- Creating a new model requires a new SQL table
- Django migrations can do this for us automatically
  - Migrations also handle adding, changing, and removing table fields (to a degree)
- Two management commands:
  - makemigrations: Create the schema migration files
  - migrate: Apply the migrations by executing SQL commands
- Migration files can alternatively be created/modified by hand, but not recommended

>>> Schema Migrations

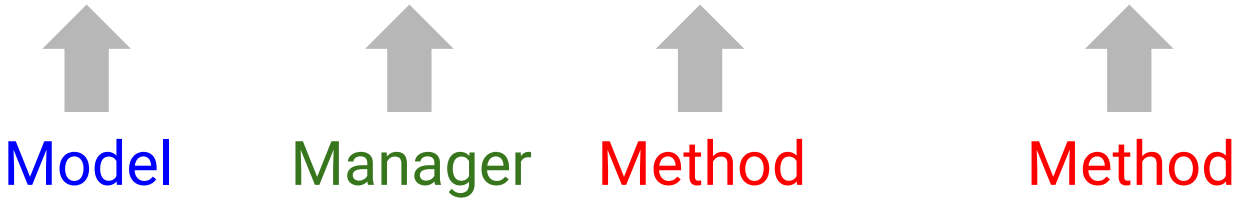
```
from django.db import migrations, models

class Migration(migrations.Migration):
    dependencies = [
        ('circuits', '0004_circuit_add_tenant'),
    ]

    operations = [
        migrations.AddField(
            model_name='circuit',
            name='upstream_speed',
            field=models.PositiveIntegerField(...),
        ),
    ]
```

>>> Django ORM

- Django's Object Relational Mapping employs QuerySets to abstract underlying SQL operations
- 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.

- 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
>>> DeviceRole.objects.all()
<QuerySet [<DeviceRole: BGP Controller>, <DeviceRole: Console Server>, <DeviceRole: Core Switch>, ...]>
# Count all objects
>>> DeviceRole.objects.count()
53

# Return only objects matching certain filters
>>> DeviceRole.objects.filter(vm_role=True)
<QuerySet [<DeviceRole: Console Server>, <DeviceRole: Core Switch>, <DeviceRole: DDoS Mitigation, ...]>
# Count filtered objects
>>> DeviceRole.objects.filter(vm_role=True).count()
17
```

>>> Django QuerySets

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

```
>>> queryset = DeviceRole.objects.filter(vm_role=True)
>>> print(queryset.query)
SELECT "id", "name", "slug", "color", "vm_role", "description"
FROM "dcim_devicerole"
WHERE "vm_role" = True
ORDER BY "name" ASC
```

filter(vm_role=True)

Default ordering; manipulated with order_by()

>>> Django QuerySets

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

```
# Modifying a QuerySet is free
>>> queryset = DeviceRole.objects.all()
>>> queryset = queryset.filter(color='c0c0c0')
>>> queryset = queryset.order_by('name')

# Forcing evaluation of a QuerySet triggers a SQL query
>>> queryset.count()
3

# Equivalent to:
DeviceRole.objects.filter(color='c0c0c0').order_by('name').count()
```

>>> Common QuerySet Methods

Name	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

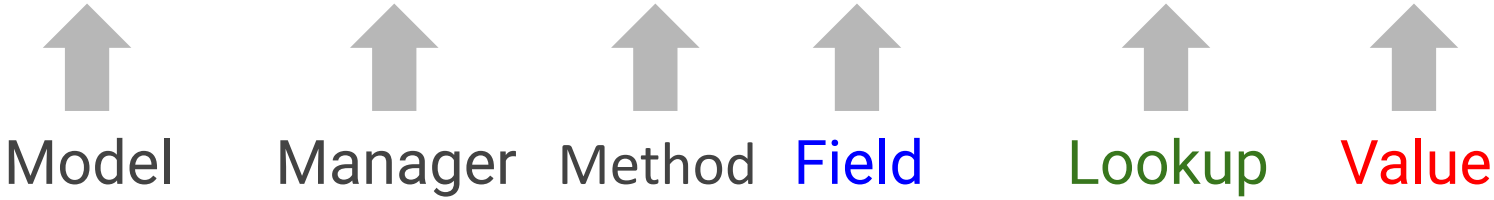
>>> Common QuerySet Methods

Name	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

```
>>> queryset = DeviceRole.objects.filter(name__startswith='A')
>>> print(queryset.query)
SELECT "id", "name", "slug", "color", "vm_role", "description"
FROM "dcim_devicerole"
WHERE "name"::text LIKE A%
ORDER BY "name" ASC
```

name__startswith='A'

>>> Common Field Lookups

Name	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

>>> 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 sites that are active AND in US
>>> US = Region.objects.get(name='United States')
>>> active = Status.objects.get(name='Active')
>>> Site.objects.filter(status=active, region=US)

# Same as above
>>> Site.objects.filter(status=active).filter(region=US)

# Find sites that are active OR in US
>>> from django.db.models import Q
>>> Site.objects.filter(Q(status=active) | Q(region=US))
```

>>> Django Admin UI

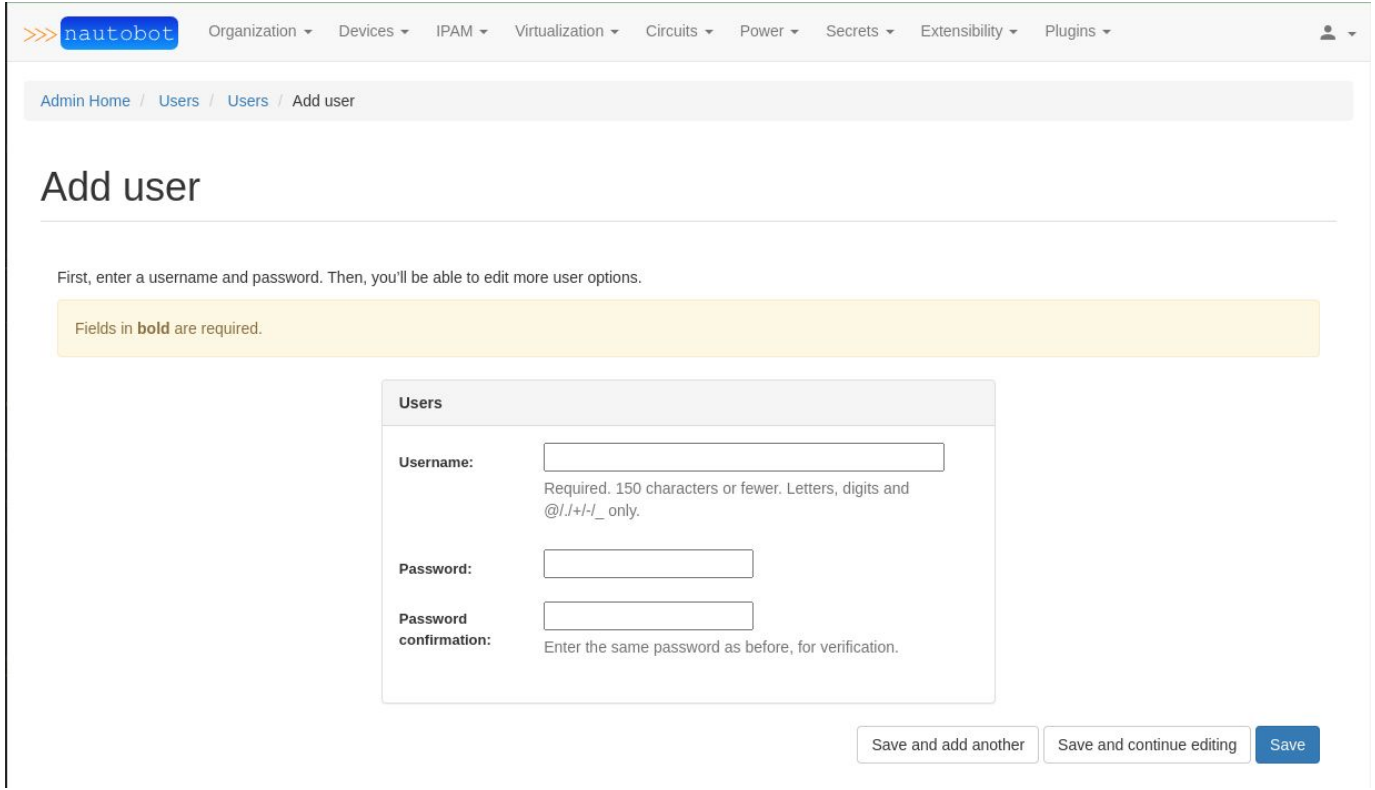
- Django provides an admin UI as a convenient backend for normal administrative functions
  - Creating, modifying, and deleting objects
  - Nautobot employs the admin UI for “backstage” objects
- Registering a model in `admin.py` provides generic forms & views for the model
- Basic functionality is robust but can be *customized extensively*

