
>>> network .toCode()
Nautobot Apps Overview
Nautobot's Platform Capabilities
Tim Fiola
Developer Advocate
>>> nautobot

Agenda

Nautobot as an App Platform

Nautobot App Ecosystem

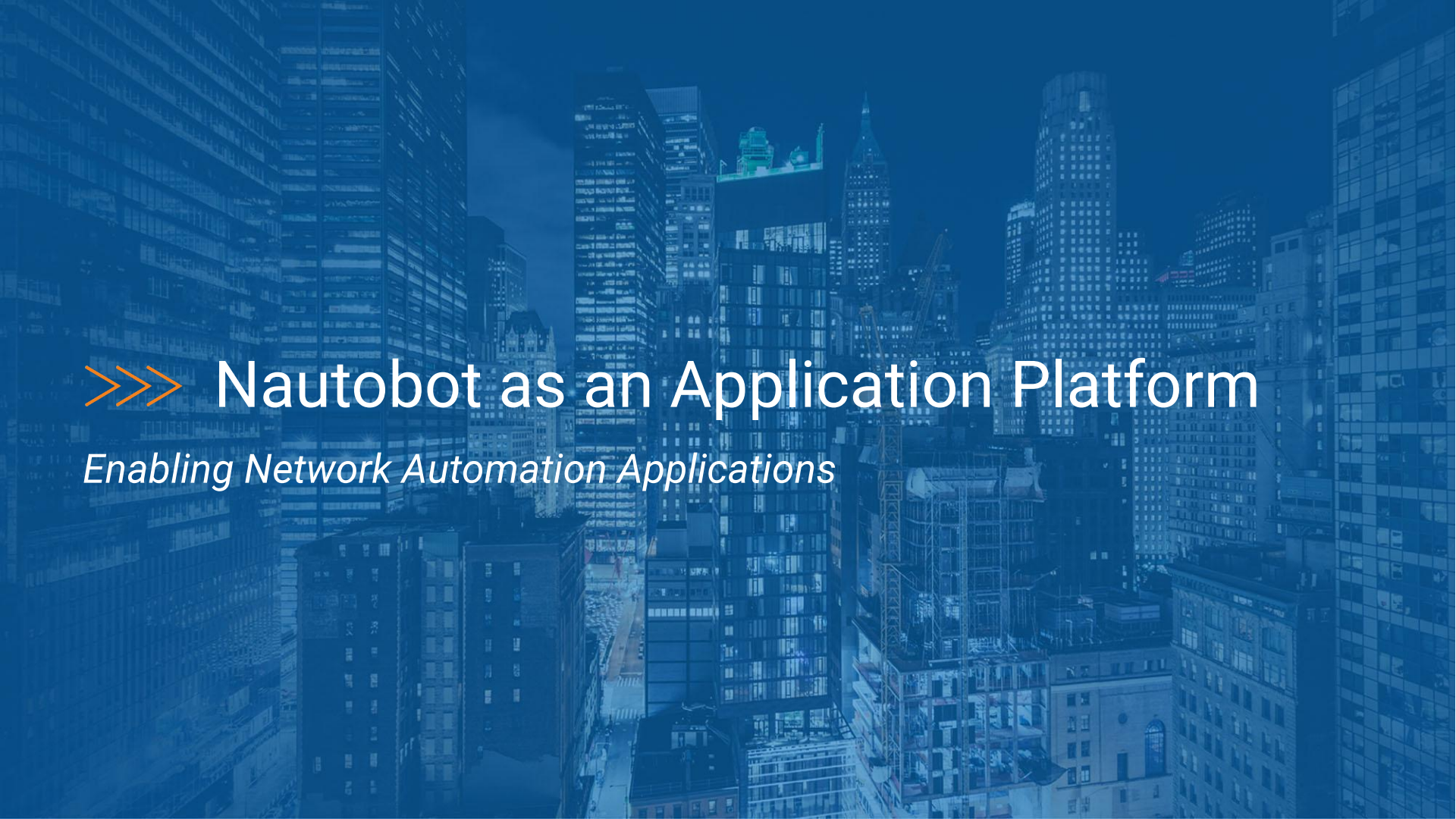
Nautobot App Capabilities Summary

Nautobot App Capabilities

Note: The term “App” is used interchangeably with “plugin.”

>>> Version Control

Author	Change Summary	Nautobot Version	Date
Tim Fiola	Initial creation of this document	v1.2.0 (pre-release)	8/16/2021



Nautobot as an Application Platform

Enabling Network Automation Applications

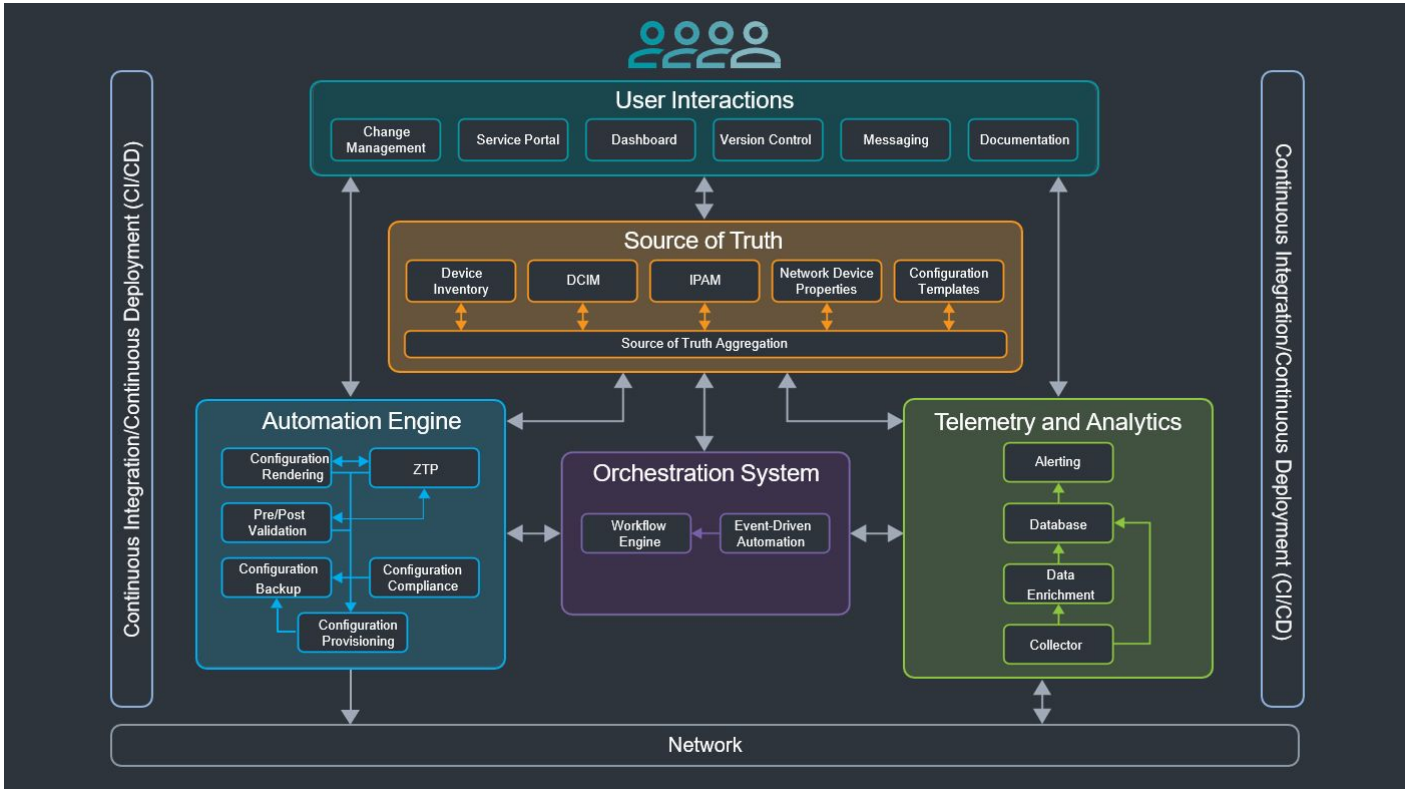
>>> What is Nautobot?

Nautobot is a source of truth *that is the foundation to enable network automation. Nautobot is an extensible network automation platform that allows for the customization of existing data models, creation of new data models, and serves as a* ***platform for network automation apps****.*

This presentation will explore Nautobot as a platform for network automation Apps.

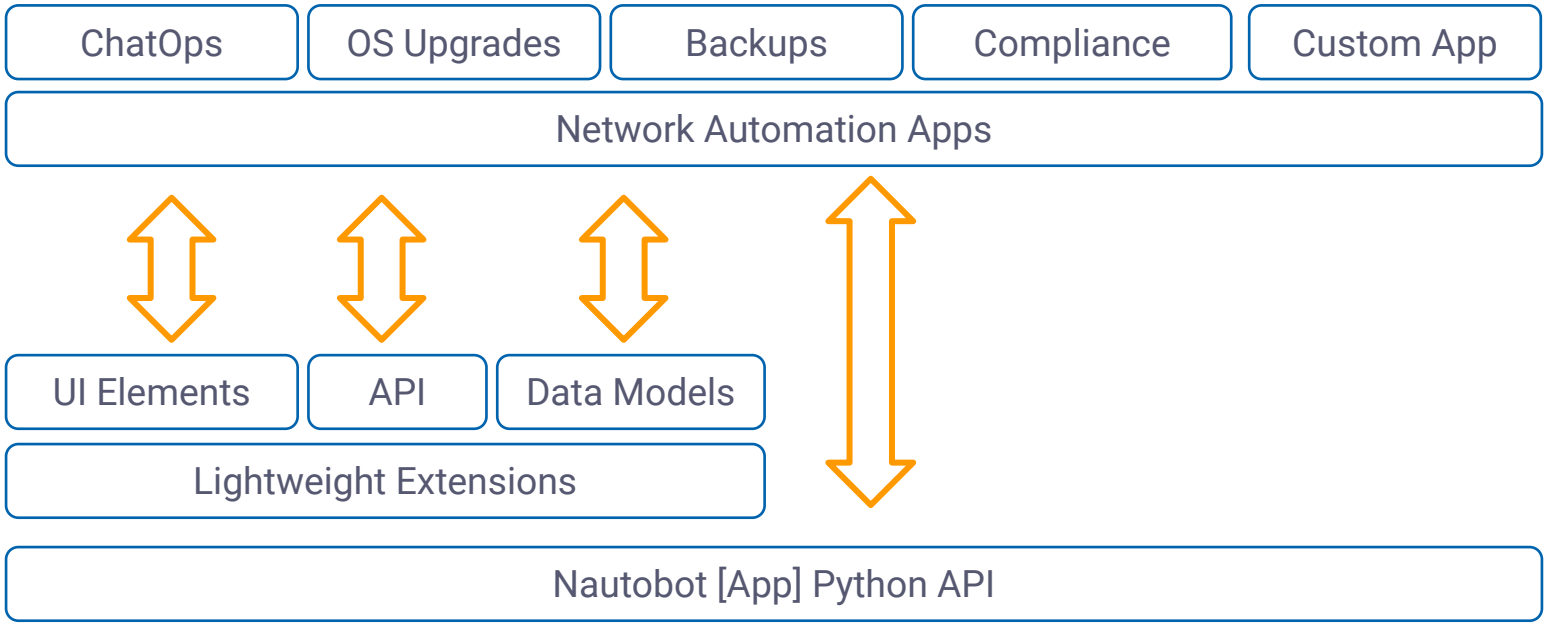
>>> Network Automation with Nautobot

Source of Truth Data Enables Network Automation



Nautobot App Platform as an Enabler

The Nautobot plugin system allows users to add extensions and robust applications to serve as a platform to enable network automation.



