JUNIPER
NETWORKS

Engineering
Simplicity

Juniper Extension Toolkit Developer Guide

junos

Juniper Networks, Inc.
1133 Innovation Way
Sunnyvale, California 94089
USA
408-745-2000
www.juniper.net

Juniper Networks, the Juniper Networks logo, Juniper, and Junos are registered trademarks of Juniper Networks, Inc. in the United States and other countries. All other trademarks, service marks, registered marks, or registered service marks are the property of their respective owners.

Juniper Networks assumes no responsibility for any inaccuracies in this document. Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice.

Juniper Extension Toolkit Developer Guide
Copyright © 2024 Juniper Networks, Inc. All rights reserved.

The information in this document is current as of the date on the title page.

YEAR 2000 NOTICE

Juniper Networks hardware and software products are Year 2000 compliant. Junos OS has no known time-related limitations through the year 2038. However, the NTP application is known to have some difficulty in the year 2036.

END USER LICENSE AGREEMENT

The Juniper Networks product that is the subject of this technical documentation consists of (or is intended for use with) Juniper Networks software. Use of such software is subject to the terms and conditions of the End User License Agreement ("EULA") posted at https://support.juniper.net/support/eula/. By downloading, installing or using such software, you agree to the terms and conditions of that EULA.

Table of Contents

3   **Additional Resources**

4   **Configuration Statements and Operational Commands**

About This Guide

Use this guide to develop, deploy, use, and debug Juniper Extension Toolkit (JET) applications that are developed on Junos OS or Junos OS Evolved and third-party applications. For information about JET APIs, see the Juniper Extension Toolkit API Guide.

1
CHAPTER

Getting Started

JET Overview

Juniper Extension Toolkit (JET), an evolution of the Junos SDK, provides a modern, programmatic interface for developers of third-party applications on Junos devices. It focuses on providing a standards-based interface to the Juniper Networks Junos operating system (Junos OS) and Junos OS Evolved for customizing management and control plane functionality.

JET also includes a toolchain along with libraries and other tools to enable developers to write on-device JET applications.

Benefits of JET

- Provides APIs to interact with any Junos device.

- Supports API development in multiple languages.

- Provides tools to develop applications that run on Junos devices.

- Uses an event notification method that enables JET applications to respond to selected system events.

JET Architecture

JET is a framework that enables developers to create applications that extend the functionality of Junos OS and Junos OS Evolved. For example, a JET application might extend the Junos CLI by adding a new operational command to show application-specific states. JET applications can run on Junos devices or on another device in your operating environment and connect over the network to a Junos device.

JET applications interact with Junos OS and Junos OS Evolved through request-response and notification services over standards based transport channels. illustrates the request-response and notification services.

Figure 1: JET Request-Response and Notification Services
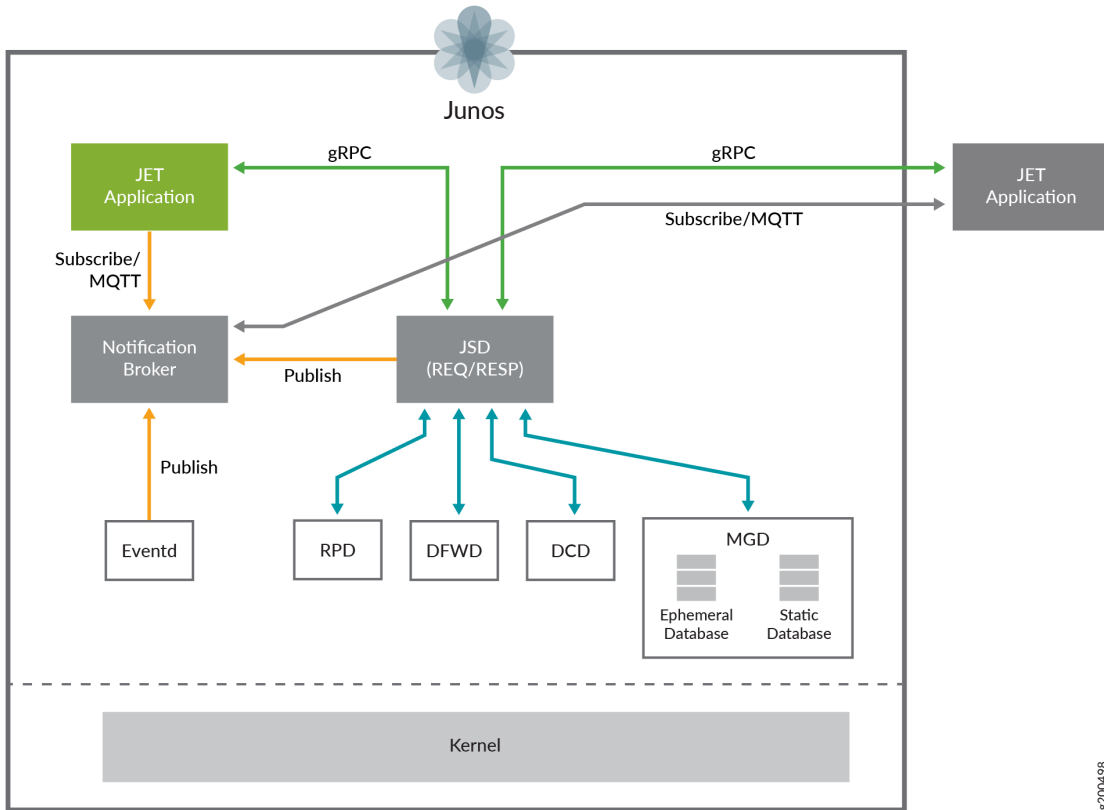
describes the request-response and notification services.

Table 1: JET Applications Interact with Junos OS Through Services

Service	Description
Request-response—An application can issue a request and wait for the response from Junos OS.	JET services process (jsd), which runs on Junos OS, provides the request-response service. When jsd receives a request (by default on TCP port 32767), it creates a new session to service the JET application. The session remains alive as long as the client and server are both up and communicating with each other. Over the lifetime of a session, jsd can execute any number of APIs. jsd can support a maximum of 8 active client sessions and execute APIs from these sessions in parallel. NOTE: For secure communications with jsd, use RSA certificates, specifically TLSv1.2 (minimum).
Notifications—An application can receive asynchronous notifications of events happening on Junos OS.	JET provides a publish-subscribe based messaging protocol and a notification broker. JET applications can register with the notification broker and inform the broker about the topics of interest to receive messages. The broker is responsible for distributing messages to the interested clients based on the topic of the message. Junos OS processes publishing the events (such as eventd) connect to the broker as a publisher and publish the events. JET utilizes Message Queue Telemetry Transport (MQTT) protocol (see https://mqtt.org/) method to implement the notification service.