>>> Django Admin UI

1. Create `admin.py`
2. Subclass Django's `ModelAdmin`
3. Register it with a model

```
from django.contrib import admin
from .models import Person

@admin.register(Person)
class PersonAdmin(admin.ModelAdmin):
      pass
```

>>> Django Admin UI



>>> Demo: Lab 03

1. Create a custom model
2. Generate & run database migrations
3. Setup testing for our model
4. Extend the Django admin UI to support the new model

Bonus Demo: Exploring the ORM

Section 4: Views and Templates

>>> Views

- The “V” in MVT
  - Handle all business logic within the application
  - Move data between database and user
- A view is defined as a class with methods corresponding to actions
  - E.g. get() handles GET requests; post() handles POST
- View methods can be written from scratch, or leverage Django’s built-in generic views
  - This class will cover only class-based views
  - We’ll be using generic views to build our plugin

>>> Generic Views

- List views display a set of objects from the database
- Detail views show or manipulate a single object
- Stock CRUD operations are included; don't need custom methods

```
from nautobot.core.views import generic
from .models import MyModel

class MyModelListView(generic.ObjectListView):
    queryset = MyModel.objects.all()

class MyModelDetailView(generic.ObjectEditView):
    queryset = MyModel.objects.all()
```

>>> Registering Views

- Create URL patterns in urls.py to map URLs to views
  - Patterns append to the `base_url` for your plugin
  - Give each URL a name for easy reference

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.MyModelListView.as_view(), name='mymodel-list'),
    path('<pk>/', views.MyModelDetailView.as_view(), name='mymodel'),
]
```

>>> Referencing Views

- URL names can be referenced by `reverse()` to return complete URLs
- Define `get_absolute_url()` on each model for convenience

```
from django.db import models
from django.shortcuts import reverse

class MyModel(Model):
    ...

    def get_absolute_url(self):
        return reverse('mymodel', args=[self.pk])
```

>>> Testing Views

- Testing views in Django is similar to testing models
  - SetUpTestData is used in both
  - Base test classes both come from Django
- Nautobot provides some boilerplate code to assist in testing views
- When testing views, it is important to note that we are not testing what we would see through a web browser. We are testing that django views act as intended.

>>> Testing Views (continued)

- Testing views mostly deals with asserting a django view gives us an expected HTTP code.
  - 200 - OK
  - 201 - Created
  - 202 - Accepted
  - 401 - Unauthorized
  - 403 - Forbidden

>>> Templates

- The “T” in MVT
  - Templates help render the response to the user
  - Ensures separation of business logic from design
- Django provides its own template language (DTL)
  - Very similar to Jinja2 but more limited
  - Some of these limitations can be overcome with custom tags & filters
- HTML files stored in the templates/directory within your plugin

>>> Templates

- Nautobot plugins typically extend Nautobot's base.html template
- Override content blocks
  - title: Title text
  - header: Page header (directly above content)
  - content: Plugin content
  - javascript: Script inclusion tags

>>> Templates

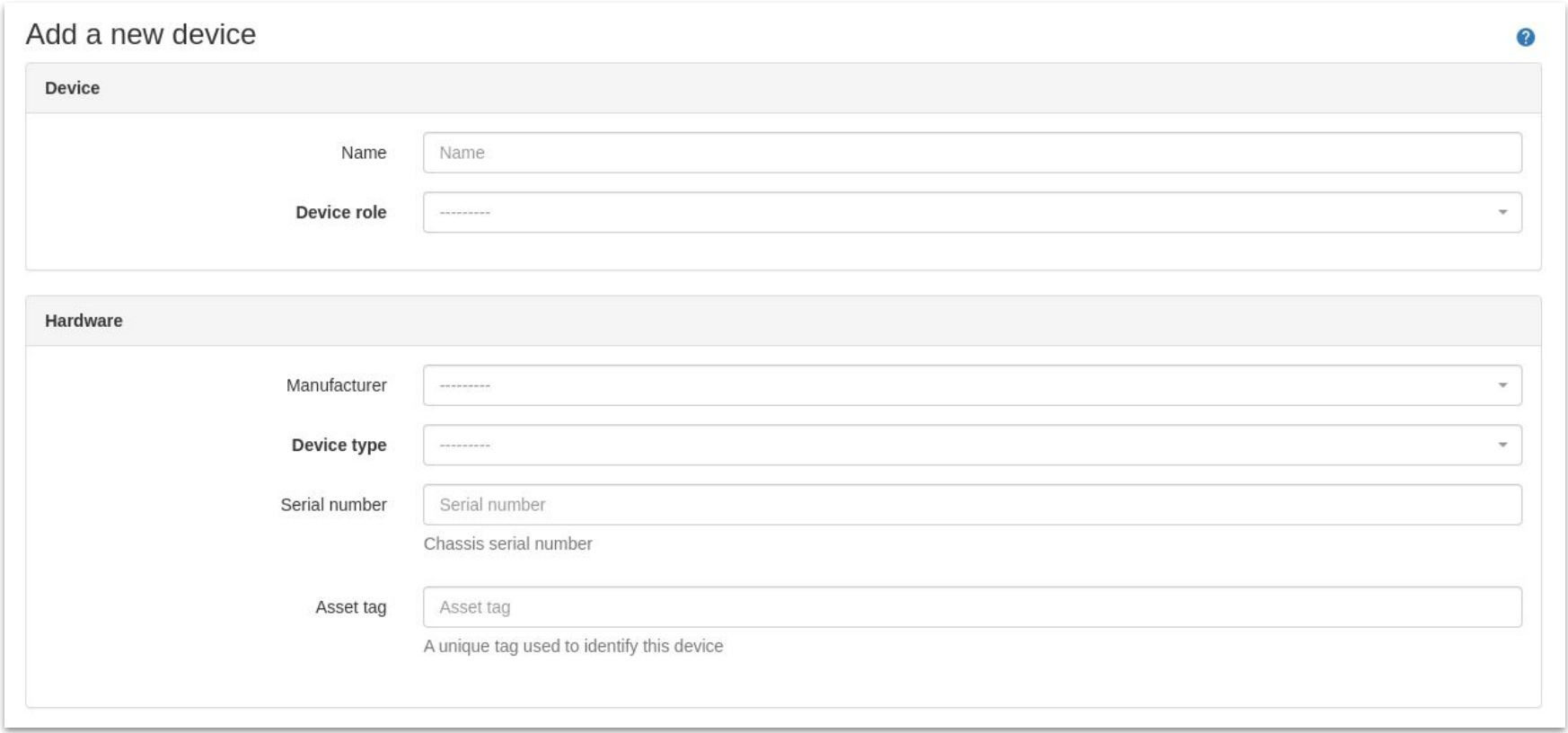
```
{% extends 'base.html' %}