>>> Nautobot is a Network Automation Application Platform

The Nautobot App Platform provides an architecture that enables automation application development

Save 65-75% Development Time

Nautobot-Specific Use Cases

- Customize Nautobot UI
- Create APIs
- Extend data models
- Integrate external tools (like ServiceNow CMDB)
- Build automation processes

Automation Platform Use Cases

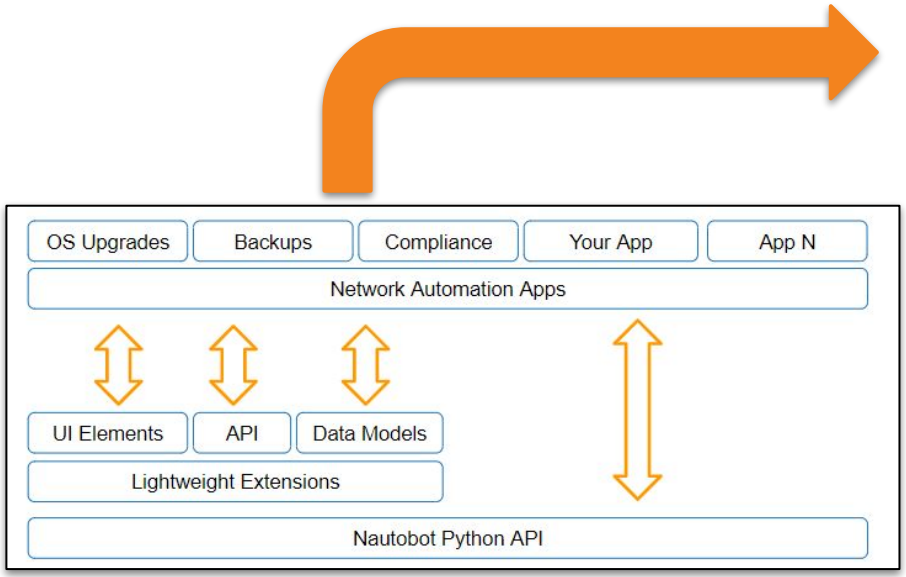
- Apps are not restricted to Nautobot-specific use cases
- Ex: the Nautobot platform can host various chatbots (Ansible AWX, CRM, CMDB, etc), perform backups, or execute any workflow

Advantages

- Flexible
- Allows for lightweight or complex apps
- Leveraging the architecture saves development time
- Leverage SoT data

>>> An Application Ecosystem

Nautobot's **Extensible Application Platform** and rich Source of Truth data **enable** a variety of **Applications** with a wide range of capabilities



go.nautobot.com/apps

Featured Apps

Change Management
Allows users to have change (workflow) management with approvals when managing data within Nautobot.
Dolt • Coming Soon • 1.2.0
✓ Commercial Support Available
✓ Open Source

Golden Configuration
Automate configuration backups, perform configuration compliance, and generate intended configurations.
Network to Code • February 24, 2021 • 1.0.0
✓ Commercial Support Available
✓ Available in Nautobot Demo Instance
✓ Open Source

Nautobot ChatOps
Adds a chatbot to Nautobot so you can easily get data from Nautobot directly from chat.
Network to Code • February 24, 2021 • 1.0.0
✓ Commercial Support Available
✓ Open Source
✓ Available in Nautobot Demo Instance

Apps & Solutions

Ansible ChatOps
Perform common Ansible AWX/Tower operations using ChatOps.
Network to Code • July 15, 2021 • 1.1.0
✓ Commercial Support Available
✓ Open Source

Arista CloudVision ChatOps
Perform common CloudVision operations using ChatOps.
Network to Code • August 3, 2021 • 1.1.0
✓ Commercial Support Available
✓ Open Source

Arista CloudVision SSoT
Synchronize key data between Nautobot and Cloud Vision.
Network to Code • August 3, 2021 • 1.1.0
✓ Commercial Support Available
✓ Open Source

Capacity Metrics
Exposes key data in Nautobot as Prometheus endpoints to be later consumed and visualized in tools like Grafana.
Network to Code • February 24, 2021 • 1.0.0
✓ Commercial Support Available
✓ Available in Nautobot Demo Instance
✓ Open Source

Change Management
Allows users to have change (workflow) management with approvals when managing data within Nautobot.
Dolt • Coming Soon • 1.2.0
✓ Commercial Support Available
✓ Open Source

Circuit Maintenance
Helps manage and view maintenances for circuits directly in Nautobot.
Network to Code • May 7, 2021 • 1.0.0
✓ Commercial Support Available
✓ Open Source
✓ Available in Nautobot Demo Instance

Nautobot App Capabilities Summary

Understanding the App (plugin) API

>>> Nautobot App Capabilities

Extending & Customizing Nautobot

Customized Navigation & UI

- Add top-level navigation menu tabs
- Add menu groups beneath *Plugins* navigation menu tab
- Add custom URL navigation to new pages
- Add new pages with custom layouts

Customized Content & UI Presentation

- Add content panels to existing object detail view pages
- Add button(s) to existing object detail pages
- Add banners in Nautobot UI
- Add panels to the Nautobot home page
- Add Jinja2 filters for customized content rendering

Data Models

- Define new data Models
- Define custom validators (validation logic) for Existing Data Models

Create custom REST API Endpoints

Add Custom Content Handlers

Package and Install Jobs With Dependencies

Full GraphQL Integration

Add additional Django Middleware to alter Nautbot's input/output

Create new management commands

Leverage Nautobot features inside Apps, including

- Custom fields
- Webhooks
- Relationships

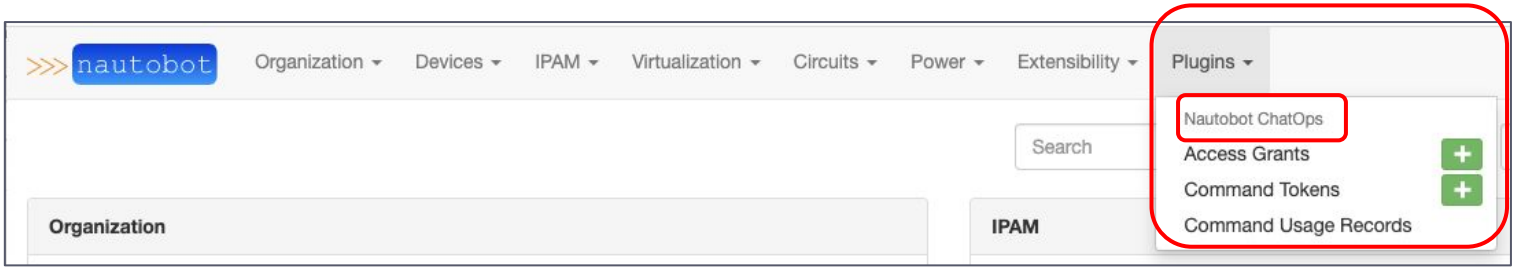

Customized Navigation & UI

>>> Navigation Menu Items

App-Defined Menu Navigation

Apps can add new links to the navigation menu under the *Plugins* top-level menu item

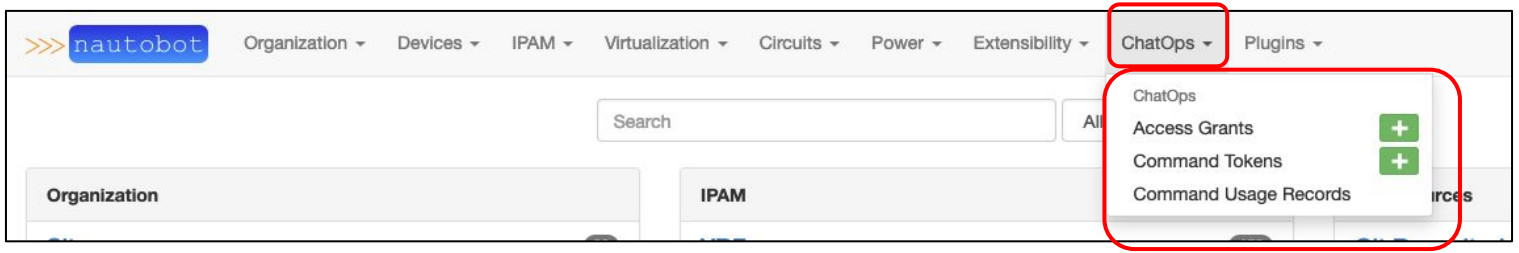
- **Nautobot ChatOps** application's Menu Group under the *Plugins* Tab



Example

Starting in 1.1.0, App developers can add Tabs, Groups, and Buttons in the top-level navigation menu

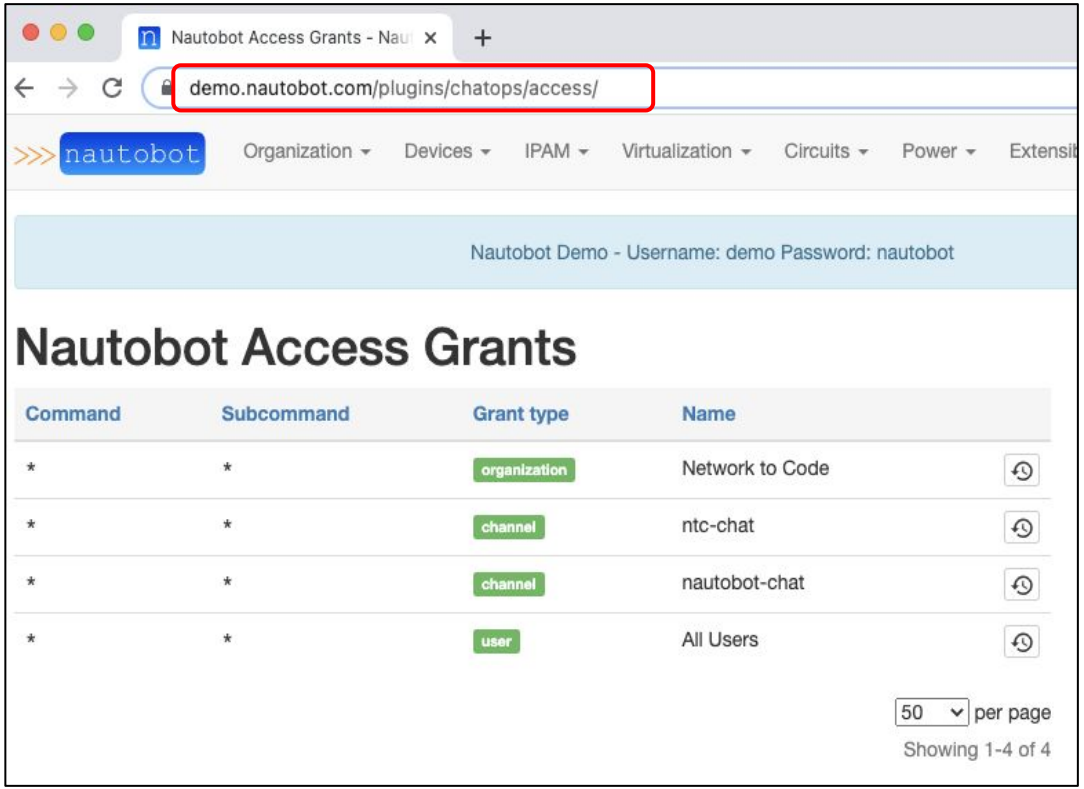
- **ChatOps** App menu as a **Top-level menu Tab**



>>> New URLs and Page Views

App-Defined URL Navigation

- Apps can create new Views (pages) with custom URLs under the https://<nautobotserver>**/plugins** root path
- This example shows a *ChatOps* plugin URL for access grants under the /plugins/chatops/ custom URL endpoint
- This example also shows a **customized page** with custom styling view for the ChatOps app's Access Grants page



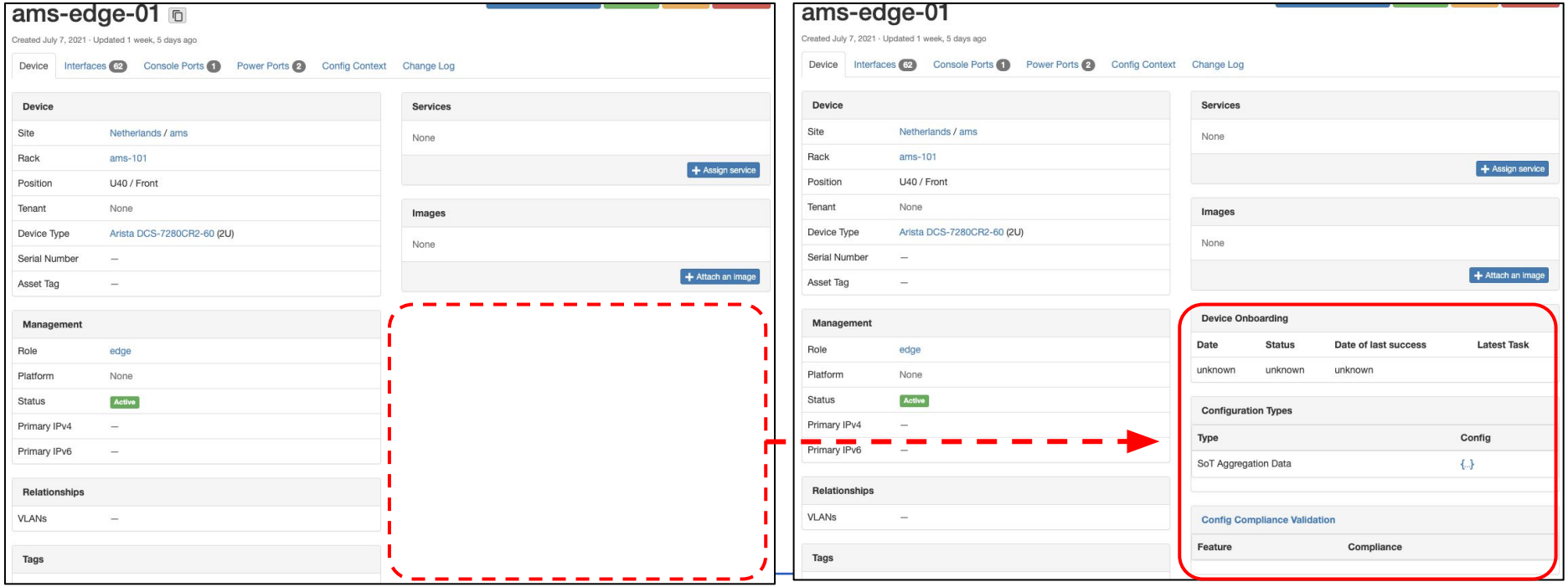
Customized Content Presentation

Inject Content to Current Pages

>>> Add Content Panels to Existing Pages

Apps can add custom content to **existing *Detail View* pages**

- The custom content can appear on the left side, right side, or bottom of the page
- In this example, Apps are adding three additional panels to the Device *Detail View*