JET and gRPC

JET supports the gRPC framework for remote procedure calls (RPCs). JET uses gRPC for cross-language services as a mechanism to enable request-response service. gRPC also provides a mechanism to define APIs that are programming language agnostic. For more information, see https://www.grpc.io/.

Set Up the JET VM

IN THIS SECTION

Overview

Install the JET bundle on an external device before developing or running JET applications. The JET bundle includes the JET toolchain, plug-ins, and other tools and libraries that are required for developing on-device or off-device applications. Once you install the JET toolchain on your external device, the device functions as your JET virtual machine (VM).

If you are developing an application with a dependency on C or C++ modules or developing a signed application, you must use the JET VM for JET application development.

Follow these instructions to download the following packages on your external device:

- JET software bundle: This contains the JET sandbox and toolchain file.

  In the **junos-jet-XX.YRZ.S.tar.gz** package name:

  - XX is the main release number of the product, for example, 21.

  - Y is the minor release number of the product, for example, 2.

  - R is the type of software release, for example, R for FRS or maintenance release.

  - Z is the build number of the product, for example, 1, indicating the FRS rather than a maintenance release.

  - S is the spin number of the product, for example, 13.

- JET client IDL library: This contains the protobuf definition files for the JET APIs. You can also view them on the JET GitHub repository.

Install the JET Software Bundle

Install JET on an external device that will function as your JET VM. This task walks you through how to install JET on a device running Ubuntu.

1. Download the JET software bundle onto your external device from the Juniper Networks downloads website. In this example, we downloaded the software bundle for JET Release 21.4R2.

2. Open a Terminal, Command, or similar window on your device. Confirm the file downloaded.

```
user@jet-vm:~$ cd ~/Downloads
user@jet-vm:~/Downloads$ ls
junos-jet-21.4R2.10.tar.gz
```

3. Extract the file.

```
user@jet-vm:~/Downloads$ tar -zxf junos-jet-21.4R2.10.tar.gz
```

4. View the contents of the extracted file.

```
user@jet-vm:~/Downloads$ ls
junos-jet-21.4R2.10  junos-jet-21.4R2.10.tar.gz
user@jet-vm:~/Downloads$ cd junos-jet-21.4R2.10/
user@jet-vm:~/Downloads/junos-jet-21.4R2.10$ ls
init_linux_prereqs.sh  junos-jet-sb-21.4R2.10-signed.tgz        README
install                junos-jet-toolchain-21.4R2.10-signed.tgz
```

5. Install the package. Use the `./install` command if you are logged in as the root user. If you are logged in as a different user, use the `sudo ./install` command. If using the `sudo ./install` command, you are prompted to enter the password for your account.

   The output in this example has been snipped for length.

```
user@jet-vm:~/Downloads/junos-jet-21.4R2.10$ sudo ./install
[sudo] password for user:
Installing jot utility
Reading package lists... Done
Building dependency tree
[...]
Installation of JET SB successful.
[...]
Installation of JET Toolchain successful.
Installation of junos-jet-sb and junos-jet-toolchain completed
```

The JET sandbox and toolchain are installed on your device. This device now functions as your JET VM.

Set Up Your Virtual Machine Environment

To set the PATH variable and prepare the JET VM:

1. Open a Terminal window in the JET VM.
2. Add the absolute path to the **/junos-jet-XX.YRZ.S.tar.gz/bin** directory to the PATH variable in **.bashrc**.

```
user@jet-vm:~$ echo 'PATH=$PATH:/usr/local/junos-jet/18.3R1/junos-jet-
XX.YRZ.S.tar.gz'>>~/.bashrc
```

3. Run the following command to display the JET **XX.YRZ.S.tar.gz** path in the output:

```
user@jet-vm:~$ source ~/.bashrc
```

4. Run the env command to ensure the PATH variable contains the directory path you just added.

```
user@jet-vm:~$ env

PATH=/usr/lib/lightdm/lightdm:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin: /
bin:/usr/games:/usr/local/junos-jet/18.3R1/junos-jet-XX.YRZ.S.tar.gz/bin
```

You are ready to develop applications in the JET VM.

Download the JET IDL

1. Download the IDL file onto your device from the Juniper Networks downloads website.
2. Make a directory on your device to store the proto definition files.

```
user@jet-vm:~$ mkdir jet
user@jet-vm:~$ cd jet
user@jet-vm:~/jet$ mkdir proto
```

3. Confirm the IDL file downloaded.

```
user@jet-vm:~/jet$ cd ../Downloads
user@jet-vm:~/Downloads$ ls
jet-idl-21.4R2.10.tar.gz  junos-jet-21.4R2.10  junos-jet-21.4R2.10.tar.gz
```

4. Unzip the IDL file into the directory you created.

```
user@jet-vm:~/Downloads$ tar -xzf jet-idl-21.4R2.10.tar.gz -C ../jet/proto/
```

5. Confirm the file unzipped properly.

```
user@jet-vm:~/Downloads$ cd ../jet/proto
user@jet-vm:~/jet/proto$ ls
1  2  README
```

6. You can view the most recent JET API proto definition files in the folder called **2**.

```
user@jet-vm:~/jet/proto$ cd 2
user@jet-vm:~/jet/proto/2$ ls
jnx_authentication_service.proto  jnx_routing_base_service.proto
jnx_common_addr_types.proto       jnx_routing_base_types.proto
jnx_common_base_types.proto       jnx_routing_bgp_service.proto
jnx_firewall_service.proto        jnx_routing_flexible_tunnel_profile.proto
jnx_interfaces_service.proto      jnx_routing_flexible_tunnel_service.proto
jnx_management_service.proto      jnx_routing_interface_service.proto
jnx_registration_service.proto    jnx_routing_rib_service.proto
```

You have downloaded the IDL file successfully. It is ready to use.

Prepare to Compile the Proto Definition Files in Python

If you are using Python, you need to install certain gRPC tools before you can compile the proto definition files in Python. These steps are also described on the Quick Start page on the gRPC site.