{% block title %}
    Viewing {{ object }}
{% endblock title %}

{% block header %}
    <h1>{{ object }}</h1>
{% endblock header %}

{% block content %}
    ...
{% endblock content %}
```

>>> Demo: Lab 04

1. Create detail and list views for a model
2. Write a template for each view
3. Set up testing for our views

Section 5: Using Forms

>>> Forms


Add a new device
Device
Name
Name
Device role

Hardware
Manufacturer

Device type

Serial number
Serial number
Chassis serial number
Asset tag
Asset tag
A unique tag used to identify this device

>>> Forms

- Django forms assist in processing user input via views
  - Analogous to HTML forms
- Allow for validation and error reporting
  - The type of a field determines its valid values
  - Forms also support custom validation
- Form classes are easily rendered as HTML forms in templates

>>> Form Fields

- We define forms by specifying form fields
- Form fields are not the same as model fields
  - Model fields define how data is stored in the database
  - Form fields define how data is input within a view

>>> Common Form Fields

Name	Description
`BooleanField`	True or false
`CharField, TextField`	Text
`ChoiceField`	Single choice (pull-down list or radio buttons)
`DateField, TimeField`	Date/time values
`DecimalField, IntegerField`	Numerical values
`FileField, ImageField`	File upload fields
`MultipleChoiceField`	One or more choices (multi-select or checkboxes)
`ModelChoiceField`	Single choice from a queryset of model instances on page render
`ModelMultipleChoiceField`	ModelChoiceField with multiple choices

>>> Additional Nautobot Form Fields

Name	Description
`DynamicModelChoiceField`	Single choice from a queryset of model instances, built based on API call from model in queryset and performed clientside
`DynamicModelMultipleChoiceField`	DynamicModelChoiceField with multiple choices

>>> Django Form Example

```
from datetime import date
from django import forms

class PersonForm(forms.Form):
    name = forms.CharField(max_length=100)
    birthdate = forms.DateField()
    height = forms.IntegerField(help_text="Height in inches")

    def clean_birthdate(self):
        birthdate = self.cleaned_data['birthdate']
        if birthdate > date.today():
            raise forms.ValidationError("Birthdate can't be in the future!")
```

Custom validation for birth date

>>> Form from Models

- *But didn't we already do this for models?*
- Django can create forms from models automatically
  - Model fields are mapped to their appropriate form fields
  - Fields can be excluded/manipulated as desired
- Subclass forms.ModelForm and create a Meta child class
  - model: The model from which to generate the form
  - fields: The fields to include (or `__all__`)
    - Alternatively, use exclude to exclude only certain fields

>>> Django ModelForm Example

```python
from datetime import date
from django import forms
from .models import Person

class PersonForm(forms.ModelForm):

    class Meta:
        model = Person
        fields = ('name', 'birthdate')

    def clean_birthdate(self):
        birthdate = self.cleaned_data['birthdate']
        if birthdate > date.today():
            raise forms.ValidationError("Birthdate can't be in the future!")
```

Omitting the height field

Custom validation still supported

>>> Navigation Menu

- Plugins can extend the Nautobot navigation menu with their own links and buttons
- Plugin items appear below the “Plugins” dropdown
  - Hidden when no plugin items have been registered
  - Grouped by plugin name
- Edit **navigation.py** and define menu_items
  - Iterable of `PluginMenuItem` instances
  - Each menu item can include buttons

>>> PluginMenuItem Example

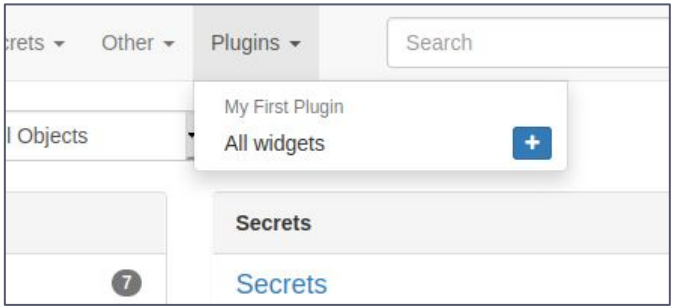
```
from extras.plugins import PluginMenuButton, PluginMenuItem
from utilities.choices import ButtonColorChoices

menu_items = (
      PluginMenuItem(
            link='plugins:myplugin:widgets',
            link_text='All widgets',
            buttons=(
                  PluginMenuButton(
                        link='plugins:myplugin:create_widget',
                        title='Add a widget',
                        icon_class='fa fa-plus',
                        color=ButtonColorChoices.BLUE
                  )
            ),
      ),
)
```

Menu items appear as links

Menu buttons are right-aligned within a link

>>> PluginMenuItem Example

- Menu items are grouped by plugin
- They appear in the order they are listed
- Menu items can be restricted to show for only certain users by specifying a set of required permissions