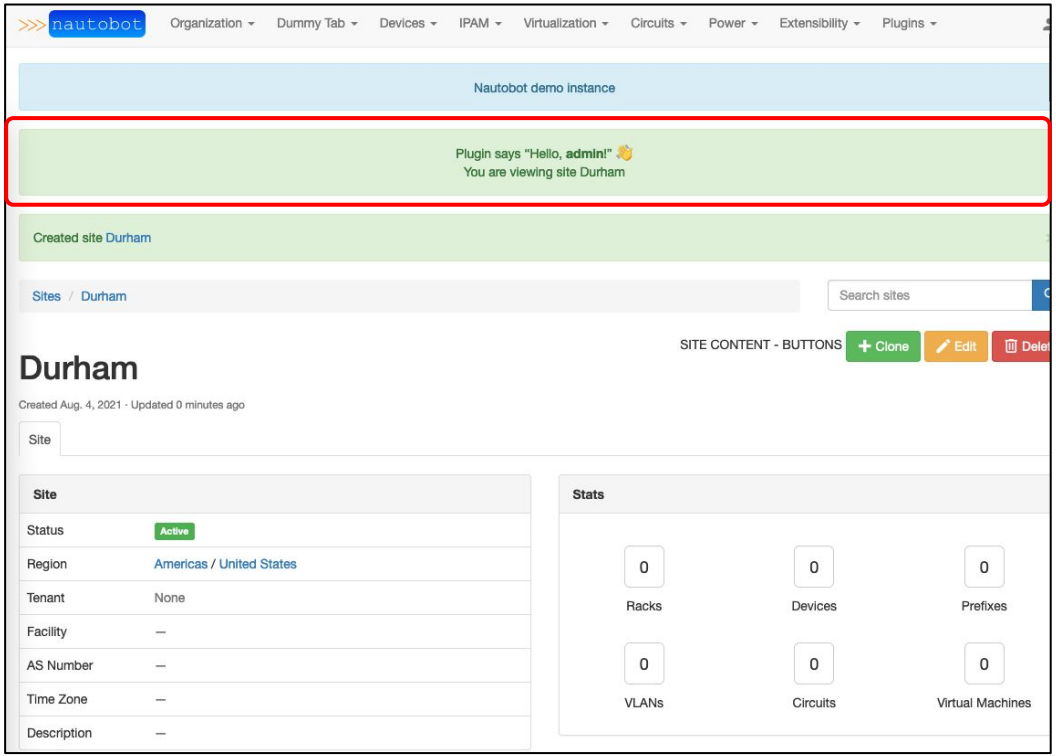


>>> Add Banners (coming in 1.2)

Apps can add banners to display specific and/or important information to the user.

Examples include:

- Reminders
- Change window information
- Maintenance Notifications
- Last known network automation task or execution



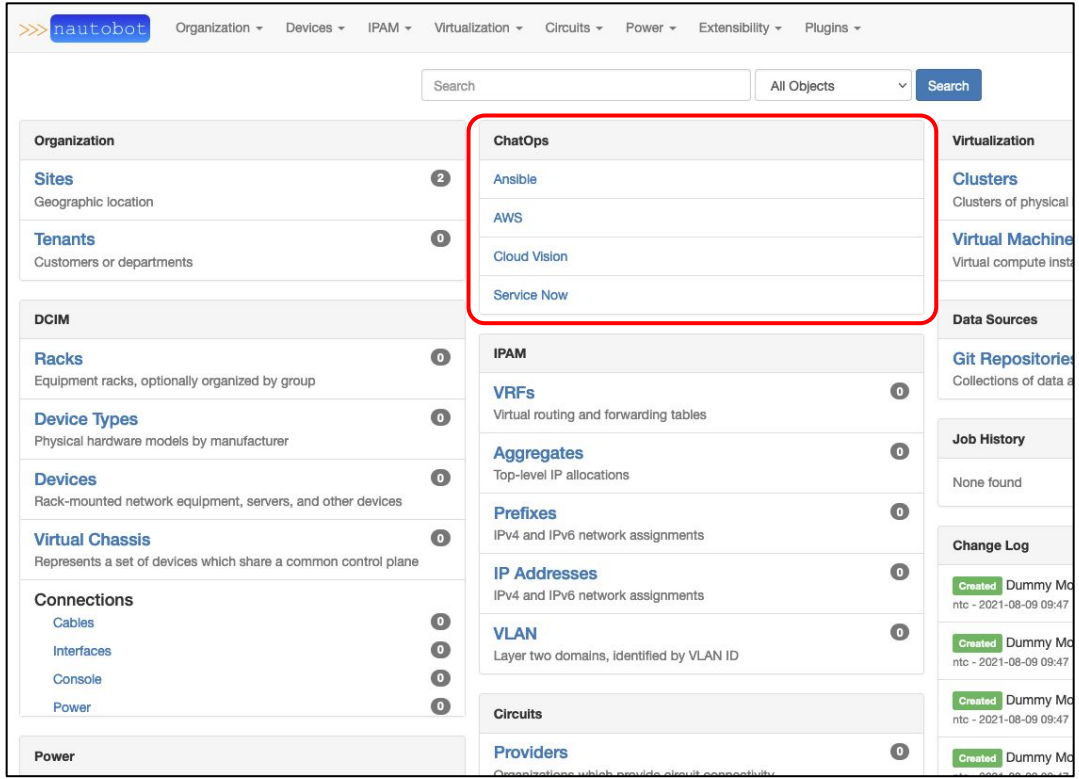
>>> Add Content Panels to the Nautobot Home Page (coming in 1.2)

Nautobot apps can add custom content panels on the **Nautobot home page**

This allows users to prominently display important app capabilities

Examples

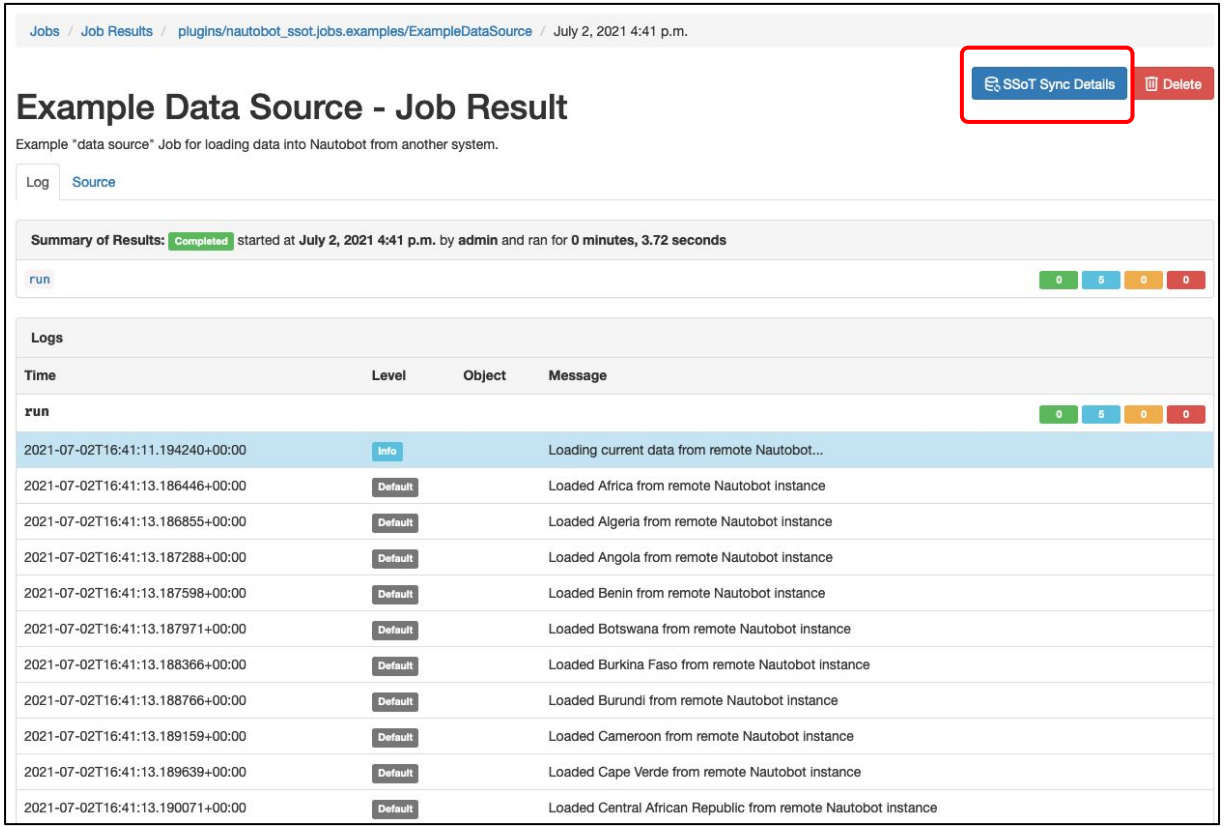
- Add ChatOps data to the homepage
- Add last devices automated
- Add devices that are failing to respond to tooling
- Add devices that aren't onboarded to specific tools



>>> Add button(s) to Detail View Pages

Customized Page Functionality

- Apps can **add buttons** to object *detail* pages
- This example shows the added **SSoT Sync Details** button on a Job Result page



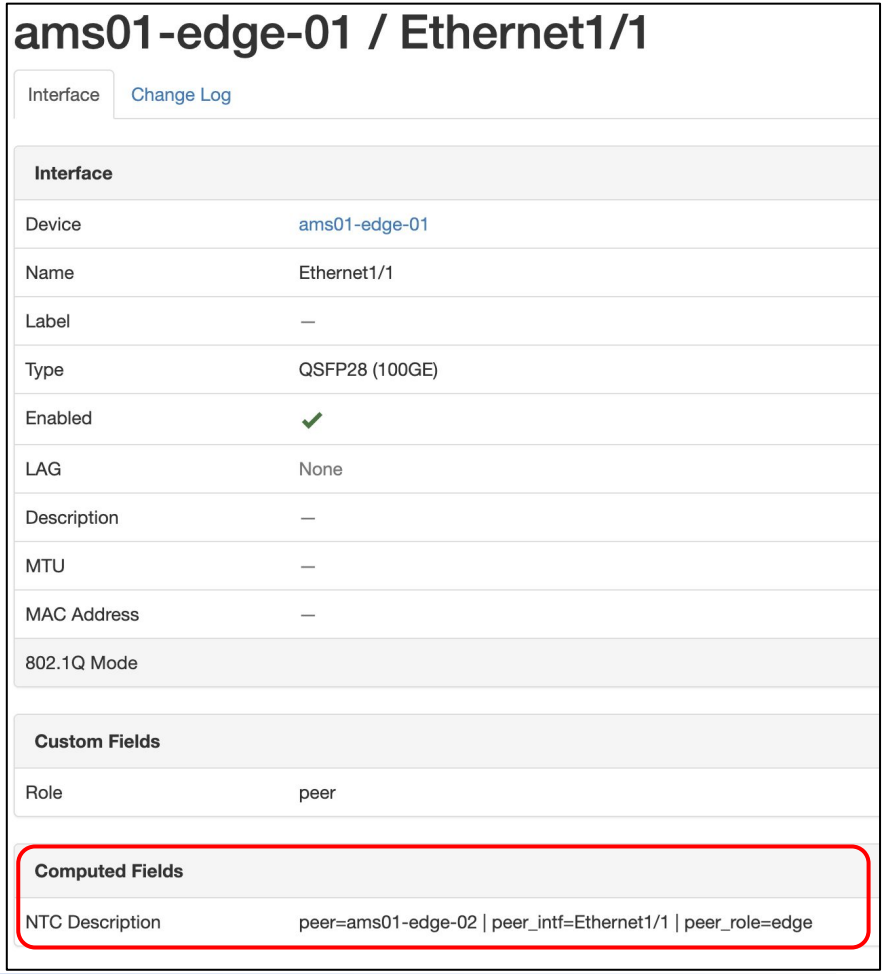
>>> Define and Register Jinja2 Filters

Apps allow developers to define custom Jinja2 filters to customize data presentation