1. (Optional) Activate a virtual environment.

```
user@jet-vm:~/PycharmProjects/jetTest$ source venv/bin/activate
(venv) user@jet-vm:~/PycharmProjects/jetTest$
```

2. Check you have the desired versions of Python and Pip installed. If you do not have them installed, install them now.

```
(venv) user@jet-vm:~/PycharmProjects/jetTest$ python --version
Python 3.8.10
(venv) user@jet-vm:~/PycharmProjects/jetTest$ python -m pip --version
pip 22.2.2 from /home/user/PycharmProjects/jetTest/venv/lib/python3.8/site-packages/pip
(python 3.8)
```

3. Install gRPC if it is not already installed.

```
(venv) user@jet-vm:~/PycharmProjects/jetTest$ python -m pip install grpcio
Collecting grpcio
  Downloading grpcio-1.48.1-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (4.6 MB)
                                                  4.6/4.6 MB 6.1 MB/s eta 0:00:00
Collecting six>=1.5.2
  Using cached six-1.16.0-py2.py3-none-any.whl (11 kB)
Installing collected packages: six, grpcio
Successfully installed grpcio-1.48.1 six-1.16.0
```

If gRPC is already installed, you will see this message:

```
(venv) user@jet-vm:~$ python -m pip install grpcio
Requirement already satisfied: grpcio in ./PycharmProjects/jetTest/venv/lib/python3.8/site-
packages (1.48.1)
```

4. Install gRPC tools if it is not already installed.

```
(venv) user@jet-vm:~/PycharmProjects/jetTest$ python -m pip install grpcio-tools
Collecting grpcio-tools
  Downloading grpcio_tools-1.48.1-cp38-cp38-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
(2.4 MB)
                                                  2.4/2.4 MB 10.6 MB/s eta 0:00:00
Requirement already satisfied: setuptools in ./venv/lib/python3.8/site-packages (from grpcio-
tools) (60.2.0)
```

```
Collecting protobuf<4.0dev,>=3.12.0
  Using cached protobuf-3.20.1-cp38-cp38-manylinux_2_5_x86_64.manylinux1_x86_64.whl (1.0 MB)
Requirement already satisfied: grpcio>=1.48.1 in ./venv/lib/python3.8/site-packages (from
grpcio-tools) (1.48.1)
Requirement already satisfied: six>=1.5.2 in ./venv/lib/python3.8/site-packages (from
grpcio>=1.48.1->grpcio-tools) (1.16.0)
Installing collected packages: protobuf, grpcio-tools
Successfully installed grpcio-tools-1.48.1 protobuf-3.20.1
```

You are ready to compile the IDL files you need to develop your application.

2
CHAPTER

Application Development

Develop Off-Device JET Applications

Overview

You can use JET to develop applications that run off-device. This allows you to leverage the benefits of JET on all devices on your network. For ease of development, you can write off-device JET applications in the language of your choice. To develop an off-device application:

1. Download and compile the IDL file.

2. Develop the application using the language of your choice.

3. Package the application.

4. Deploy the application package on an external server or run the application directly from the JET VM.

shows the off-device application development workflow.

Figure 2: Off-Device JET Application Workflow
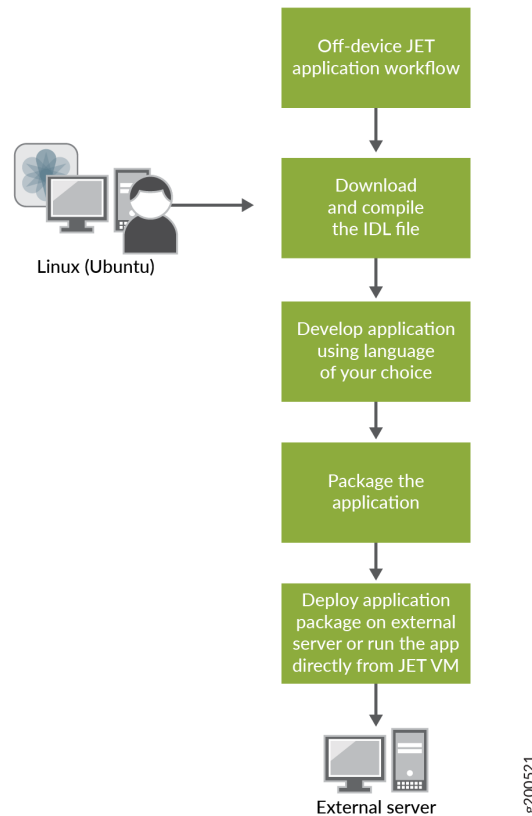
Develop and Package Your Application

Before devloping your application, make sure you have already followed the instructions in "Set Up the JET VM" on page 4 to set up the JET VM and download the IDL file.

1. Compile the proto definition files that you plan to use in the language of your choice.

   In this example, we are using Python. We compile the files for the management, authentication, and common base types APIs.

```
user@jet-vm:~/jet$ python -m grpc_tools.protoc -I./proto/2 --python_out=. --grpc_python_out=.
proto/2/jnx_management_service.proto
user@jet-vm:~/jet$ python -m grpc_tools.protoc -I./proto/2 --python_out=. --grpc_python_out=.
proto/2/jnx_authentication_service.proto
user@jet-vm:~/jet$ python -m grpc_tools.protoc -I./proto/2 --python_out=. --grpc_python_out=.
```

```
proto/2/jnx_common_base_types.proto
user@jet-vm:~/jet$ ls -lrt
total 48
drwxrwxr-x 4 user user  4096 Sep  1 15:34 proto
-rw-rw-r-- 1 user user 13969 Sep  1 16:02 jnx_management_service_pb2.py
-rw-rw-r-- 1 user user  8342 Sep  1 16:02 jnx_management_service_pb2_grpc.py
-rw-rw-r-- 1 user user  2472 Sep  1 16:04 jnx_authentication_service_pb2.py
-rw-rw-r-- 1 user user  3086 Sep  1 16:04 jnx_authentication_service_pb2_grpc.py
-rw-rw-r-- 1 user user  3971 Sep  1 16:04 jnx_common_base_types_pb2.py
-rw-rw-r-- 1 user user   159 Sep  1 16:04 jnx_common_base_types_pb2_grpc.py
```

You have compiled Python and gRPC modules for each specified API.

2. Develop the application using the language of your choice.

   If you are developing an application with a dependency on C or C++ modules or developing a signed application, you must use the JET VM for JET application development.

   > *(i)* **NOTE**: Starting in Junos OS Release 21.1R1 and Junos OS Evolved Release 22.3R1, Python 2.7 is no longer supported. Use Python 3 instead.

   You can write the application using a stub after a client side stub is generated. For more information on generating the gRPC client side stubs, writing the application using the stub, and generating code from an IDL file in the language of your choice, see https://www.grpc.io/docs/.