```
PluginMenuItem(
    link='plugins:myplugin:widgets',
    link_text='All widgets',
    permissions=['my_plugin.view_widget'],
    ...
)
```

>>> Permissions

- Nautobot provides an object-based permissions framework
- Replaces Django's built-in permissions model
- Object-based provides granular control
  - Not restricted to applying permissions to records of a specific type.

>>> Object Permissions

- A permission represents a relationship shared by several components:
  - Object types
  - Users/Groups
  - Actions - View, Add, Change, or Delete
  - Constraints - An arbitrary filter used to limit the granted actions to a specific subset of objects

>>> Object Permissions

- A permission represents a relationship shared by several components:
  - Object types
  - Users/Groups
  - Actions - View, Add, Change, or Delete
  - Constraints - Limit the granted actions to a subset of objects

>>> Object Permissions

- Four core actions
- Analogous to the CRUD operations
  - Add - Create a new object
  - View - Read an object from the database
  - Change - Update an existing object
  - Delete - Delete an existing object

>>> Constraints

Constraint	Description
`{ "status": "active"}`	Status is active
`{ "status__in": [ "planned", "reserved" ] }`	Status is planned OR reserved
`{ "status": "active", "role": "testing" }`	Status is active OR role is testing
`{ "name__startswith": "Foo" }`	Name starts with "Foo" (case-sensitive)
`{ "name__iendswith": "bar" }`	Name ends with "bar" (case-insensitive)
`{ "vid__gte": 100, "vid__lt": 200 }`	VLAN ID is greater than or equal to 100 AND less than 200
`[ {"vid__lt": 200}, {"status": "reserved"}` `]`	VLAN ID is less than 200 OR status is reserved

>>> Demo: Lab 05

1. Extend the Nautobot navigation menu
2. Restrict access based on permissions
3. Add a widget to a form
4. Associate a user to a maintenance notice

Section 6: Customizing the Navigation Menu

>>> App Settings

- `nautobot_config.py`
  - `PLUGINS_CONFIG`
- Per deployment settings

```
PLUGINS_CONFIG = {
    'my_nautobot_app': {
        'token': 'abc',
        'max_connections': 123,
     }
}
```

>>> PluginConfig

- Default settings
- Required settings
  - Will generate a `PluginImproperlyConfigured` exception if missing
- Specify in `PluginConfig`, typically found in `__init__.py`

```
class MyPluginConfig(PluginConfig):
    name = 'my_nautobot_app'
    ...
    required_settings = ['token']
    default_settings = {
        'max_connections': 50,
    }
```

>>> Access Settings

- Import settings
- Optionally assign to a name
- Access settings as dictionary values

```
from django.conf import settings

SETTINGS = settings.PLUGINS_CONFIG['my_nautobot_app']

def useful_function():
    token = SETTINGS['token']
    max_connections = SETTINGS['max_connections']
```

>>> Embedding Custom Content

- Plugins can embed custom content in certain locations within core views
  - Supported only for instance detail views
  - Locations include left side, right side, full page, and buttons

- Create template_content.py and define template_extensions
  - Iterable of PluginTemplateExtension subclasses
  - Specify a model and define the desired method(s) to render content

>>> PluginTemplateExtension Example

```
from extras.plugins import PluginTemplateExtension

class MyContent(PluginTemplateExtension):
      model = 'dcim.device'

      def right_page(self):
            return self.render('extra_device_content.html')

      def buttons(self):
            return self.render('extra_device_buttons.html')

template_extensions = [MyContent]
```

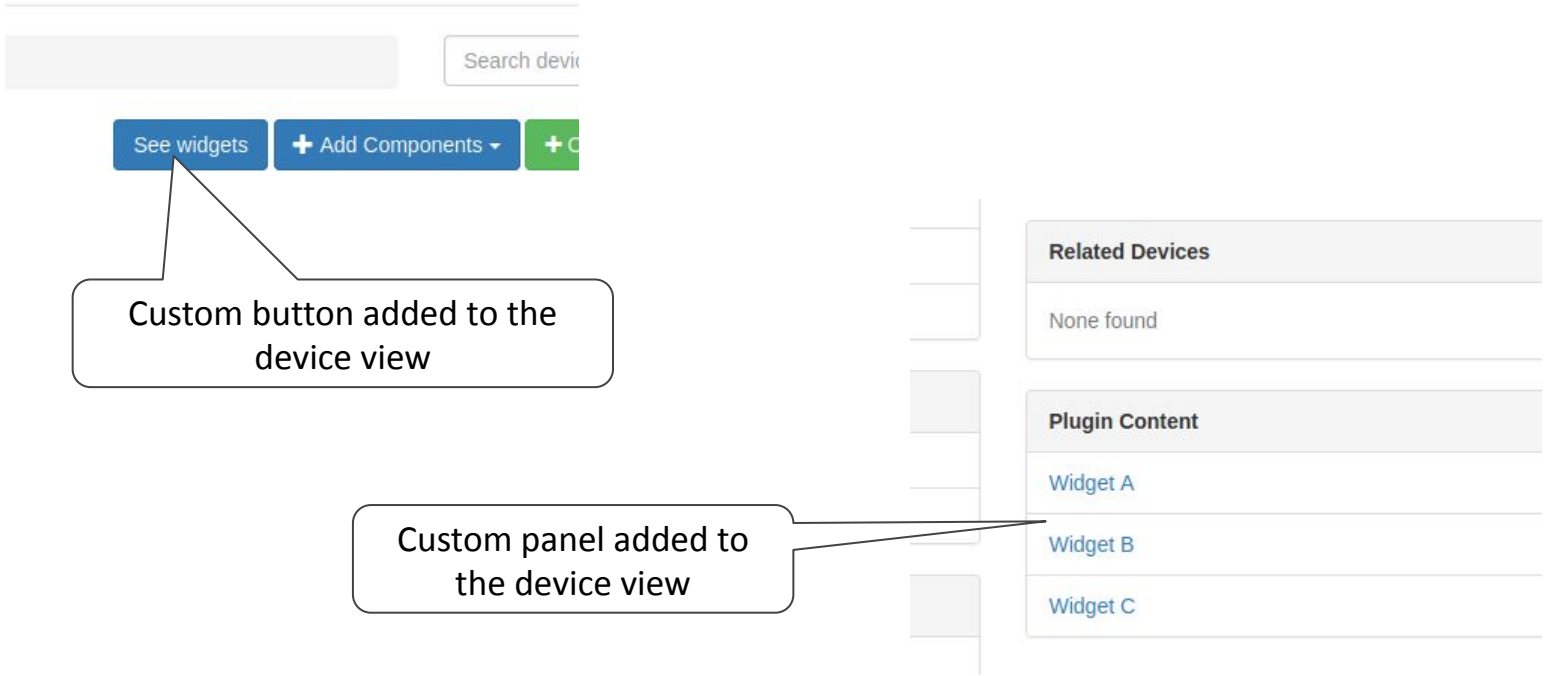
Methods can render templates from file or return content directly

>>> PluginTemplateExtension Example

```
<div class="panel panel-default">
    <div class="panel-heading">
        <strong>Plugin Content</strong>
    </div>
    <ul class="list-group">
    {% for widget in object.plugin_widgets.all %}
        <li class="list-group-item">
            <a href="{{ widget.get_absolute_url }}">{{ widget }}</a>
        </li>
    {% empty %}
        <li class="list-group-item">No widgets created for this device!</li>
     {% endfor %}
    </ul>
</div>
```

The object being viewed is provided by the template context

>>> PluginTemplateExtension Example



>>> Demo: Lab 06

1. Add a navigation menu link to view all maintenance notices
2. Add a button within that link to create a new notice
3. Extend the device view to show all active maintenance notices for a device
4. Add a button to the device view to create a maintenance notice for that device

Section 7: Extending the REST API

- Django REST Framework (DRF)
  - Django app which provides all the REST API components that we will use
  - The examples here are *not* Nautobot specific

- Serializers handle data display and validation
  - Analogous to Django forms, but with JSON
- Views contain the business logic
  - Separate view functions for e.g. GET vs. POST vs. DELETE
  - Separate view functions for detail vs. list views
  - Typically combined in a single ViewSet class
- Routers map URLs to views
  - AKA URL patterns in Django

>>> REST API Serializers

- Serializers perform two functions:
  - Validate input data on object creation/modification
  - Serialize objects for display in a standard format (JSON)
- As with forms, we can write our own serializers or build them automatically from models
  - Generated serializers can be extended/overriden with custom fields & methods

>>> REST API Serializers