Jinja2 filters can be used in

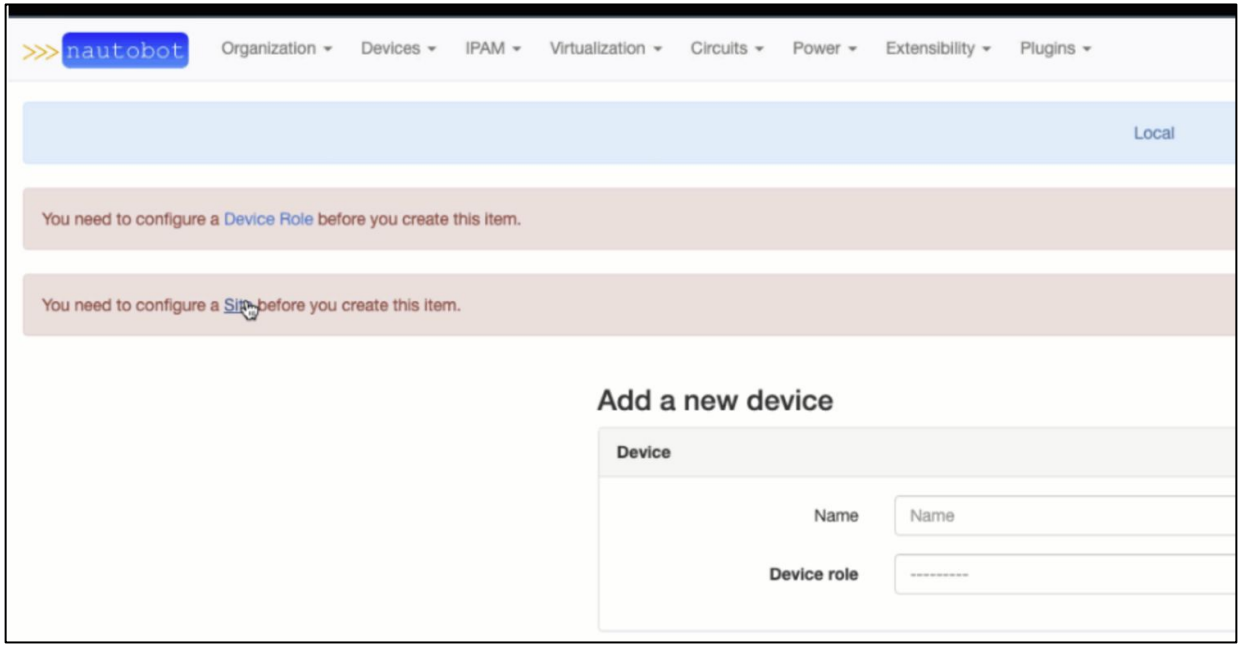
- Computed fields
- Custom links
- Webhooks
- Export templates

Example: this Computed Field shows output generated by an App's Jinja2 filter



Add Additional Django Middleware

- There may be times when an application requires modifying Nautobot's default input/output.
- Apps can specify Django middleware to load, in addition to Nautobot's built-in middleware
- This example, from the *Nautobot Welcome Wizard App*, shows custom middleware detecting certain conditions and displaying multiple banners in response



Data Models

Create & Extend Data Models

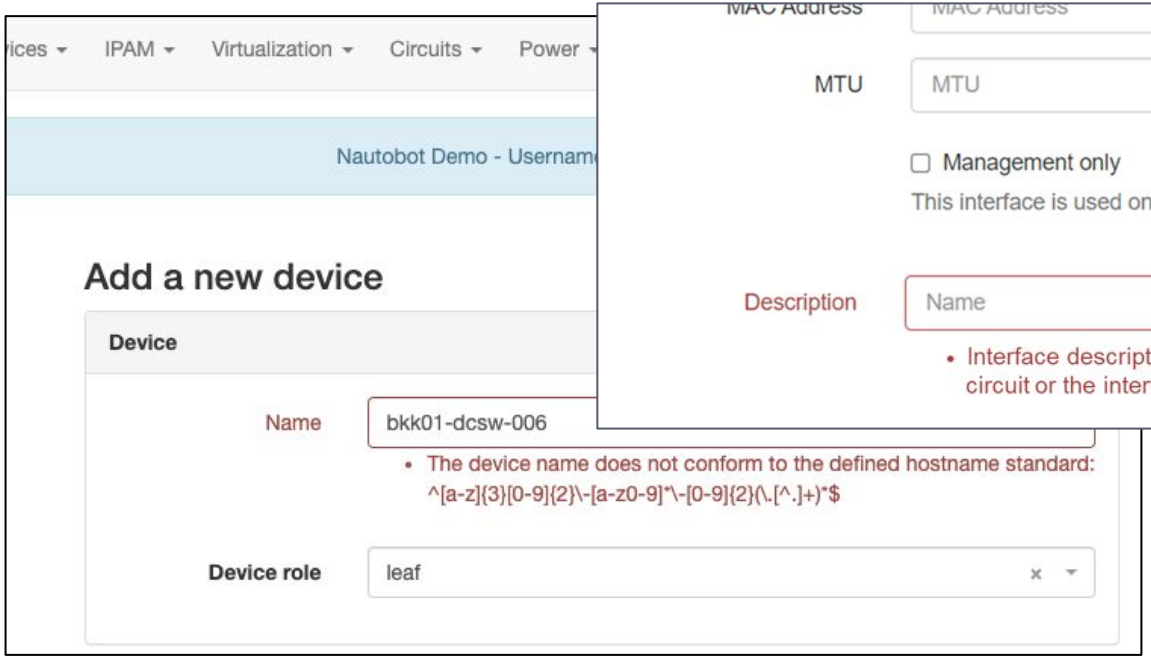
>>> Apps Can Define New/Custom Data Models

- Nautobot Apps can define and leverage new/custom data Models leveraging Django's ORM framework, optimizing performance and reducing risk
- This example shows part of the application-defined **CircuitMaintenance** data Model, from the *Circuit Maintenance* Nautobot Application.
- Common examples include configuration features not natively supported: EVPN/VXLAN, OSPF, SNMP, ACLs.

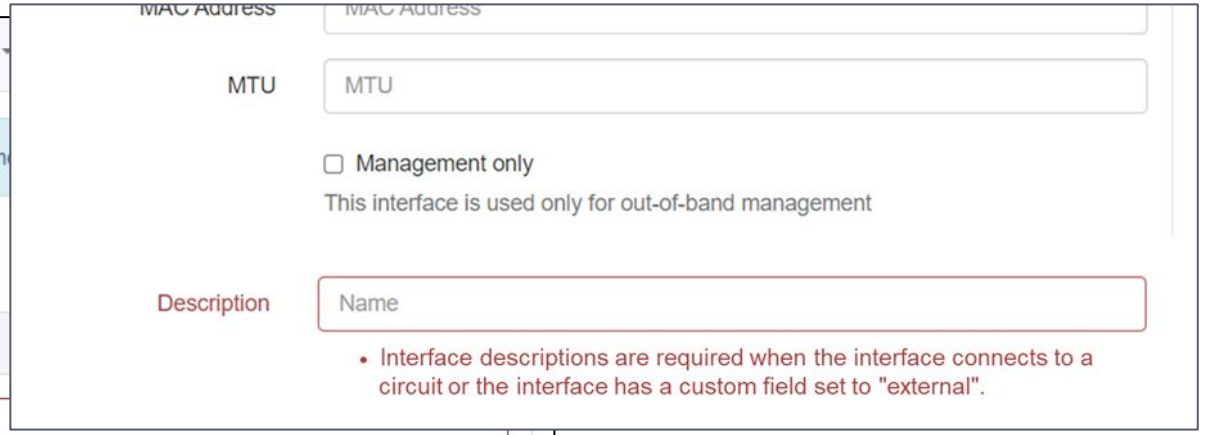
```python
class CircuitMaintenance(PrimaryModel):
    """Model for circuit maintenances."""

    name = models.CharField(max_length=100, default="", unique=True, blank=False)
    start_time = models.DateTimeField()
    end_time = models.DateTimeField()
    description = models.TextField(null=True, blank=True)
    # TODO: Status could use the new general Status model
    status = models.CharField(
        default=CircuitMaintenanceStatusChoices.TENTATIVE,
        max_length=50,
        choices=CircuitMaintenanceStatusChoices,
        null=True,
        blank=True,
    )
    ack = models.BooleanField(default=False, null=True, blank=True)

    csv_headers = ["name", "start_time", "end_time", "description", "status", "ack"]

    class Meta:  # noqa: D106 "Missing docstring in public nested class"
        ordering = ["start_time"]

    def __str__(self):
```

>>> Custom Validation Logic for Existing Data Models

An App can provide additional logic that adds constraints to attributes for existing and/or new objects



Example #1

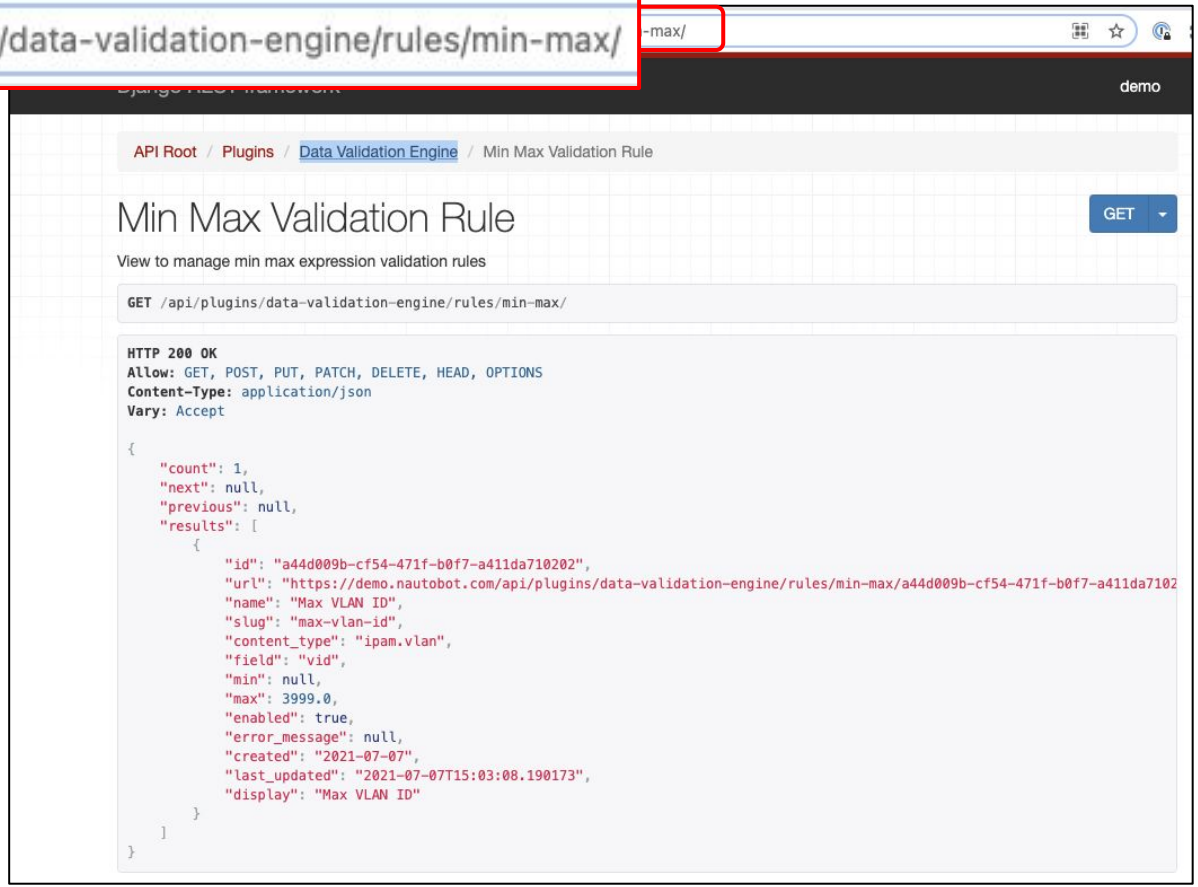


Example #2

App REST API Endpoints

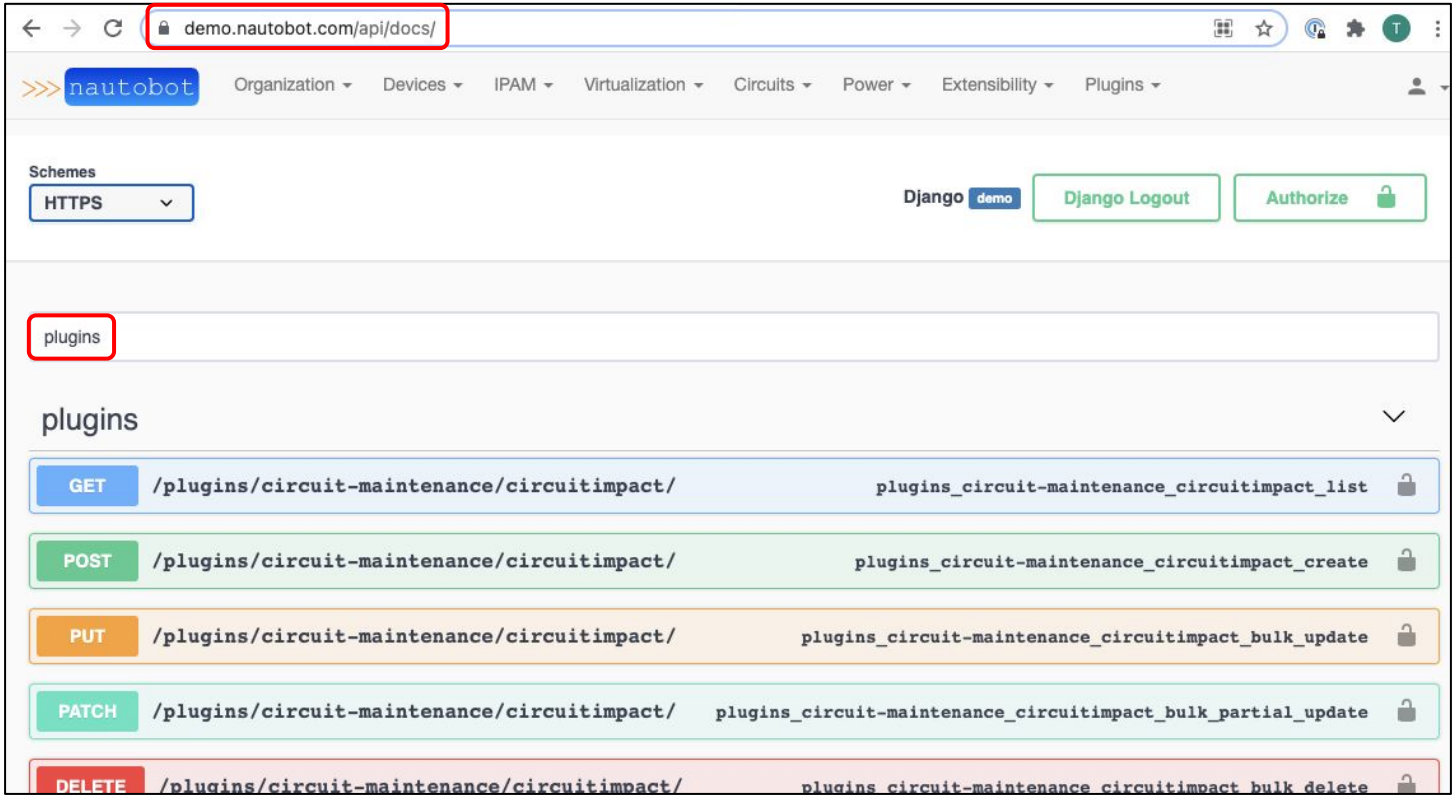
>>> REST API Endpoints for Nautobot Apps