3. Package the application using JSON. See "Package JET Applications" on page 41 for more information.

SEE ALSO

Understanding Python Automation Scripts for Devices Running Junos OS

IPv6 Support in Python Automation Scripts

Prepare to Deploy Your Application

IN THIS SECTION

- Configure JET Interaction with Junos OS | **15**

Run your application on an external server or directly from the JET VM. Before you deploy your application on an external server, you need to configure JET interaction with Junos OS.

Configure JET Interaction with Junos OS

To run an off-device application, you need to enable the `request-response` configuration on Junos OS or Junos OS Evolved. When using the request-response service, the client application issues a request and synchronously waits for the response from the Junos server. Use this section to configure the JET service process (jsd) for the request-response service to run in Secure Sockets Layer (SSL) mode. This provides increased security and enables SSL-based API connections.

> ⓘ **NOTE**: Currently, JET supports Transport Layer Security (TLS) version 1.2 for certificate exchange and supports multiple encryption algorithms, but does not support mutual authentication. This means that clients can authenticate the server, but the server can not authenticate clients using SSL/TLS certificates. For client authentication, use the LoginCheck() procedure from the authentication service API.

1. Enable jsd to use SSL by adding and configuring the certificate name locally. The certificate must be an RSA certificate. ECDSA and DSA SSL certificates are not supported.

   This method is same as other SSL-based services in Junos OS like xnm-ssl. Keep track of the certificate name entry you specify during certificate generation. You will use it for the `HOST_OVERRIDE` option in the example Python application in the next section. In this example, the certificate name is `router`.

   ```
   user@jet-vm:~ jet$ openssl genrsa -aes256 -out router.key.orig 2048
   user@jet-vm:~ jet$ openssl req -new -key router.key.orig -out router.csr
   user@jet-vm:~ jet$ openssl rsa -in router.key.orig -out router.key
   user@jet-vm:~ jet$ openssl x509 -req -days 365 -in router.csr -signkey router.key -out router.crt
   user@jet-vm:~ jet$ cat router.crt router.key > router.pem
   ```

> ⓘ **NOTE**: If a certificate is updated with the same identifier, the changes will not be reflected for jsd. You need to either configure the certificate with a new identifier in the jsd hierarchy or perform a jsd restart to reflect the changes made.

2. Copy the SSL certificate **.pem** file to the Junos device.

```
user@device% scp pem-file-name device-name:/var/tmp
```

For example:

```
user@device% scp router.pem device:/var/tmp
```

3. Load the certificate into the keychain on the Junos device. For example, if the local name of the SSL certificate is `sslcert`:

```
[edit]
user@device# set security certificates local sslcert load-key-file /var/tmp/router.pem
```

4. Enable support for SSL for the loaded certificate.

```
[edit system services extension-service request-response grpc]
user@device# set ssl local-certificate cert-name
```

For example:

```
[edit system services extension-service request-response grpc]
user@device# set ssl local-certificate sslcert
```

5. (Optional) Specify the specific IP address or port that will use SSL. SSL makes that address or port a secure channel.

```
[edit system services extension-service request-response grpc]
user@device# set ssl address address
user@device# set ssl port port-number
```

If you set the address to 0.0.0.0, the device uses SSL on all ports. For example, to enable support for SSL on the gRPC endpoint on all ports and the default TCP port 51051:

```
[edit system services extension-service request-response grpc]
user@device# set ssl address 0.0.0.0
user@device# set ssl port 51051
```

6. Specify the maximum number of simultaneous connections for request-response that can be attached to jsd. The higher the number, the higher the impact on the client's performance.

```
[edit system services extension-service request-response grpc]
user@device# set max-connections 8
```

You have configured jsd for request-response service to run in SSL mode. You are ready to deploy your JET off-device application.

7. Specify the scripts to use.

```
[edit]
user@device# set system scripts language python3
```

> **NOTE**: Starting in Junos OS Release 21.1R1 and Junos OS Evolved Release 22.3R1, Python 2.7 is no longer supported and the `set system scripts language python` statement is deprecated. Use the `set system scripts language python3` statement instead.

Example: Python JET Application

IN THIS SECTION

Use this example to develop an off-device JET application written in Python. You can follow the same guidance for other languages that are supported by gRPC. This Python JET application runs the command `get-system-uptime-information` in XML format.

In this example, the `HOST_OVERRIDE` option uses the certificate name that you specified during the certificate generation. See "Prepare to Deploy Your Application" on page 14.

> **(i)** **NOTE**: Juniper Networks supports both of the following forms for denoting XML opening and closing tags: `<xml-tag/>` and `<xml-tag></xml-tag>`.

Junos OS Release 18.4R1 and Later

Use the example Python application shown in this section as a guide if you are using Junos OS Release 18.4R1 or later.