```
### models.py
from django.db import models

class Person(Model):
     first_name = models.CharField()
     last_name = models.CharField()
     birthdate = models.DateField()

     class Meta:
          ordering = ['name']
```

>>> REST API Serializers

```
### api/serializers.py
from rest_framework import serializers
from my_app.models import Person

class PersonSerializer(serializers.ModelSerializer):
    full_name = serializers.SerializerMethodField(read_only=True)

    class Meta:
        model = Person
        fields = ('id', 'first_name', 'last_name', 'full_name', 'birth_date')

    def get_full_name(self, instance):
        return f"{self.first_name} {self.last_name}"
```

Custom serializer field not defined on the model

Custom method renders read-only values for full_name

>>> REST API Serializers

- The serializer takes data from PostgreSQL and serializes it for display as JSON
  - Ex: Date value becomes an ISO 8601-formatted string

```
{
      "id": 67,
      "first_name": "Hank",
      "last_name": "Hill",
      "full_name": "Hank Hill",
      "birthdate": "1953-04-19"
}
```

>>> REST API Views

- Views handle object display, creation, modification, and deletion
- HTTP verbs map to actions
- List views
  - GET: Retrieve a list of objects
  - POST: Create a new object
- Detail views
  - GET: Retrieve a single object
  - PUT/PATCH: Modify an existing object
  - DELETE: Delete an existing object

>>> REST API Views

```
class PersonListView(APIView):
      def get(self, request):
         ...

      def post(self, request):
         ...

class PersonDetailView(APIView):
      def get(self, request, pk):
         ...

      def put(self, request, pk):
         ...

      def delete(self, request, pk):
         ...
```

- Five total methods to handle all standard CRUD operations
- Writing these individually for every model would be extremely inefficient
- DRF provides *viewsets* to tackle the boilerplate for us

>>> REST API Views

```
from rest_framework.viewsets import ModelViewSet
from my_app.models import Person
from .serializers import PersonSerializer


class PersonViewSet(ModelViewSet):
      queryset = Person.objects.all()
      serializer_class = PersonSerializer

    def destroy(self, request, pk=None):
        super().detroy(request, pk)
        inform_next_of_kin(pk)
```

- ModelViewSet combines view *mixins*, each of which handles a separate CRUD operation
- Possible to mix and match mixins for an endpoint, or write your own
- Alternatively, override any of the stock methods
  - Ex: Override `destroy()` to implement custom logic upon deletion

>>> REST API Routers

- Routers provide automatic URL mapping to views
  - Simpler approach vs. specifying endpoint URLs manually
- Views/viewsets are registered to routers, which then return a list of URL patterns

```
### api/urls.py
from rest_framework.routers import DefaultRouter
from maintenance_notices.api import views

router = DefaultRouter()
router.register('people', views.PersonViewSet)

urlpatterns = router.urls
```

>>> Filtering API Queries

- Nautobot employs the django-filter package to enable limiting the objects returned in an REST API response
  - Functionality is built-into DRF viewsets
- Subclass FilterSet and define the appropriate model & fields
  - Can extend with custom filters as well
- Filters are invoked by appending query parameters to the URL
  - Ex: `GET /api/plugins/maintenance/notices/?duration=60`

>>> Filtering API Queries

```
import django_filters
from django.utils import timezone
from .models import MaintenanceNotice

class MaintenanceNoticeFilterSet(django_filters.FilterSet):
      active = django_filters.BooleanFilter(method='filter_active')

      class Meta:
            model = MaintenanceNotice
            fields = ('start_time', 'end_time', 'duration', 'devices', 'created_by')

      def filter_active(self, queryset, name, value):
            if value:
                  return queryset.filter(end_time__gt=timezone.now())
            else:
                  return queryset.exclude(end_time__gt=timezone.now())
```

Custom filters allow for arbitrary logic

Basic filters are built automatically from model fields

>>> Demo: Lab 07

1. Create a serializer for the MaintenanceNotice model
2. Create a viewset that support all five basic API operations
3. Register all API endpoint URLs using a router
4. Create and apply a FilterSet for MaintenanceNotices

Section 8: Creating Jobs

>>> What Are Jobs?

- Jobs provide a way for users to execute custom logic on demand within the Nautobot UI
- Jobs can interact directly with Nautobot data
  - Jobs have direct access to the Django ORM
- Jobs execute asynchronously as background tasks
  - Ideal for connecting to external data sources
- Jobs log messages and report status to the database
  - Update `JobResult` records
  - Create `JobLogEntry` records

>>> The Worker

- Jobs rely on the Python Celery package to execute jobs in the background
  - Celery loads your Python code at startup
  - Must be manually restarted to reflect code changes
- Nautobot communicates with the Celery worker via a Redis in-memory data store
  - Redis acts as a message broker
  - Provides caching, task queueing, and webhook features

>>> Three Ways to Install Jobs

- Installation in the `JOBS_ROOT` path
  - Defaults to `$NAUTOBOT_ROOT/jobs/`
  - Each file acts as a standalone module
- Imported from an external Git repository
  - Each file acts as a standalone module
- Packaged as part of a Plugin
  - Can import code from elsewhere
  - Can have dependencies on other packages

>>> Job Scheduling and Approvals

- Jobs can be scheduled to run:
  - Immediately
  - At a future date and time
  - On a specific interval
- Jobs can be scheduled via the UI or the API
- Jobs can be have `approval_required` flag set to `True`
  - One user can schedule a job that requires approval
  - The approver can approve or deny the request

>>> Creating Jobs

- Jobs subclass the `nautobot.extras.jobs.Job` class
  - `from nautobot.extras.jobs import Job`
  - `class DeviceEndOfLifeReport(Job):`
- Jobs implement some or all of these
  - Class attributes detailing behavior
  - Optional variables for input
  - A `run()` method that receives any input and executes first
  - A `post_run()` method that can be use for cleanup, sending emails, or triggering webhooks

>>> class Meta

- Class Metadata Attributes
  - `name:` Human friendly name displayed in Nautobot UI
  - `description:` Can be plain text or markdown
  - `approval_required:` Boolean Checkbox
  - `commit_default:` Should changes be committed to DB?
  - `field_order:` Order fields are presented on forms
  - `hidden:` Prevents job from being displayed in `Jobs` list
  - `read_only:` Prevents job from making changes to DB
  - `time_limit:` Terminate a `Job` if it runs too long

>>> Variables

- Pass user input to `Job` via the `run()` method
- Variable types define how the input is accepted in a UI form
- Variable options define how the input form is presented to a user
  - `default:` Default value presented to user
  - `description:` Brief description of the field
  - `label:` Human readable field name
  - `required:` Is this field mandatory
  - `widget:` A class of form widget used for input

>>> Variables Types

- `StringVar`: String text
  - `min_length, max_length, regex` (for validation)
- `TextVar`: Arbitrary text of any length
- `IntegerVar`: Numeric
  - `min_value, max_value`
- `BooleanVar`: True/False
- `ChoiceVar`: A set of user choices
  - `choices` (value, label) tuples of available choices
- `MultiChoiceVar`: Allows multiselection

>>> Variables Types (continued)

- `ObjectVar`: A specific Nautobot model
- `MultiObjectVar`: Allows multiple selection
- `FileVar`: An uploaded file that is stored in memory during job execution
- `IPAddressVar`: Returns a `netaddr.IPAddress` object
- `IPAddressWithMaskVar`: Returns `netaddr.IPNetwork`

>>> The post_run() method

- Runs after the `run()` and any `test_*()` methods.
- Runs even if `run()` or `test_*()` raise an exception.
  - This allows you to perform any cleanup needed.
- The result of the job is available using `self.results`
- You must save user data in the `run()` function to the job instance if you want to use that data in the `post_run()` function.
- Logging still applies

>>> The `run()` Method