- Apps can define custom API endpoints under the **/api/plugins/** root path to provide new REST API views
- This example is showing an API created by the Data Validation plugin



>>> REST API Endpoints for Apps (continued)

A listing of all available API calls for Apps is at the **/api/docs/** root, then search for *plugins*



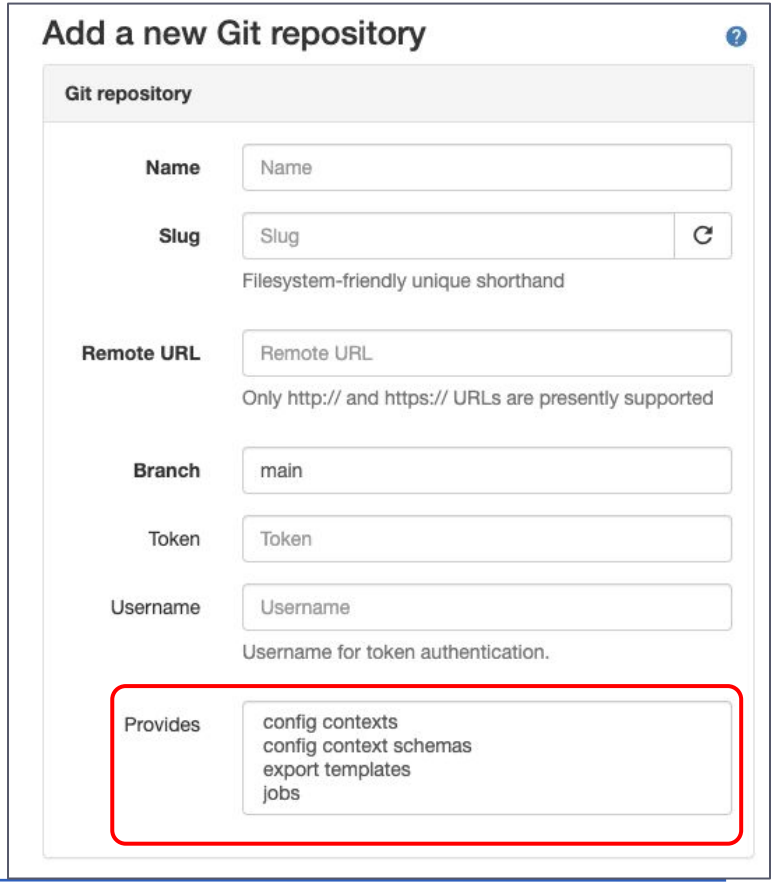
Adding Custom Content Handlers

Integration Git-stored Data into Nautobot

About Git Content Handlers

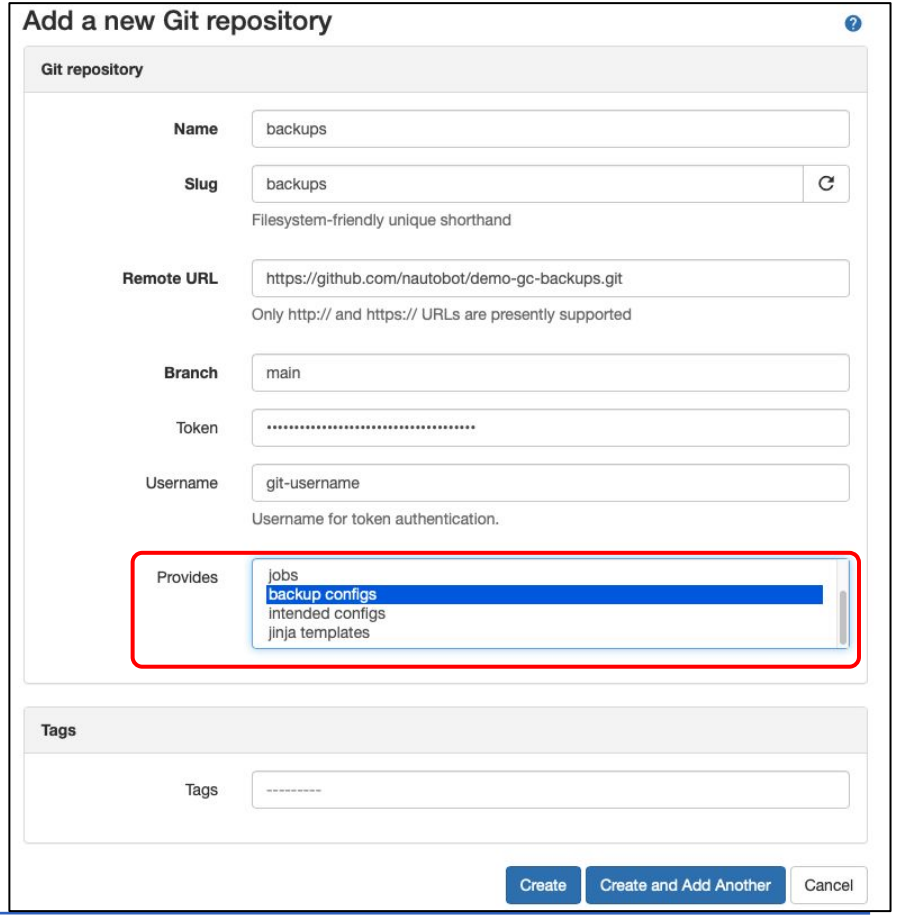
By default, Nautobot allows a new Git repository to provide:

- Config contexts
  - Allows Nautobot to store arbitrary JSON data which can be used for different automation tooling
- Config context schemas
  - Enforce data validation of *config contexts*
- Export templates
  - Allow user to export Nautobot objects based on custom template
- Jobs
  - Execute custom logic/code on demand from Nautobot UI



>>> Add Git Content Handlers (continued)

- Nautobot Apps can add additional, custom content handlers
- This allows App developers to customize content handling for the data type(s) that the business requires
- The additional content handlers ("providers") are available once the user installs the App




Nautobot Jobs
Simplify Job Management

>>> Package and Install Jobs With Dependencies

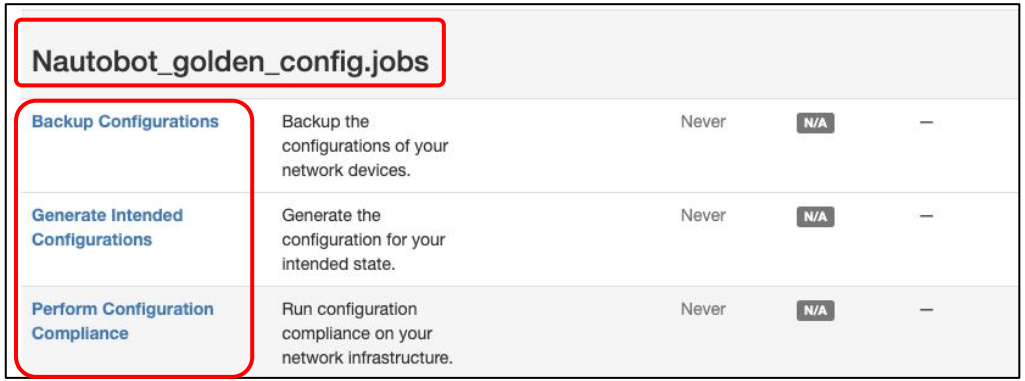
Apps can be a convenient way to package and install Jobs

- Jobs are a way for users to execute custom logic/code, on demand, from within the Nautobot UI or via an API call

Jobs that are packaged and installed via Apps can

- Import modules from other parts of the Application
- Specify and install additional Python dependencies for the Job

This example shows three jobs installed by the Nautobo *Golden Config* App



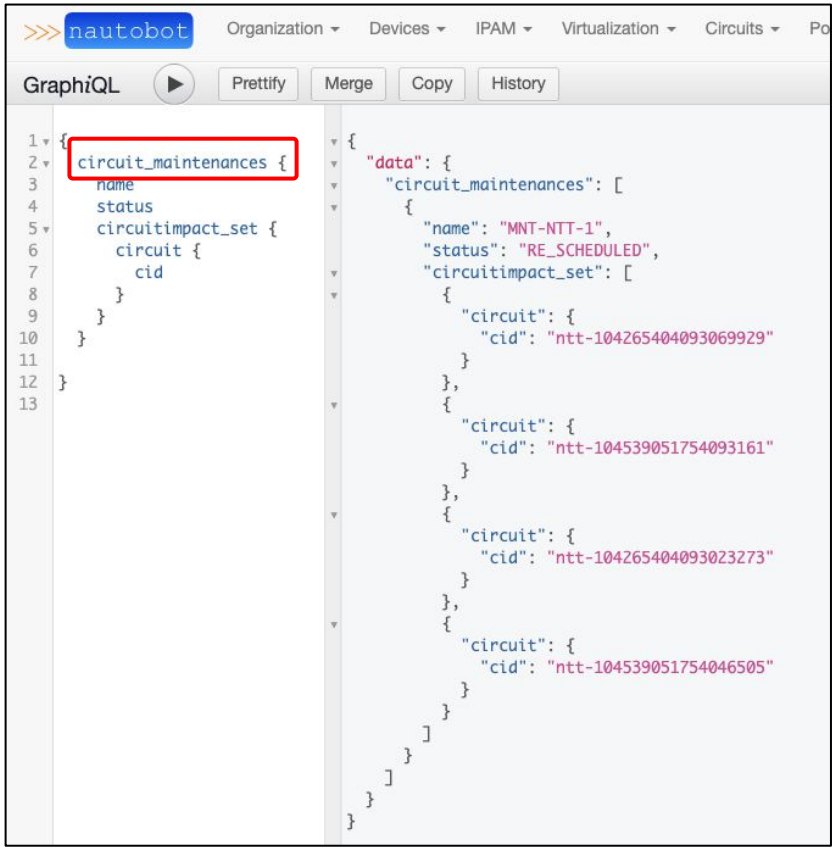
GraphQL Integration

Extend the GraphQL API with Apps

GraphQL Integration

- Application-defined custom Models can be used in GraphQL queries

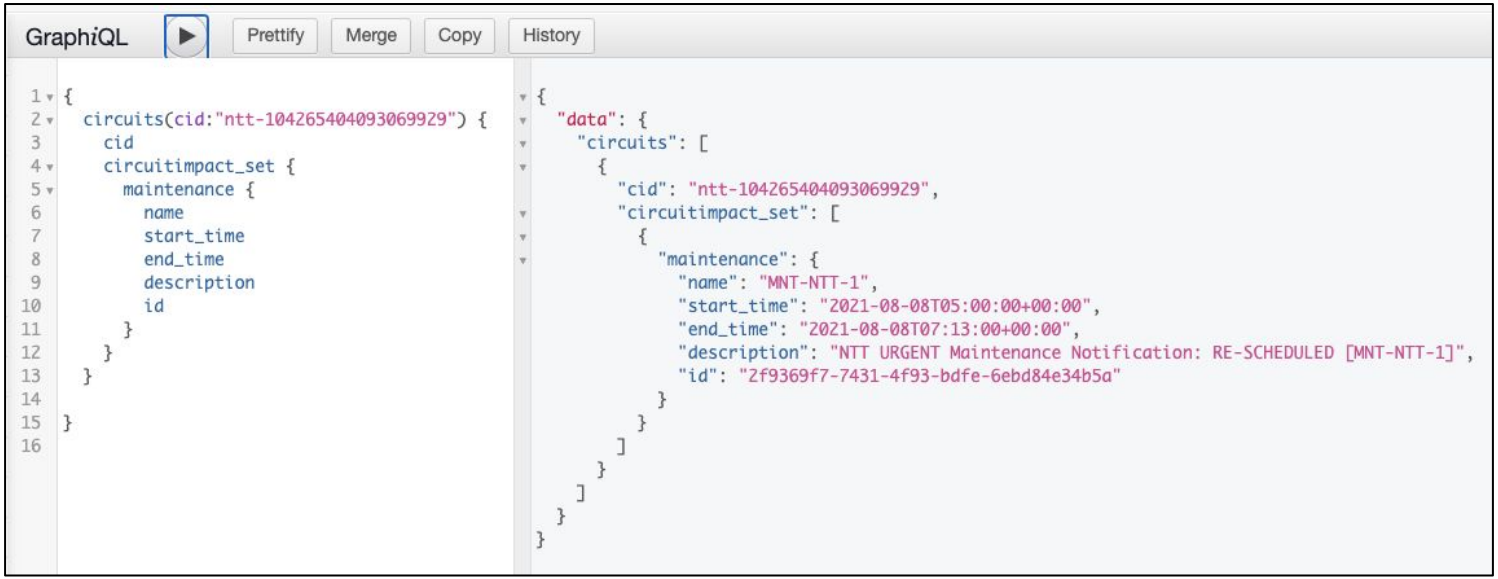
- This example queries for instances of a **CircuitMaintenance** Model defined in the *Circuit Maintenance* app