If you are writing your application using Python 3, include the PASS keyword in the Exception block of the script.

```
except Exception as tx:
    pass
```

```python
#!/usr/bin/env python

# A simple Python client to run XML OP command 'get-system-uptime-information'

# Environment
# Python 2.7.12
# grpcio (1.12.0)
# grpcio-tools (1.12.0)

# Following files should be available in current working directory
# jnx_authentication_service_pb2_grpc.py
# jnx_authentication_service_pb2.py
# jnx_management_service_pb2_grpc.py
# jnx_management_service_pb2.py

import argparse
import grpc
import os
import stat
```

```python
import jnx_authentication_service_pb2
import jnx_authentication_service_pb2_grpc
import jnx_management_service_pb2
import jnx_management_service_pb2_grpc
import jnx_common_base_types_pb2


_HOST_OVERRIDE = 'router'

def Main():
    try:
        parser = argparse.ArgumentParser()

        parser.add_argument('-d','--device', help='Input hostname',
            required=True)
        parser.add_argument('-t','--timeout', help='Input time_out value',
            required=True,type=int)
        parser.add_argument('-u', '--user', help='Input username',
            required=True)
        parser.add_argument('-pw', '--password', help='Input password',
            required=True)

        args = parser.parse_args()

        #Establish grpc channel to jet router
        creds = grpc.ssl_channel_credentials(open('/tmp/router.pem').read(),
                                             None, None)
        channel = grpc.secure_channel(args.device + ":32767", creds,
            options=(('grpc.ssl_target_name_override', _HOST_OVERRIDE,),))

        #create stub for authentication services
        stub = jnx_authentication_service_pb2_grpc.AuthenticationStub(channel)
        #Authenticate
        login_request = jnx_authentication_service_pb2.LoginRequest(
            username=args.user, password=args.password, client_id="SampleApp")
        login_response = stub.Login(login_request, args.timeout)
        #Check if authentication is successful
        if login_response.status.code == jnx_common_base_types_pb2.SUCCESS:
            print "[INFO] Connected to gRPC Server"
        else:
            print "[ERROR] gRPC Server Connection failed:"
            print login_response.status.message
```

```python
        #Create stub for management services
        stub = jnx_management_service_pb2_grpc.ManagementStub(channel)
        print "[INFO] Connected to management service"
        for i in range(1):
            #Provide API request details
            op_xml_command = "<get-system-uptime-information></get-system-uptime-information>"
            op = jnx_management_service_pb2.OpCommandGetRequest(
                xml_command=op_xml_command, out_format=2)
            # Invoke API
            op_response = stub.OpCommandGet(op, args.timeout)
            # Check API response like status and output
            for resp in op_response:
                if resp.status.code == jnx_common_base_types_pb2.SUCCESS:
                    print "[INFO] Invoked OpCommandGetRequest succeeded"
                    print "[INFO] Return output in CLI format = "
                    print resp.data
                else:
                    print "[ERROR] Invoked OpCommandGetRequest failed"
                    print "[ERROR] " + resp.status.message


    except Exception as ex:
        print ex


if __name__ == '__main__':
    Main()
```

```
user@jet-vm:~ jet$ python mgd_api_new_doc_example_ssl.py -d JUNOS_DEVICE -t TIMEOUT -u USER -pw
PASSWORD

[INFO] Connected to gRPC Server
[INFO] Connected to management service
[INFO] Invoked OpCommandGetRequest succeeded
[INFO] Return output in CLI format =

Current time: 2018-11-08 09:36:40 PST
Time Source:  NTP CLOCK
System booted: 2018-10-09 17:02:56 PDT (4w1d 17:33 ago)
Protocols started: 2018-10-09 17:05:09 PDT (4w1d 17:31 ago)
```

```
Last configured: 2018-11-08 09:30:28 PST (00:06:12 ago) by root
 9:36AM up 29 days, 17:34, 2 users, load averages: 1.05, 0.77, 0.57
```

Before Junos OS Release 18.4R1

Use the example Python application in this section as a guide if you are using Junos OS releases prior to 18.4R1.

```python
#!/usr/bin/env python

# A simple Python client to run XML OP command 'get-system-uptime-information'

# Environment
# Python 2.7.12
# grpcio (1.12.0)
# grpcio-tools (1.12.0)

# Following files should be available in current working directory
# authentication_service_pb2_grpc.py
# authentication_service_pb2.py
# management_service_pb2_grpc.py
# management_service_pb2.py

import argparse
import grpc

import authentication_service_pb2
import authentication_service_pb2_grpc
import management_service_pb2
import management_service_pb2_grpc

_HOST_OVERRIDE = 'router'

def Main():
    try:
        parser = argparse.ArgumentParser()

        parser.add_argument('-d','-device', help='Input hostname',
            required=True)
        parser.add_argument('-t','-timeout', help='Input time_out value',
            required=True,type=int)
```

```python
parser.add_argument('-u', '-user', help='Input username',
    required=True)
parser.add_argument('-pw', '-password', help='Input password',
    required=True)

args = parser.parse_args()

#Establish grpc channel to jet router
creds = grpc.ssl_channel_credentials(open('/tmp/router.pem').read(),
                                     None, None)
channel = grpc.secure_channel(args.device + ":51051", creds,
    options=(('grpc.ssl_target_name_override', _HOST_OVERRIDE,),))

#create stub for authentication services
stub = authentication_service_pb2_grpc.LoginStub(channel)
#Authenticate
login_request = authentication_service_pb2.LoginRequest(
    user_name=args.user, password=args.password, client_id="SampleApp")
login_response = stub.LoginCheck(login_request, args.timeout)

#Check if authentication is successful
if login_response.result == True:
    print "[INFO] Connected to gRPC Server:"
    print login_response.result
else:
    print "[ERROR] gRPC Server Connection failed!!!"
    print login_response.result

#Create stub for management services
stub = management_service_pb2_grpc.ManagementRpcApiStub(channel)
print "[INFO] Connected to JSD and created handle to mgd services"

for i in range(1):
    #Provide API request details
    op_xml_command = "<get-system-uptime-information>" \
    "</get-system-uptime-information>"
    op = management_service_pb2.ExecuteOpCommandRequest(
        xml_command=op_xml_command, out_format=2, request_id=1000)
    # Invoke API
    result = stub.ExecuteOpCommand(op, 100)
    # Check API response like status and output
    for i in result:
        print "[INFO] Invoked ExecuteOpCommand API return code = "
```

```
            print i.status
            print "[INFO] Return output in CLI format = "
            print i.data
    except Exception as ex:
        print ex


if __name__ == '__main__':
    Main()
```

```
user@jet-vm:~ jet$ python mgd_api_doc_example_ssl.py -d JUNOS_DEVICE -t TIMEOUT_VAL -u USER -pw
PASSWORD

[INFO] Connected to gRPC Server:
True
[INFO] Connected to JSD and created handle to mgd services
[INFO] Invoked ExecuteOpCommand API return code =
0
[INFO] Return output in CLI format =
Current time: 2018-09-04 11:24:36 PDT
Time Source:  NTP CLOCK
System booted: 2018-08-31 10:58:22 PDT (4d 00:26 ago)
Protocols started: 2018-08-31 11:00:52 PDT (4d 00:23 ago)
Last configured: 2018-08-31 14:21:32 PDT (3d 21:03 ago) by root
11:24AM  up 4 days, 26 mins, 0 users, load averages: 1.20, 1.27, 1.10
```

RELATED DOCUMENTATION

Develop On-Device JET Applications

IN THIS SECTION

Overview

JET applications written in C, C++, and Python languages can run on-device. You can develop the applications in the downloaded JET VM and then deploy these applications on the device running Junos OS. You can sign on-device JET applications to show that they can be trusted.

> ⓘ **NOTE**: The Python 2.7 end-of-life and end-of-support date is January 1, 2020. The official upgrade path for Python 2.7 is to Python 3. As support for Python 3 is added to devices running Junos OS for the different types of on-device scripts, we recommend that you migrate supported script types from Python 2 to Python 3.
>
> Starting in Junos OS Release 21.1R1 and Junos OS Evolved Release 22.3R1, Python 2.7 is no longer supported and the `set system scripts language python` statement is deprecated. Use the `set system scripts language python3` statement instead.

Figure 3 on page 25 shows the application development workflow for unsigned and signed on-device JET applications.