```
from nautobot.extras.jobs import Job, MultiObjectVar
from nautobot.dcim.models import Device

class DeviceEOLReport(Job):
      devices = MultiObjectVar(model=Device)

      class Meta:
            name = 'Device End of Life Report'
            read_only = True

      def run(self, data=None, commit=None):
          """Executes the job's business logic."""
          devices = data["devices"]
          for device in devices:
              ...
```

>>> Logging

- Logs are stored in `JobLogEntry` records
- Each `JobLogEntry` record is associated with a `JobResult`
- Logging instance methods are available during Job execution
  - `self.log(message)`
  - `self.log_debug(message)`
  - `self.log_success(obj=None, message=None)`
  - `self.log_info(obj=None, message=None)`
  - `self.log_warning(obj=None, message=None)`
  - `self.log_failure(obj=None, message=None)`

>>> Demo: Lab 08

1. Creating a Query Job

Challenge Demo: Adding a Banner

>>> Challenge Demo: Adding a Banner

The goal is to add a banner across the top of the `Device` detail page only when that device is currently within a maintenance window.

- Search Nautobot Core documentation for banner
- Check the example_plugin within the Nautobot source code
- Start with sample code and make small changes each time
  - Limit the banner display to only Device detail pages
  - Limit the banner display to only when a within the Maintenance time window

Section 09: Populating Extensibility Features

>>> Nautobot Extensibility Features

- Computed Fields
  - `ComputedField` model
  - Actual computed fields are not stored in the database
- Custom Fields
  - Values stored as json field in model
  - `ModelInstance._custom_field_data`
- Relationships
  - `Relationship` model
  - Values stored in `RelationshipAssociation`

>>> Database Ready Signal

- Nautobot emits a custom signal
  - `nautobot_database_ready`
- Triggered after
  - `nautobot-server migrate`
  - `nautobot-server post_upgrade`

>>> Call Back Procedures

- Define callback (*receiver*) functions
  - `signals.py`
- Define `ready` method
  - `__init__.py`
  - `PluginConfig`

>>> Callback / Receiver Functions

```python
from nautobot.extras.choices import RelationshipTypeChoices

def add_relationships(sender, *, apps, **kwargs):
    ContentType = apps.get_model("contenttypes", "ContentType")
    Relationship = apps.get_model("extras", "Relationship")
    Provider = apps.get_model("circuits", "Provider")
    CircuitTermination = apps.get_model("circuits", "CircuitTermination")

    Relationship.objects.get_or_create(
        name="local access provider",
        type=RelationshipTypeChoices.TYPE_ONE_TO_MANY,
        source_type=ContentType.objects.get_for_model(Provider)
        source_label="circuit_termination",
        destination_type=ContentType.objects.get_for_model(CircuitTermination)
        destination_label="local_access_provider",
    )
```

>>> The `ready()` Method

```
from nautobot.extras.plugins import PluginConfig
from nautobot.core.signals import nautobot_database_ready
from . import signals

class MyPluginConfig(PluginConfig):
    ...

    def ready(self, data=None, commit=None):
        super().ready()
        nautobot_database_ready.connect(signals.add_custom_fields, sender=self)
        nautobot_database_ready.connect(signals.add_relationships, sender=self)
        nautobot_database_ready.connect(signals.add_custom_status, sender=self)

config = MyPluginConfig
```

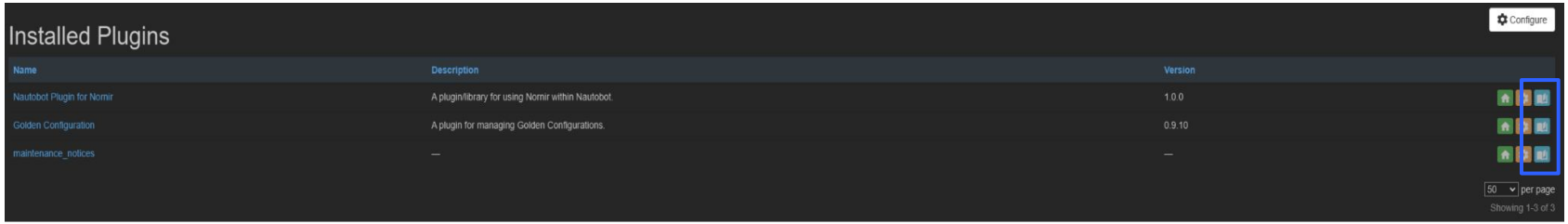
>>> Demo: Lab 9

1. Adding a Status Field to the Model
2. Populating a Relationship

Section 10: Nautobot-hosted Documentation

>>> Nautobot Hosted Documentation

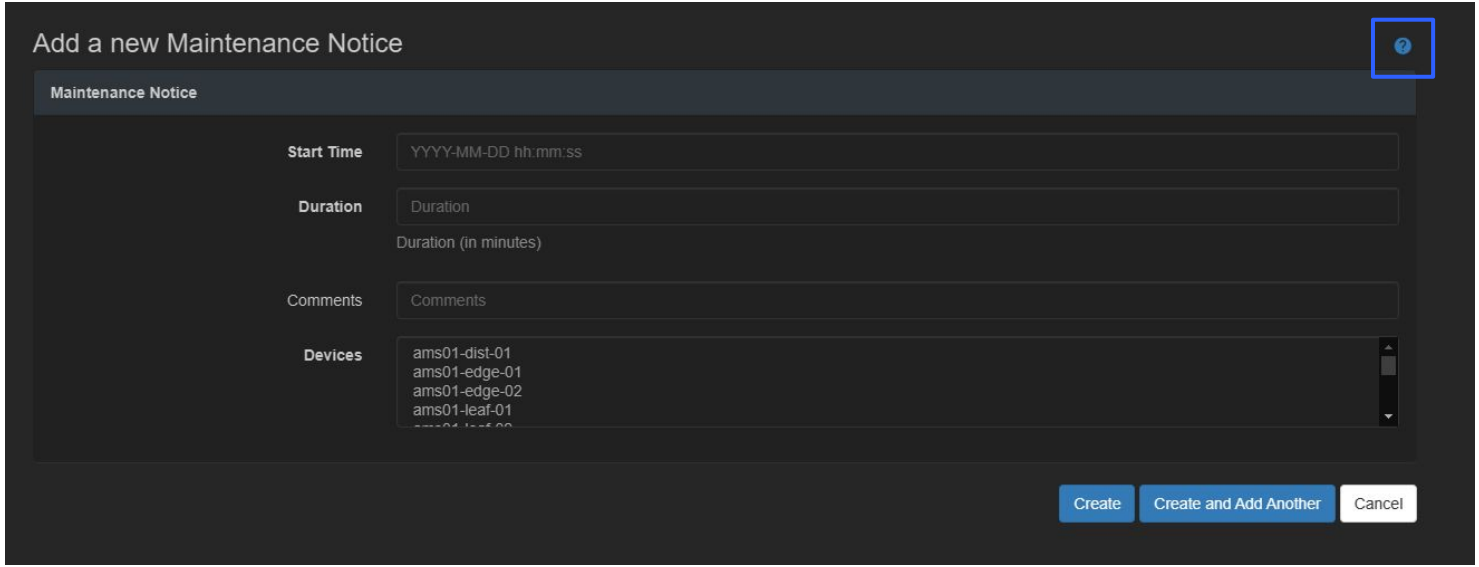
- Nautobot can internally host documentation for your plugin
- Nautobot provides two easy places to access this documentation from the GUI. The first is under the Installed Plugins page.



- The light blue buttons will take you to the docs homepage for the corresponding Nautobot plugin.

>>> Nautobot Hosted Documentation

- The second is on the on the add/edit form of any new models your plugin may have.



>>> Adding Plugin Level documentation

- Mkdocs is used as our static site generator and we will be using the settings from the Example Plugin in our demo Nautobot instance.
- These settings define things such as CSS, output directory (site_dir in the image below) of generated files.