>>> GraphQL Integration (continued)

- An app's custom Models can also be queried for as attributes of Nautobot Core Models by creating a relationship between them

- The example below queries for a Circuit object that is a member of any (app-defined) **CircuitImpact** instances

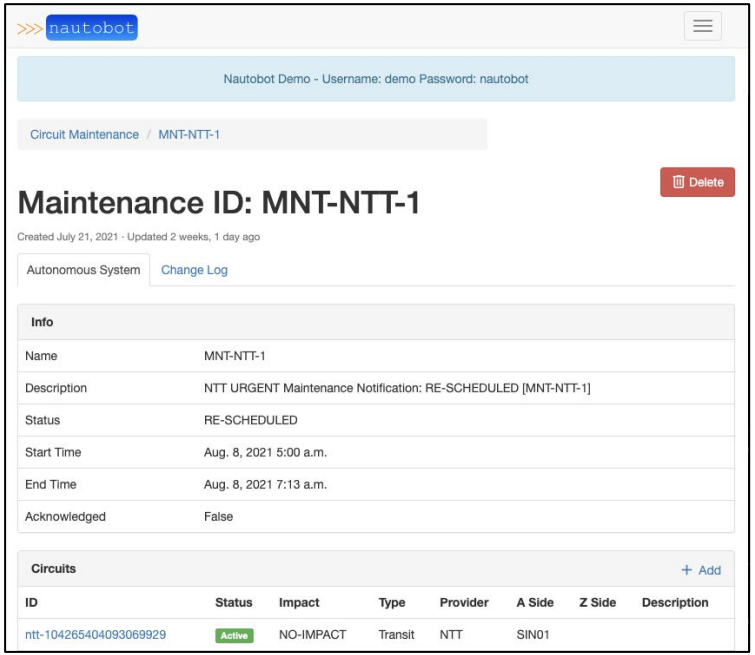



Leverage Nautobot **Extras** Features

>>> Nautobot Extras Features

Custom models defined in applications can leverage Nautobot's *extras* features

These features include *relationships*, *webhooks*, and *custom fields*

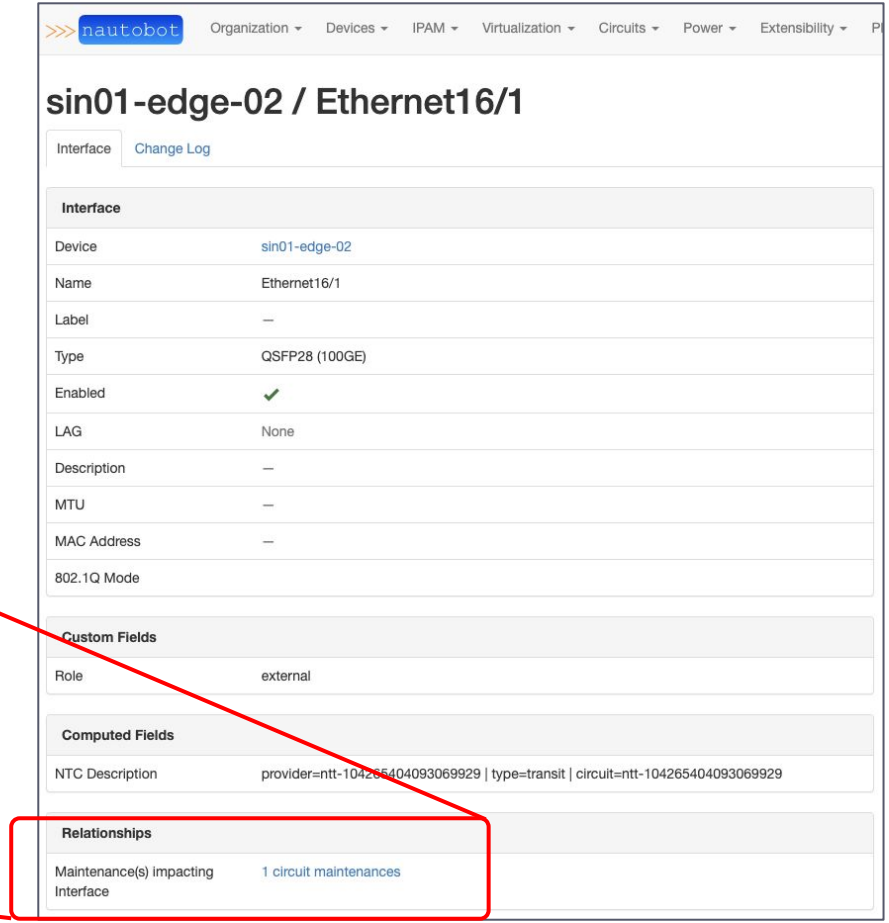



```python
class CircuitMaintenance(PrimaryModel):
    """Model for circuit maintenances."""

    name = models.CharField(max_length=100, default="", unique=True, blank=False)
    start_time = models.DateTimeField()
    end_time = models.DateTimeField()
    description = models.TextField(null=True, blank=True)
    # TODO: Status could use the new general Status model
    status = models.CharField(
        default=CircuitMaintenanceStatusChoices.TENTATIVE,
        max_length=50,
        choices=CircuitMaintenanceStatusChoices,
        null=True,
        blank=True,
    )
    ack = models.BooleanField(default=False, null=True, blank=True)
```

>>> Relationships

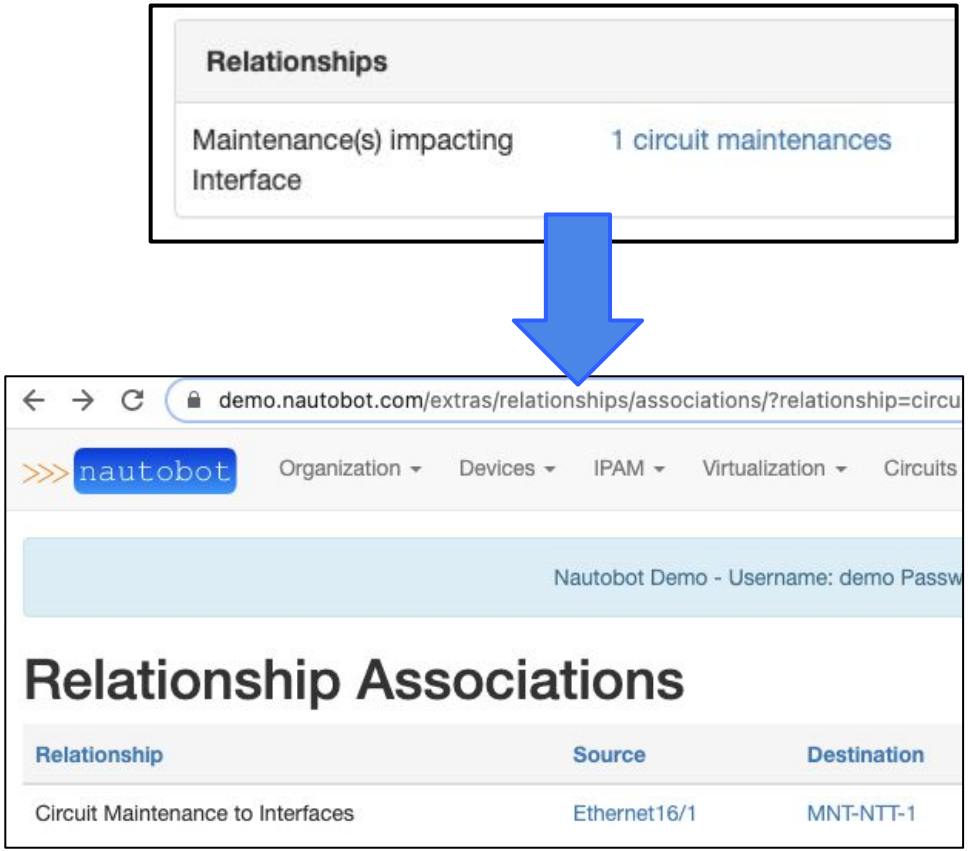
- Nautobot's *Relationships* feature can reference the app-defined Model

- This example shows this device's Interface has one circuit maintenance associated with it



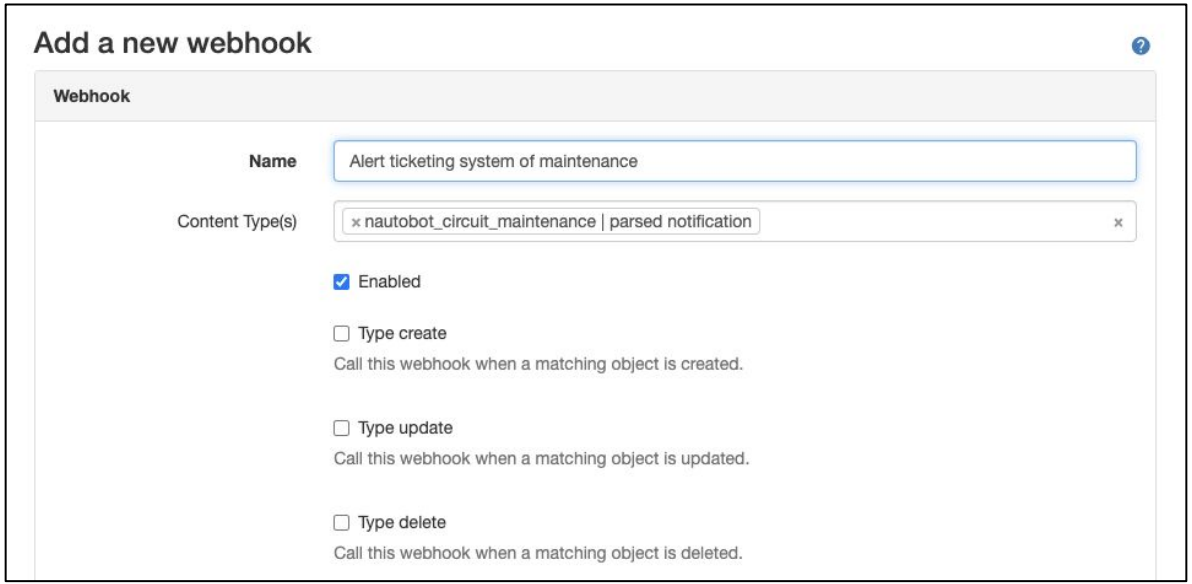
>>> Relationships (continued)

- The circuit maintenances link on the interface detail page links to the specific *interface-maintenance* relationship associations



>>> Webhooks

- Models created by Apps can leverage Nautobot's webhook feature to alert external systems when create/update/delete changes occur in app-defined Model instances
- The example here shows webhook creation for the **ParsedNotification** Model, defined in the *Circuit Maintenance* Application



Webhooks

- Application-defined Models can leverage Nautobot's webhook feature
- Nautobot webhooks allow users to have Nautobot make outbound APIs on specific events: Create, Update, and Delete
- Notify 3rd party systems when the Source of Truth data changes
- Common use cases: Trigger automation jobs, Chat alerts, ITSM notifications

Nautobot triggers outbound API call (webhook)