Figure 3: On-Device JET Application Workflow
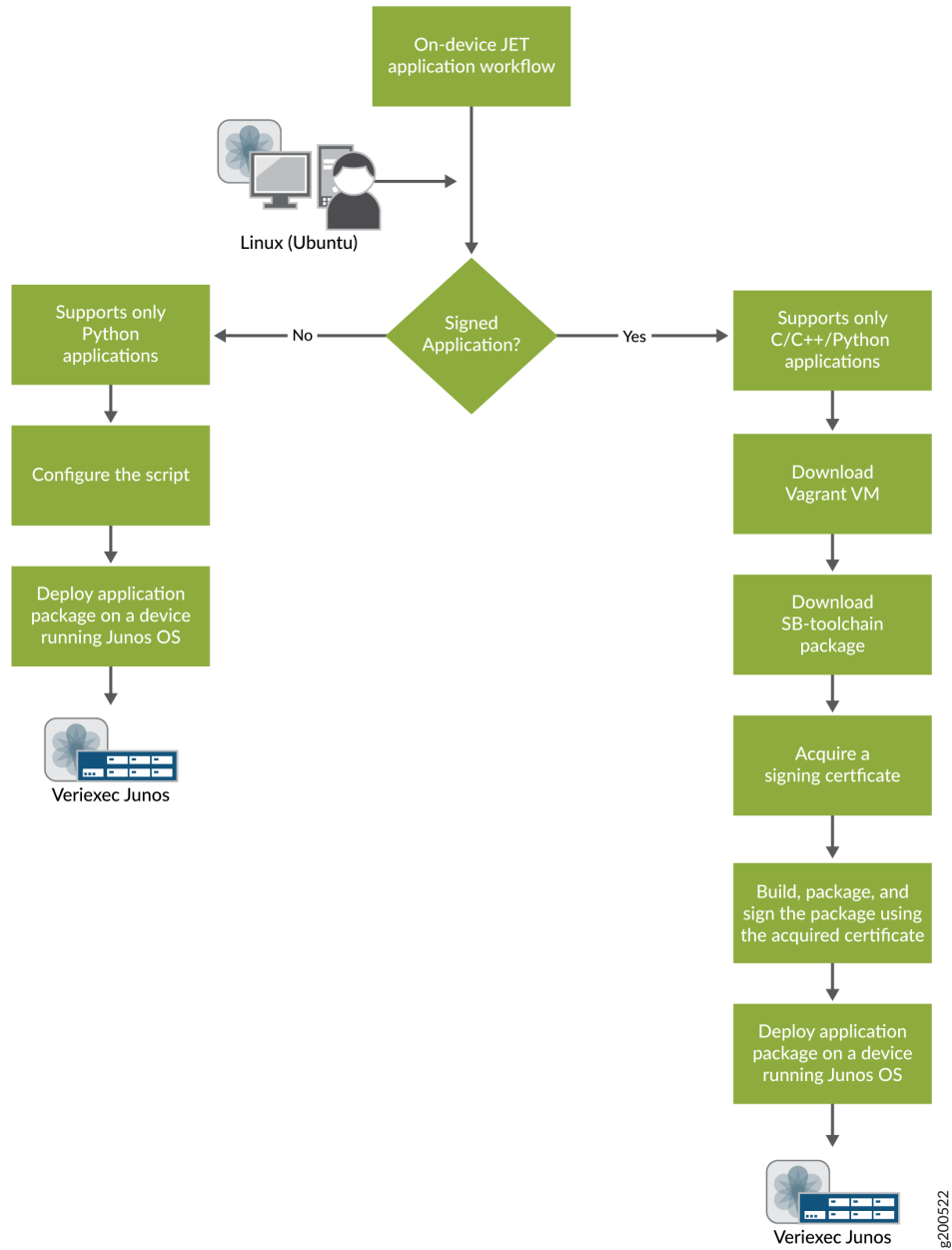
SEE ALSO

| Understanding Python Automation Scripts for Devices Running Junos OS

| IPv6 Support in Python Automation Scripts

Develop Unsigned JET Applications

Unsigned JET applications can only be written in Python.

To develop an unsigned JET application:

1. (Optional) Download and set up the JET VM. See "Set Up the JET VM" on page 4.

2. Develop your application in Python.

3. Package your application. See "Package JET Applications" on page 41.

4. Configure the `language` statement on the Junos device. For example, to use Python 3 to run a JET script that supports Python 3:

```
[edit]
user@device# set system scripts language python3
```

See Understanding Python Automation Scripts for Devices Running Junos OS for more information.

5. Run the application on a device running on Junos OS.

Develop Signed JET Applications

IN THIS SECTION

- Request a Signing Certificate | 27

You can develop signed applications in C, C++, or Python.

To develop a signed application:

1. Download the packages you need and set up the JET VM. See "Set Up the JET VM" on page 4.

2. Request a signing certificate. See "Request a Signing Certificate" on page 27.

3. Develop the application.

4. Configure the license if your application is written in C or C++. This step is optional for applications written in Python. See Configuring the JET Application and its License on a Device Running Junos OS for details.

5. Build the package and sign the package using the acquired certificate.

6. Deploy the application on a device running Junos OS.

Request a Signing Certificate

To develop and distribute JET applications, you must install a package signing certificate on the JET VM by executing the certificate request script. This script assists you in creating a signing key and a certificate request for use with JET.

> ⚠️ **CAUTION**: Never share the signing key with anyone, including Juniper Networks. The key enables anyone to sign applications that your router will trust. Therefore, treat the key with the same level of security as the root password for the routers. Once you obtain your signing key, save it in a file outside of the VM.

The certificate request script asks for the following information:

- City, state, and country.

- Your organization and unit. The organization should not be vague. There cannot be any hyphens ("-").

- Certificate type: Specify whether the certificate is commercial or private. Non-Juniper entities must use commercial. Private certificates are only assigned when the organization is Juniper.

- Provider prefix: This is the unique provider name assigned by Juniper to each JET partner.

- User string: This is an additional specification of your choosing. It could be a string specifying the development team or project name. The user string can consist of a lowercase letter followed by one or more lowercase letters or numbers (for example, teamjet2).

- Deployment scope: The deployment scope is the string assigned by Juniper to differentiate multiple certificates for the same JET partner. This defines the validity period for the generated certificate. The scope can be commercial or evaluation. If none is assigned to you, leave it empty.

- Index number: This number is known as a certificate generations number. It will be 1 for your initial certificate. When a certificate expires and when you request a new one, this number will be incremented.