```
---
dev_addr: "127.0.0.1:8001"
site_dir: "maintenance_notices/static/maintenance_notices/docs"
site_name: "Maintenance Notice Plugin Documentation"
copyright: "Copyright &copy; The Authors"
```

>>> Updating our Plugin to utilize generated documentation

- Generated documentation lives in the **`/static/{plugin_name}/docs`** folder in a plugin.
- Once documentation is generated, we must add a url to our **`urls.py`** that points to the documentation. This is typically the **`index.html`** file found in the aforementioned folder.

```
urlpatterns = [
  ...,
  path('docs/', RedirectView.as_view(url=static("maintenance_notices/docs/index.html")), name='docs'),
  ...,
]
```

>>> High Level overview of docs process

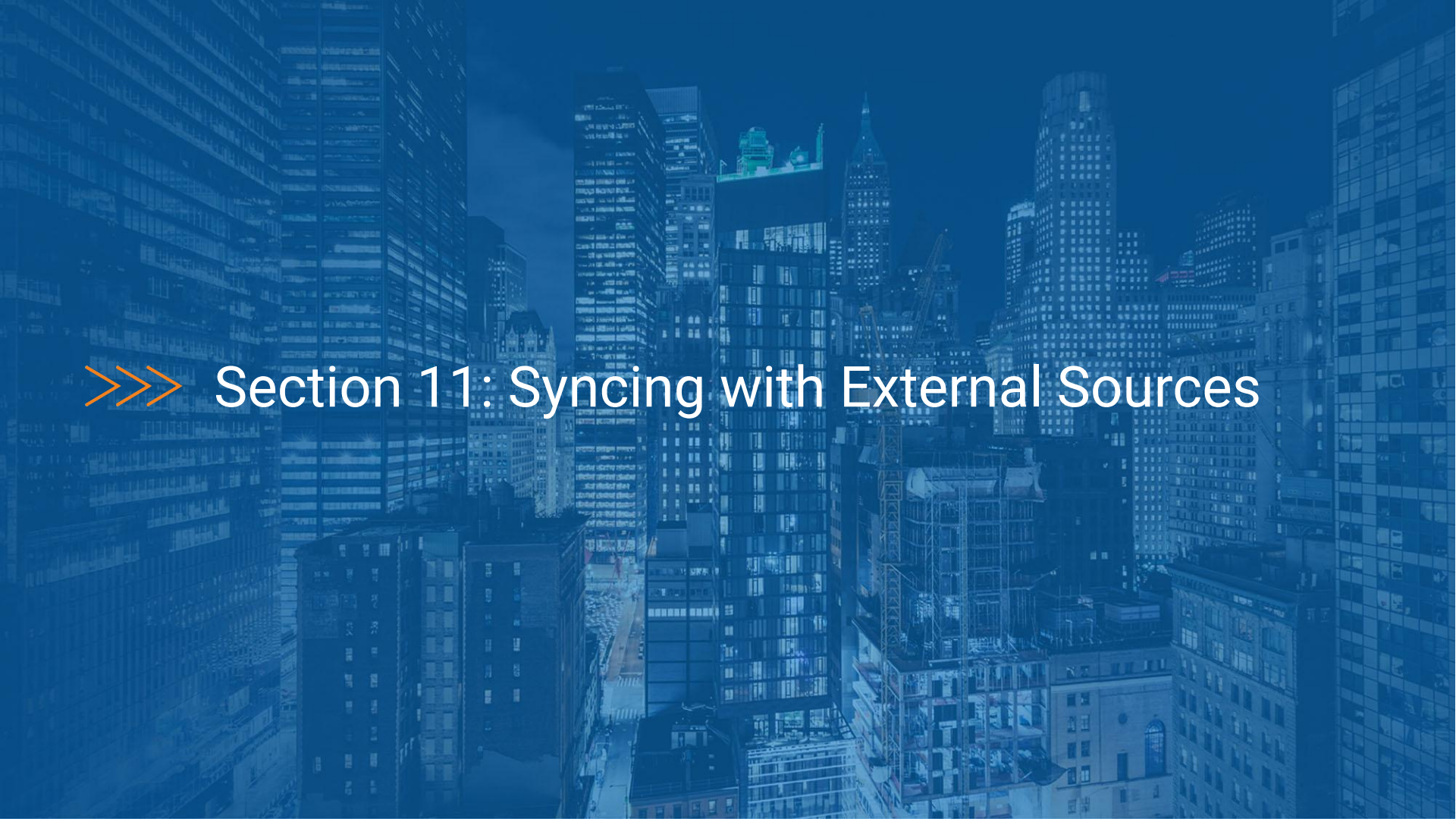
- Install prerequisites such as *mkdocs*.
- Define settings for your static documentation.
- Create any template files.
- Run mkdocs to generate html files.
- Update plugin **`urls.py`** pointing to the **index.html** of your created docs.
- Update the **`docs_view_name`** in your plugin **`__init__.py`** file.

>>> High Level overview of Model Documentation

- Create model folder in your plugin **docs** folder.
- Create a markdown file naming it the slug of the model you wish to create documentation for.
- Rebuild documentation to convert your markdown files into static html.

>>> Demo: Lab 10

1. Add documentation to your app
2. Add documentation to your app model



Section 11: Syncing with External Sources

>>> Syncing with External Sources

At its core, Nautobot acts as a “System of Record” (SoR)

As such, we expect the data within Nautobot to be accurate

How can we reliably ensure that Nautobot is up to date? 🤔

How can we reliably ensure that external systems are in sync with Nautobot? 🤔

>>> Nautobot Single Source of Truth (SSoT) App

The `nautobot-ssot` app facilitates data synchronization between various "Source of Truth" (SoT) systems

- Assumes only one source of truth for data (one-way sync)
- Different data groups may require different sync jobs
  - Sync office locations from a facilities management system
  - Sync IP Addresses to/from an IPAM/DHCP system
  - Sync device inventory to an SNMP management system
  - Sync data for specific tenants with their respective systems

>>> Nautobot Single Source of Truth (SSoT) App

`nautobot_ssot`

- Imported by your data sync app
- Provides base classes for data sync jobs
- Uses the Python DiffSync library to compare source and target data sources

DiffSync is a Python package that can be used to compare and/or synchronize sets of data between two sources

- DiffSync Models the attributes common to the data sources
- DiffSync Adapters read the data from the the data sources
- Adapters inherit the DiffSync comparison and sync methods
- An adapter can compare itself to/from another adapter
- An adapter can sync itself to/from another adapter

>>> DiffSyncModel

- Each instance of a `DiffSyncModel` represents a single record in a single system.
- The `DiffSyncModel` inherits from the Pydantic `BaseModel`.
- `Pydantic` is a Python package models data using type hints.
- `Pydantic` guarantees the object type for each attribute.

>>> Pydantic Model

- The Pydantic model provides schema validation for data records
- Pydantic will try to coerce values to the desired type when possible
- If a value is of the wrong type a `ValidationError` is raised.