>>> Webhook Example

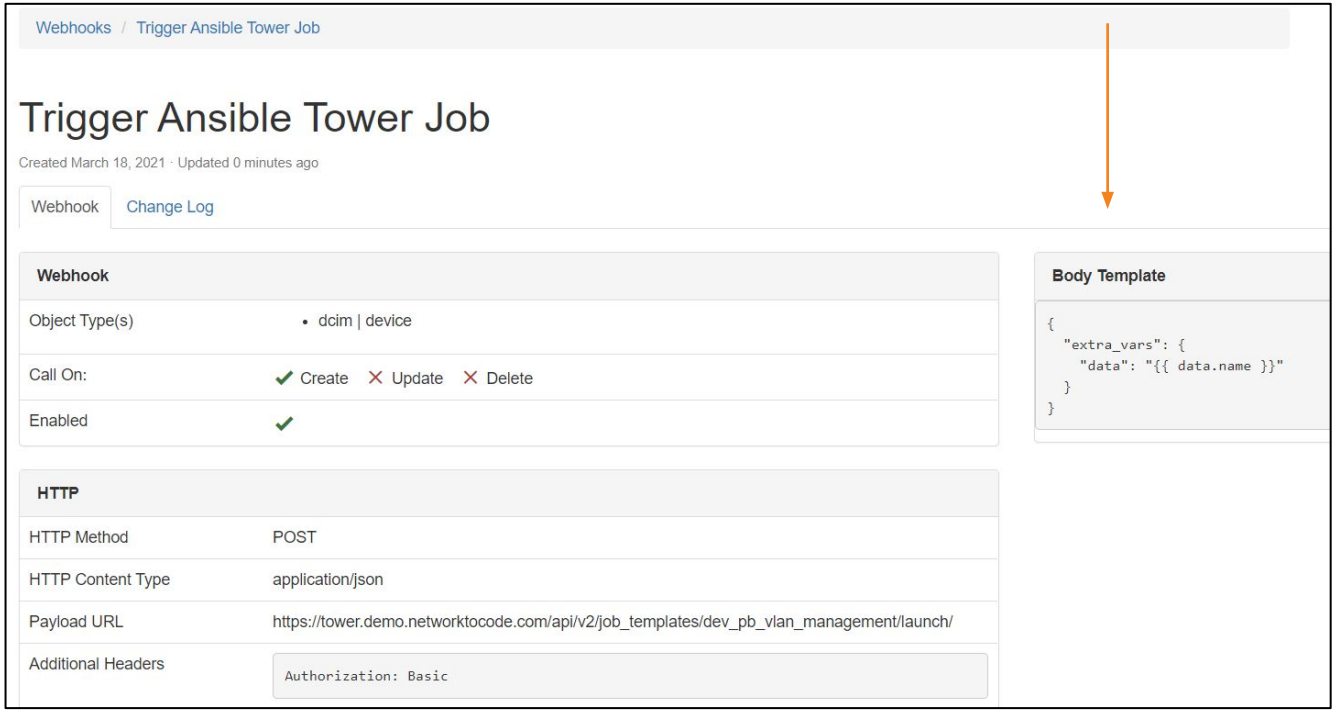
Execute Ansible Tower Job (Playbook) When Device is Created

Jinja2 template support for the webhook body

Specify Model

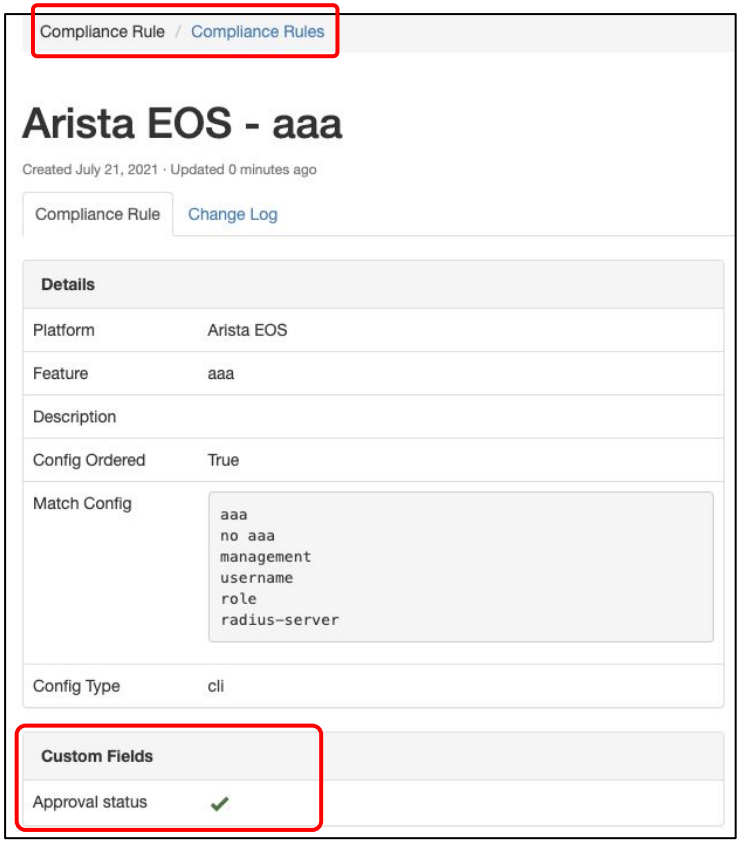
Decide when webhook is triggered

Configurable HTTP Methods



>>> Custom Fields

- App-defined Models can accommodate Nautobot's custom fields
- Custom fields allow users to store custom data, specific to their operational or business needs, within objects
- Custom fields allow users to further customize the App-defined Models
- This example shows and *Approval status* custom field on the **ComplianceRule** Model from the *Golden Config* application



Custom Management Commands

>>> Add Custom Management Command

The `nautobot-server` utility serves as the single point of entry for most administrative tasks in Nautobot

Nautobot apps can create new management (`nautobot-server`) commands used at the Nautobot server CLI to simplify administration by leveraging the existing `nautobot-server` entry point

```
nautobot@say-my-name:~$ nautobot-server run_backup
```


>>> network .toCode()
Thank you!

Backup Slides

Plugin Limitations

>>> Applications May Not Modify Nautobot Core Components . . .

This includes

- Core Models
  - Apps may not alter, remove, or override core Nautobot models in any way
- Core Templates
  - Apps can inject additional content where supported, but may not manipulate or remove core content
- Core Settings
  - A configuration registry is provided for Apps, however they cannot alter or delete the core configuration
- Core Components
  - Nautobot core components may not be disabled or hidden
- URL space
  - Appss may not register URLs outside of the `/plugins` root

>>> . . . HOWEVER . . .

Despite the Application limitations in the prior slide, Nautobot's extensible architecture still allows heavy customization options for Core Models:

- Custom Fields - Allow users to store additional, customized object attributes for Core Models
- Custom Validators - basic customized validation for custom field values
- Relationships - Define specific links between object Models to reflect business logic or other useful associations

Technical Slides

>>> Provisioning Plugin URLs and Views

The code snips below correspond to the example on the prior slide

- The application's __init__.py file specifies the app's base URL as `/chatops`
- The application's urls.py file defines a new View (page) under `/chatops/access/`

__init__.py

```
8   class NautobotChatOpsConfig(PluginConfig):
9       """Plugin configuration for the nautobot_chatops plugin."""
10
11      name = "nautobot_chatops"
12      verbose_name = "Nautobot ChatOps"
13      version = __version__
14      author = "Network to Code"
15      author_email = "opensource@networktocode.com"
16      description = "A plugin providing chatops capabilities."
17      base_url = "chatops"
```

urls.py

```
20  urlpatterns = [
21      path("", NautobotHomeView.as_view(), name="home"),
22      path("access/", AccessGrantListView.as_view(), name="accessgrant_list"),
```

>>> Add Git Content Handlers

Plugins can register additional types of data that a Git repository can provide

By default, Nautobot looks for an iterable named `datasource_contents` within the plugin's `datasources.py` file

This example to the right shows `datasource_contents` in the Golden Config Plugin's `datasources.py file` that can allow for repositories to provide one or more the following data types, depending on plugin options:

- Backup configs
- Intended configs
- Jinja templates

```python
datasource_contents = []
if ENABLE_INTENDED or ENABLE_COMPLIANCE:
    datasource_contents.append(
        (
            "extras.gitrepository",
            DatasourceContent(
                name="intended configs",
                content_identifier="nautobot_golden_config.intendedconfigs",
                icon="mdi-file-document-outline",
                callback=refresh_git_intended,
            ),
        )
    )
if ENABLE_INTENDED:
    datasource_contents.append(
        (
            "extras.gitrepository",
            DatasourceContent(
                name="jinja templates",
                content_identifier="nautobot_golden_config.jinjatemplate",
                icon="mdi-text-box-check-outline",
                callback=refresh_git_jinja,
            ),
        )
    )
if ENABLE_BACKUP or ENABLE_COMPLIANCE:
    datasource_contents.append(
        (
            "extras.gitrepository",
            DatasourceContent(
                name="backup configs",
                content_identifier="nautobot_golden_config.backupconfigs",
                icon="mdi-file-code",
                callback=refresh_git_backup,
            ),
        )
    )
```

>>> Plugin-Defined Models Can Leverage Core Nautobot Features

Plugin-defined Models can leverage core implementations of GraphQL, webhooks, logging, custom relationships, custom fields, and tags

Example shown is from the Nautobot Golden Config plugin:

https://github.com/nautobot/nautobot-plugin-golden-config/blob/18d03df21866474ab557fef2eb248b746e217f54/nautobot_golden_config/models.py#L34

```
34  @extras_features(
35      "custom_fields",
36      "custom_validators",
37      "export_templates",
38      "relationships",
39      "graphql",
40      "webhooks",
41  )
42  class ComplianceFeature(PrimaryModel):
43      """ComplianceFeature details."""
44
45      name = models.CharField(max_length=100, unique=True)
46      slug = models.SlugField(max_length=100, unique=True)
47      description = models.CharField(max_length=200, blank=True)
```

>>> REST API Endpoints for Plugins (continued)

The custom REST API endpoints for the plugin are defined in the app's `api/urls.py` file

This example below, from the Data Validation Application, shows how the following REST API endpoints are crafted:

- `/api/plugins/data-validation-engine/rules/regex/`
- `/api/plugins/data-validation-engine/rules/min-max/`

```
"""
API routes
"""
from nautobot.core.api import OrderedDefaultRouter

from nautobot_data_validation_engine.api import views


router = OrderedDefaultRouter()
router.APIRootView = views.DataValidationEngineRootView

# Regular expression rules
router.register("rules/regex", views.RegularExpressionValidationRuleViewSet)

# Min/max rules
router.register("rules/min-max", views.MinMaxValidationRuleViewSet)


urlpatterns = router.urls
```

>>> Add Callbacks When Repository is Refreshed

Plugins can define callbacks that notify when the repository has refreshed data

The example below shows defined callbacks from the Golden Config Plugin's `datasources.py`

```
8  def refresh_git_jinja(repository_record, job_result, delete=False):  # pylint: disable=unused-argument
9      """Callback for gitrepository updates on Jinja Template repo."""
10     job_result.log(
11         "Successfully Pulled git repo",
12         level_choice=LogLevelChoices.LOG_SUCCESS,
13     )
14
15
16 def refresh_git_intended(repository_record, job_result, delete=False):  # pylint: disable=unused-argument
17     """Callback for gitrepository updates on Intended Config repo."""
18     job_result.log(
19         "Successfully Pulled git repo",
20         level_choice=LogLevelChoices.LOG_SUCCESS,
21     )
22
23
24 def refresh_git_backup(repository_record, job_result, delete=False):  # pylint: disable=unused-argument
25     """Callback for gitrepository updates on Git Backup repo."""
26     job_result.log(
27         "Successfully Pulled git repo",
28         level_choice=LogLevelChoices.LOG_SUCCESS,
29     )
```

>>> Declare Configuration Parameters

Each plugin can define required, optional, and default configuration parameters within its unique namespace

Within the plugin's `__init__.py`, there is a PluginConfig subclass

The `default_settings` attribute is a dictionary of configuration parameters, along with the default value of each

The values for these parameters would be populated in the `nautobot_config.py` file

```
class NautobotChatOpsConfig(PluginConfig):
    """Plugin configuration for the nautobot_chatops plugin."""
    default_settings = {
        "enable_slack": False,
        "enable_ms_teams": False,
        "enable_webex_teams": False,
        # Should menus, text input fields, etc
        "delete_input_on_submission": False,
        # Slack-specific settings
        "slack_api_token": None,  # for example
        "slack_signing_secret": None,
        # Any prefix that's prepended to all s
        # in order to identify the actual comma
        "slack_slash_command_prefix": "/",
        # Microsoft-Teams-specific settings
        "microsoft_app_id": None,
        "microsoft_app_password": None,
        # WebEx-Teams-specific settings
        "webex_teams_token": None,
        "webex_teams_signing_secret": None,
        "enable_mattermost": False,
        # Mattermost-specific settings
        "mattermost_api_token": None,
        "mattermost_url": None,
    }
```

>>> Define and Register Jinja2 Filters

Plugins allow authors to define custom Jinja2 filters for use in

- Computed fields
- Custom links
- Webhooks
- Export templates

Import the `library` module from django_jinja library and then use `@library.filter` decorator on the filter:

```
8  @library.filter
9  def ntc_description(obj):
10     """Create an interface description dynamically."""
11     if obj.connected_endpoint and isinstance(obj.connected_endpoint, CircuitTermination):
12         circuit = obj.connected_endpoint.circuit
13         return f"provider={circuit.cid} | type={circuit.type.slug} | circuit={circuit.cid}"
14
15     elif obj.connected_endpoint and isinstance(obj.connected_endpoint, Interface):
16         interface = obj.connected_endpoint
17         return (
18             f"peer={interface.device.name} | peer_intf={interface.name} | peer_role={interface.device.device_role.slug}"
19         )
20
21     return ""
```