- Email address: The email address for the certificate contact will be embedded into the certificate. We recommend using the email address of a department or unit in your organization. We recommend that you do not use a personal email address.

To create a signed application, request certificates and copy them as explained in the following procedure. This procedure is optional if you want to create an unsigned application.

To manually request a certificate:

1. Create the **/usr/local/junos-jet/certs** directory if it does not already exist on your device.

2. In a VM terminal, run the `jet-certificate-request` command.

3. The script leads you through a series of questions. Answer the questions and press Enter after each answer. See the requirements for each answer listed above.

4. Based on your answers, the script generates two files in the **/usr/local/junos-jet/certs** directory: ***certname*_key.pem** and ***certname*_req.pem**. The *certname* is the name of the certificate.

   The certificate name must follow the format ORGANIZATION-USER-TYPE-NUMBER. All four parameters are mandatory or else you will see the following error:

   ```
   ERROR: CN has invalid format; regex: ^([a-z0-9]+)-([a-z0-9]+)-(commercial|private)-([1-9]
   [0-9]*)$
         Expected format: ORGANIZATION-USER-TYPE-NUMBER
             organization: [a-z0-9]+
                 Must be "juniper" for type is "private"
             user:   [a-z0-9]+
             type:   commercial|private
             number: [1-9][0-9]*
   ```

5. Save the ***certname*_key.pem** file outside the VM. This is your signing key. Ensure that no one outside of your development organization has access to it. Never share the signing key with anyone, including Juniper Networks.

   The key enables anyone to sign applications that your router will trust. Therefore, treat the key with the same level of security as the root password for the routers.

6. Send the ***certname*_req.pem** file to JET Certificate Processing at mailto:jet-cert@juniper.net. This file contains your certificate request. The requestor should be authorized to request signing certificates on behalf of their organization. JET Certificate Processing will immediately send your certificate to you.

7. When you receive your certificate, save it as *certname* and copy it to the **/usr/local/junos-jet/certs** directory.

8. Verify the certificate and the signing key are available in the **/usr/local/junos-jet/certs** directory.

Compile 64-Bit Applications

JET supports 64-bit applications for Junos OS with FreeBSD and Junos OS with upgraded FreeBSD. When you are ready to package your application, use the following commands to compile 64-bit applications for use with the AMD64 or ARM64 64-bit processor architecture.

To compile the application for use with AMD64 and Junos OS with FreeBSD:

1. Check you have the GCC toolchain, which should be included in the JET package.

2. Use the command `mk-amd64` *application name* to build the application package.

To compile the application for use with AMD64 and Junos OS with upgraded FreeBSD:

1. Check you have the Clang toolchain, which should be included in the JET package.

2. Use the command `mk-amd64,bsdx` *application name* to build the application package.

To compile the application for use with ARM64 and Junos OS with upgraded FreeBSD:

1. Check you have the Clang toolchain, which should be included in the JET package.

2. Use the command `mk-arm64,bsdx` *application name* to build the application package.

Example: Develop a Signed C Package

After you have set up the JET VM and acquired a signing certification, you are ready to create the development sandbox in the VM and start developing your signed, on-device application. Use this example to create C applications echoclient and echoserver.

1. Check out the sandbox. A sandbox is a build tree with a little environment file called **.sandbox-env** at the top that is used by a wrapper script **mk** to ensure the build environment is properly conditioned.

```
user@jet-vm:~$ mksb  -n capp echoclient /home/user/capp_server/src/echoclient.json
```

> **NOTE**: The echo client is a demo application. In the **bin/** directory, all the necessary configuration and build related files are available within the sandbox along with source file for the echo client.

2. Build an echo client package.

```
user@jet-vm:/home/user/capp/src$ mk-i386,bsdx echoclient
```

> ℹ **NOTE**: Starting in Junos OS Release 20.2R1, if you will be running your JET
> application on a ACX710 device, you can use the Clang toolchain for ARM-based
> compilation of JET applications written in C, Python, or Ruby. Use the command `mk-arm,bsdx` instead of `mk-i386,bsdx` to use the Clang toolchain to compile your application.

3. Copy the echo client package onto the device running on Junos OS.

```
user@jet-vm:/home/user/capp/src$ scp /home/user/capp/junos-jet-sb-obj//ship/echoclient-x86-32-20180829.065039_user.tgz root@device:/var/tmp
```

4. Enter configuration mode on the Junos device.

```
root@device:~ # cli
```

5. Install the echo client package.

```
root@device> request system software add /var/tmp/echoclient-x86-32-20180829.065039_user.tgz

Verified echoclient-x86-32-20180829.065039_user signed by junosmanageability-dev-beta-1
method RSA2048+SHA1
```

Confirm it was installed successfully.

```
root@device> show version

Hostname: device
Model: mx480
Junos: 18.4-20180627_dev_common.1
JUNOS OS Kernel 32-bit  [20180621.191151_fbsd-builder_stable_11]
…
…
JET app echoclient [20180829.065039_user]
```

6. Check out the echo server sandbox.

```
user@jet-vm:/home/user$ mksb -n capp_server echoserver /home/user/capp_server/src/
echoclient.json
```

7. Build the echo server package.

```
user@jet-vm:/home/user/capp_server/src$ mk-i386,bsdx echoserver
```

8. Copy the echo server package to the Junos VM.

```
user@jet-vm:/home/user/capp_server/src$ scp /home/user/capp_server/junos-jet-sb-obj//ship/
echoserver-x86-32-20180829.065703_user.tgz root@device:/var/tmp/
```

9. Add the echo server package to the Junos device.

```
root@device> request system software add /var/tmp/echoserver-x86-32-20180829.065703_user.tgz

Verified echoserver-x86-32-20180829.065703_user signed by junosmanageability-dev-beta-1
method RSA2048+SHA1
```

10. Check that the packages were added successfully.

```
root@device> show version

Hostname: device
Model: mx480
Junos: 18.4-20180627_dev_common.1
JUNOS OS Kernel 32-bit  [20180621.191151_fbsd-builder_stable_11]
JUNOS OS libs [20180621.191151_fbsd-builder_stable_11]
……….
………
JET app echoserver [20180829.065703_user]
JET app echoclient [20180829.065039_user]
```

11. Configure the echo server's provider's ID, license type, and deployment scope on the Junos device. Use the same provider license that you used to package it.

```
root@device# set system extensions providers 12345 license-type juniper deployment-scope
commercial
```

For more information, see Configuring the JET Application and its License on a Device Running Junos OS.