```
record = {
  "item": "gum",
  "cost": 2.00,
  "quantity": "7",
}
```

```
class ItemModel(Base):
  item: str
  cost: float
  quantity: int
```

```
obj = ItemModel(**record)
obj.item == "gum"
obj.cost == 2.00
obj.quantity == 7
```

>>> DiffSyncModel

DiffSync Model requires several mandatory class-level attributes

- `_modelname`: Name used by Adapters to identify the models
- `_identifiers`: Instance field names used as a primary key
  - Used to match data between source and target

DiffSync Model supports several optional class-level attributes

- `_attributes`: Field names to be compared between systems
- `_children`: Dict `{<model_name>: <field_name>}` indicating how to store references to child model instances

>>> DiffSyncModel

The `DiffSyncModel` is where we define the write operations for the target system.

- Define your model attributes based on the data fields to sync
- Implement methods for writing data to the data target system
  - `create`: Create a new record on target system
  - `update`: Modify attributes of a record on target system
  - `delete`: Remove a record on target system

The `_identifiers` attribute specifies the names of the attributes that uniquely identify each record

DiffSync matches the `DiffSyncModel` instances from the source and target adapters, and compares them, then takes an action:

Result of Comparison	`DiffSyncModel` method called
Data exists in source but not destination	Call the `model.create` method
Source and destination attributes do not match	Call the `model.update` method
Data exists in destination but not source	Call the `model.delete` method
Source and destination attributes match	No method call needed

>>> DiffSync Adapters

A DiffSync Adapter represents a data source.

- An Adapter is defined for each system
- The `load` method is used to read data from the data source
  - The `load` method creates an instance of a DiffSyncModel for every data record
  - The Adapter collects each of the DiffSyncModel instances

>>> DiffSync Adapters

The DiffSync Adapter inherits the DiffSync engine, which is a collection of methods to handle comparison (diff) and synchronization with another adapter.

- `diff_from`
- `diff_to`
- `sync_from`
- `sync_to`

>>> DiffSync: Global Flags

Global flags can be can be passed to the `diff` and `sync` methods

```
from diffsync.enum import DiffSyncFlags
flags = DiffSyncFlags.SKIP_UNMATCHED_DST
source_adapter.sync_to(target_adapter, flags=flags)
```

Name	Description
`CONTINUE_ON_FAILURE`	Continue synchronizing even if failures are encountered when syncing individual models.
`SKIP_UNMATCHED_SRC`	Ignore objects that only exist in the source. (Disable create)
`SKIP_UNMATCHED_DST`	Ignore objects that only exist in the target. (Disable delete)
`SKIP_UNMATCHED_BOTH`	Combines `SKIP_UNMATCHED_SRC` and `SKIP_UNMATCHED_DST` into a single flag
`LOG_UNCHANGED_RECORDS`	Generated a log message during synchronization for each model, even unchanged ones.

The biggest benefit to the Nautobot SSoT app is the ability to wrap your DiffSync classes in a Nautobot job.

Additionally, Nautobot SSoT also provides:

- A Dashboard within the Nautobot UI
- Automatic saving of data sync results to the Nautobot database
- `SyncLog` to capture `DiffSync` logging

>>> SSoT Jobs

The SSoT job inherits from either the `DataSource` or `DataTarget` class, as well as the Nautobot `extras Job` class.

- `DataSource` is chosen when we are syncing from an external SoR system to Nautobot

```
class ExampleDataSource(DataSource, Job):
```

- `DataTarget` is chosen when Nautobot is the SoR and we are syncing to an external system

```
class ExampleDataTarget(DataTarget, Job):
```

>>> SSoT Job Methods

These are the Sync job methods that we need to define:

- `load_source_adapter`: Instantiate the source adapter
  - Assign adapter to the `self.source_adapter` attribute
  - Load data from the source system of record
- `load_target_adapter`: Instantiate the target adapter
  - Assign adapter to the `self.target_adapter` attribute
  - Load data from the target system

>>> Putting it all together

Here are the high level steps for building a data synchronization job.

1. Define the required data models with `DiffSyncModel`
2. Define a `DiffSync Adapter` for the System of Record
3. Define a `DiffSync Adapter` for the target system
4. Define a Nautobot SSoT `DataSource` or `DataTarget` job
5. Register the new job in Nautobot by triggering a migration
6. Enable the job

>>> Reference

DiffSync:

https://diffsync.readthedocs.io/en/latest/overview/index.html

https://blog.networktocode.com/blog/tags/diffsync

Nautobot Single Source of Truth (SSoT)

https://docs.nautobot.com/projects/ssot/en/latest/

https://blog.networktocode.com/blog/tags/ssot

>>> Demo: Lab 11

1. Installing the `nautobot-ssot` app
2. Setting up data sync capability
3. Adding a child model to existing sync job

Thank you

adam.byczkowski@networktocode.com
greg.mueller@networktocode.com