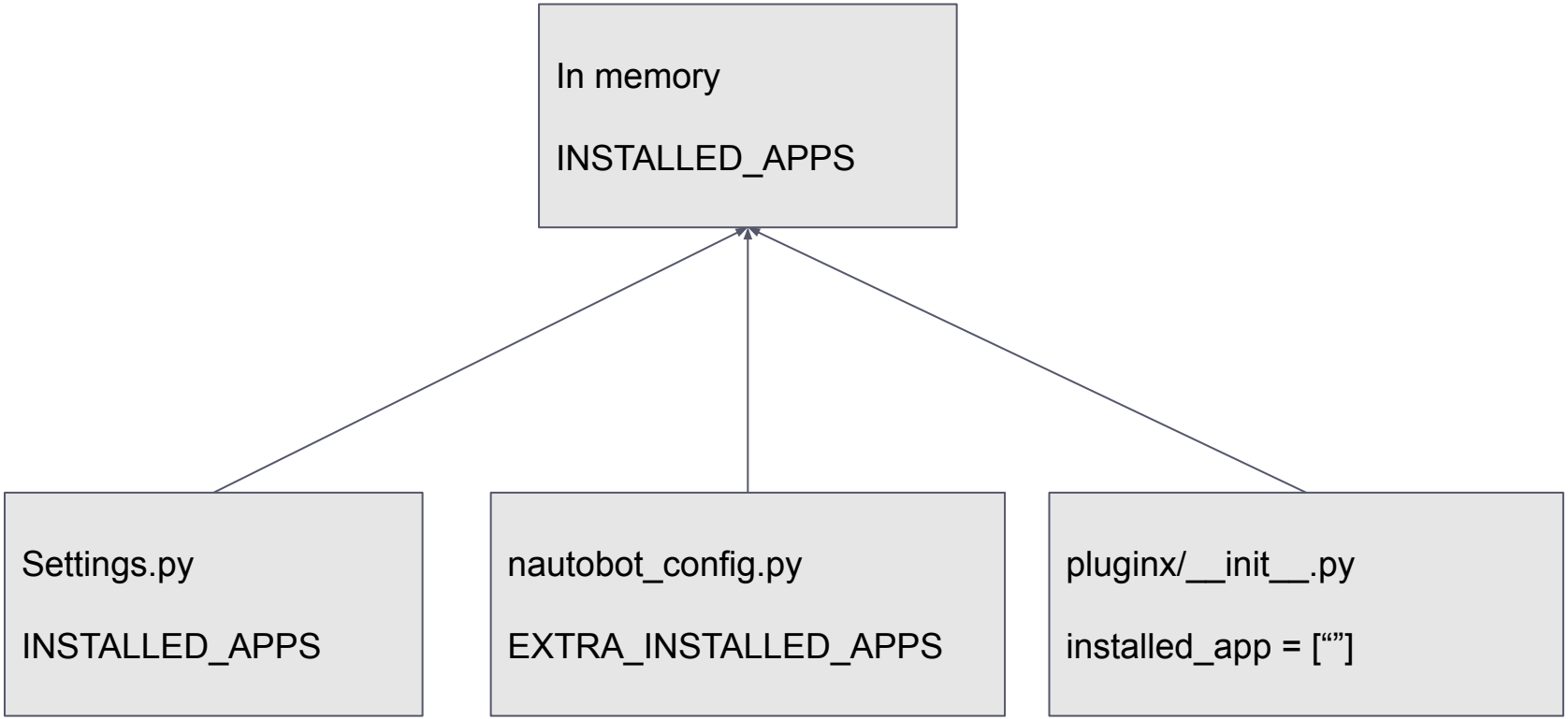
>>> Flexibility in Limiting Installation by Nautobot Version

Nautobot plugins can specify constraints on the Nautobot version

pyproject.toml

```
20  [tool.poetry.dependencies]
21  python = "^3.6"
22  napalm = ">=2.5.0, <4"
23  zipp = "^3.4.0"
24  nautobot = ">=1.0.0"
```

>>> How Installed Apps Get in Memory



>>> Custom Validation Logic for Existing Data Models

A plugin can provide additional logic that adds constraints to attributes for existing and/or new objects

```
# custom_validators.py
from nautobot.extras.plugins import PluginCustomValidator


class SiteValidator(PluginCustomValidator):
    """Custom validator for Sites to enforce that they must have a Region."""

    model = 'dcim.site'

    def clean(self):
        if self.context['object'].region is None:
            # Enforce that all sites must be assigned to a region
            self.validation_error({
                "region": "All sites must be assigned to a region"
            })


custom_validators = [SiteValidator]
```

>>> Add New (Plugin-Defined) Data Models

Plugins can introduce additional models to hold data

A model is essentially a Python representation of a database table

Example shown is from Nautobot ChatOps plugin:

https://github.com/nautobot/nautobot-plugin-chatops/blob/3f942cc07eec71cd07457c26ff9aafa357394f80/nautobot_chatops/models.py#L14

```python
class CommandLog(BaseModel):
    """Record of a single fully-executed Nautobot command.

    Incomplete commands (those requiring additional user input) should not be recorded,
    nor should any "help" commands or invalid command entries.
    """

    start_time = models.DateTimeField(null=True)
    runtime = models.DurationField(null=True)

    user_name = models.CharField(max_length=255, help_text="Invoking username")
    user_id = models.CharField(max_length=255, help_text="Invoking user ID")
    platform = models.CharField(max_length=64, help_text="Chat platform")
    platform_color = ColorField()

    command = models.CharField(max_length=64, help_text="Command issued")
    subcommand = models.CharField(max_length=64, help_text="Sub-command issued")
    params = ArrayField(
        ArrayField(models.CharField(default="", max_length=255)), default=list, help_text="user_input_parameters"
    )

    status = models.CharField(
        max_length=32,
        choices=CommandStatusChoices,
        default=CommandStatusChoices.STATUS_SUCCEEDED,
    )
    details = models.CharField(max_length=255, default="")

    @property
    def status_label_class(self):
```

>>> GraphQL and User-Defined Models

There are two ways to expose plugin-defined Models via GraphQL:

Method 1: Using the `graphql` parameter in the `@extras_features` decorator shown below

- Nautobot will detect this and will automatically create a GraphQL type definition based on the Model

```
34  @extras_features(
35      "custom_fields",
36      "custom_validators",
37      "export_templates",
38      "relationships",
39      "graphql",
40      "webhooks",
41  )
42  class ComplianceFeature(PrimaryModel):
43      """ComplianceFeature details."""
44
45      name = models.CharField(max_length=100, unique=True)
46      slug = models.SlugField(max_length=100, unique=True)
47      description = models.CharField(max_length=200, blank=True)
```

>>> GraphQL and User-Defined Models

Method 2: Creating a custom GraphQL type

- For certain instances when the auto-generated GraphQL type definition will not work
- See the example on the right

Only one of these two methods can be used per Model

38 lines (24 sloc) | 1.11 KB

```python
"""Types for Circuit Maintenance GraphQL."""
import graphene
from graphene_django import DjangoObjectType
from graphene_django.converter import convert_django_field
from taggit.managers import TaggableManager

from nautobot.extras.graphql.types import TagType

from nautobot_circuit_maintenance import models, filters


@convert_django_field.register(TaggableManager)
def convert_field_to_list_tags(field, registry=None):
    """Convert TaggableManager to List of Tags."""
    return graphene.List(TagType)


class CircuitMaintenanceType(DjangoObjectType):
    """Graphql Type Object for CircuitMaintenace model."""

    class Meta:
        """Meta object boilerplate for CircuitMaintenanceType."""

        model = models.CircuitMaintenance
        filterset_class = filters.CircuitMaintenanceFilterSet

```

>>> Plugin-Defined Model GraphQL Example 1

This query gathers data on circuit maintenances as defined in the *Nautobot Circuit Maintenance Application*

The `circuit_maintenances` and `circuit_impact` parameters are defined in the **CircuitMaintenanceType** and **CircuitImpactType** classes in the application code

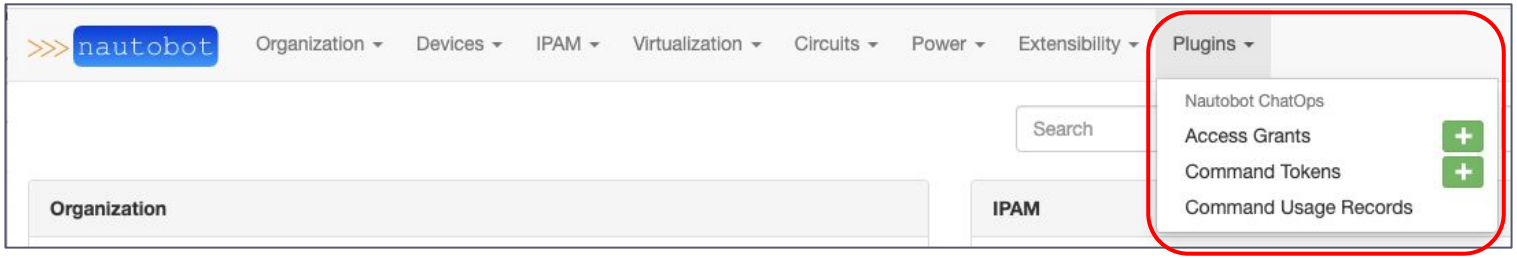
```python
class CircuitMaintenanceType(DjangoObjectType):
    """Graphql Type Object for CircuitMaintenace model."""

    class Meta:
        """Meta object boilerplate for CircuitMaintenanceType."""

        model = models.CircuitMaintenance
        filterset_class = filters.CircuitMaintenanceFilterSet

```

```python
class CircuitImpactType(DjangoObjectType):
    """Graphql Type Object for CircuitImpact model."""

    class Meta:
        """Meta object boilerplate for CircuitImpactType."""

        model = models.CircuitImpact
        filterset_class = filters.CircuitImpactFilterSet

```

>>> Add Menu Items Beneath the Top-Level *Plugins* Tab

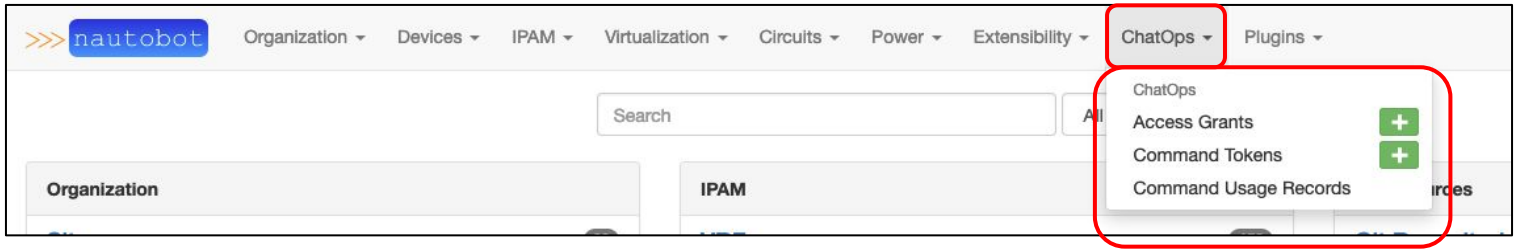
Each application can provide its own navigation menu under the *Plugins* top-level Navigation Menu Tab

This example shows the **Nautobot ChatOps** Application's Menu Group under the *Plugins* Tab



>>> Add Top-Level Menu Navigation Tabs

Applications can also customize the top-level Menu Tabs



>>> Add Additional Django Middleware

There may be times when an application requires modifying Nautobot's default input/output

Plugins can specify Django middleware to load, in addition to Nautobot's built-in middleware

The example to the right shows how to specify the `middleware` in the plugin's `__init__.py` file.

```
11  class WelcomeWizardConfig(PluginConfig):
12      """Plugin configuration for the welcome_wizard plugin."""
13
14      name = "welcome_wizard"
15      verbose_name = "Nautobot Welcome Wizard"
16      version = __version__
17      author = "Network to Code, LLC"
18      description = "Nautobot's Geting Started Wizard."
19      base_url = "welcome_wizard"
```

```
27      middleware = ["welcome_wizard.middleware.Prerequisites"]
```

>>> Add Additional Django Dependencies and Middleware

Plugins can specify Django middleware and application dependencies

```python
class WelcomeWizardConfig(PluginConfig):
    """Plugin configuration for the welcome_wizard plugin."""

    name = "welcome_wizard"
    verbose_name = "Nautobot Welcome Wizard"
    version = __version__
    author = "Network to Code, LLC"
    description = "Nautobot's Geting Started Wizard."
    base_url = "welcome_wizard"
    required_settings = []
    min_version = "1.0.0b4"
    default_settings = {
        # Add devicetype-library to Nautobot Git Repositories
        "enable_devicetype-library": True,
    }
    caching_config = {}
    middleware = ["welcome_wizard.middleware.Prerequisites"]
```

```python
from nautobot.extras.plugins import PluginConfig


class DummyPluginConfig(PluginConfig):
'dummy_plugin"
name = "Dummy plugin"
= __version__
ion = "For testing purposes only"
= "dummy-plugin"
ion = "0.9"
ion = "9.0"
re = ["dummy_plugin.middleware.DummyMiddleware"]
d_apps = ["nautobot.extras.tests.dummy_plugin_dependency"]
settings = {
my_default_key": "dummy_default_value",
myPluginConfig
```

>>> Package and Install Jobs With Dependencies (continued)

For each plugin, Nautobot will look for an iterable named `jobs` within the `jobs.py` file for the plugin

- This iterable holds the names of the included Job subclasses

```
238  # Conditionally allow jobs based on whether or not turned on.
239  jobs = []
240  if ENABLE_BACKUP:
241      jobs.append(BackupJob)
242  if ENABLE_INTENDED:
243      jobs.append(IntendedJob)
244  if ENABLE_COMPLIANCE:
245      jobs.append(ComplianceJob)
246  jobs.extend([AllGoldenConfig, AllDevicesGoldenConfig])
```

>>> Package and Install Jobs With Dependencies (continued)

In contrast to Jobs that are installed with plugins, Jobs installed manually in `JOBS_ROOT` must be self-contained

- Manually installed Jobs don't allow dependencies on other Python modules


End of Backup Slides