12. Configure the echo server on the Junos device.

```
root@device# set system extensions extension-service application file echoserverd
[edit]
root@device# commit
commit complete
[edit]
root@device# exit
root@device> request extension-service start echoserverd


Extension-service application 'echoserverd' started with pid: 12345
```

13. Configure the echo client's provider's ID, license type, and deployment scope on the Junos device. Use the same provider license that you used to package it.

```
root@device# set system extensions providers 56789 license-type juniper deployment-scope
commercial
```

14. Configure the echo client application on the Junos device.

```
root@device# set system extensions extension-service application file echoclientd arguments
"127.0.0.1 Testmessage"
[edit]
root@device# commit
commit complete
[edit]
root@device# exit
```

15. Run the echo client application.

```
root@device> request extension-service start echoclientd
```

```
Extension-service application 'echoclientd' started with pid: 56789
-- server reply:Testmessage
-- Testmessage
```

Example: Develop a Signed Python Package Without C Dependencies

After you have set up the JET VM and acquired a signing certification, you are ready to create the development sandbox in the VM and start developing your signed, on-device application. Use this example to develop a signed Python package without C dependencies.

1. In the VM, go to the **/home/user** directory.
2. Create a sandbox by using the `mksb` command, where `SamplePyApp` is the name of the sandbox. A sandbox is a build tree with a little environment file called **.sandbox-env** at the top that is used by a wrapper script **mk** to ensure the build environment is properly conditioned.

```
user@jet-vm:~$ mksb  -n SamplePyApp
% mksb  -n SamplePyApp
```

3. Create subdirectories in the sandbox.

   First, use the `workon` command to go into your sandbox. The `workon` command takes you directly to the **$SB/src** directory and sets the sandbox correctly.

```
user@jet-vm:/home/user$ workon SamplePyApp
```

   Alternatively, you can `cd` to the **src** directory of your sandbox.

   Next, create subdirectories for application code in **$SB/src/python**, **$SB/src/lib**, or **$SB/src/bin**, based on whether you need Python, library, or bin (executable) files.

```
user@jet-vm:/home/user/pyapp/src/python$ mkdir SamplePyApp
```

4. Develop the code.

```
user@jet-vm:$ /home/user/pyapp/src/python/SamplePyApp$ ls cmdline_args.py.
```

If you are writing your application using Python 3, include the PASS keyword in the Exception block of the script.

```
    except Exception as tx:
        pass
```

5. Write an application JSON file to package the application.

```
SamplePyApp.json
{
"app-name" : "SamplePyApp",
"app-path" : "python/SamplePyApp",
"language" : "python",
"main-scripts" : ["cmdline_args.py"],
"app-type" : "standalone",
"sign" : "yes",
"os-type" : "bsd11",
"target-arch" : "i386",
"description" : "Simple Python App"
}
```

See"Package JET Applications" on page 41 for more information.

6. Add the path to **jet-pkg-gen.py** to your **.bashrc** file.

```
user@jet-vm:/home/user$ echo 'PATH=$PATH:/usr/local/junos-jet/18.2R1.9/junos-jet-sb/src/junos/
host-utils/scripts' >> ~/.bashrc
user@jet-vm:/home/user/pyapp/src$ source ~/.bashrc
```

7. Autogenerate the appropriate makefiles by running the `jet-pkg-gen.py` command. The `jet-pkg-gen.py` command takes two options:

  • The `-i` option is followed by the path and filename of the JSON file.

  • The `-p` option is followed by the path to the **src** directory of the sandbox.

For example, if the sandbox name is SamplePyApp:

```
user@jet-vm:~/SamplePyApp/src$ jet-pkg-gen.py -I /home/user/pyapp/src/SamplePyApp.json -p /
home/user/pyapp/src
```

> **(i)** **NOTE**: The autogenerated application makefile will be correct in most cases. If there are any external library dependencies, adjust the makefile accordingly.

8. Build the entire package.

   First, return to the **src** directory (**$SB/src**). Next, run the `mk-i386` *SamplePyApp* command, where *SamplePyApp* is the `"app-name"` from the JSON file in Step 5.

   ```
   user@jet-vm:/home/user/pyapp/src$ mk-i386,bsdx SamplePyApp
   ```

   > **(i)** **NOTE**: Starting in Junos OS Release 20.2R1, if you will be running your JET application on a ACX710 device, you can use the Clang toolchain for ARM-based compilation of JET applications written in C, Python, or Ruby. Use the command `mk-arm,bsdx` instead of `mk-i386,bsdx` to use the Clang toolchain to compile your application.

9. Copy the package onto a device running Junos OS.

   ```
   user@jet-vm:/home/user/pyapp/src$ scp../junos-jet-sb-obj/ship/SamplePyApp-
   x86-32-20180828.231545_user.tgz root@device:/var/tmp/
   ```

Now configure the Junos device and install the package.

1. Enter configuration mode.

   ```
   root@device> configure
   Entering configuration mode
   [edit]
   root@device#
   ```

2. Configure the application's provider's ID, license type, and deployment scope on the Junos device, if necessary. Use the same provider license that you used to package it.

   ```
   root@device# set system extensions providers 12345 license-type juniper deployment-scope
   commercial
   ```

   For more information, see Configuring the JET Application and its License on a Device Running Junos OS.

3. Exit to operational mode and install the copied package on the Junos device.

```
root@device# exit
root@device> request system software add /var/tmp/SamplePyApp-x86-32-20180828.231545_user.tgz
```

> **ⓘ NOTE**: This step will fail if `providers` is not configured.

4. Verify the package was installed successfully.

```
root@device> show version

Hostname: device
Model: mx480
…
…
JET app SamplePyApp [20180828.231545_user]
```

5. Enter configuration mode.

```
root@device> configure
Entering configuration mode
[edit]
root@device#
```

6. Configure the command-line arguments through the Junos OS CLI. If a Python JET script is available in the **/var/db/scripts/jet/** directory on a device running Junos OS, you can configure command-line arguments for the file and supply the arguments from the Junos CLI.

Here are the arguments in the application.

```
import argparse

def main():
    parser = argparse.ArgumentParser(description='This is a demo script.')

    parser.add_argument('-arg1', required=True)
    parser.add_argument('-arg2', required=True)
```

```
    args = parser.parse_args()

    print args.arg1
    print args.arg2

if __name__ == '__main__':
    main()
```

Configure the command-line arguments in the CLI. In this example, the script filename is
`cmdline_args.py`.

```
root@device# set system extensions extension-service application file cmdline_args.py
arguments "-arg1 jet -arg2 application"
```

7. Commit the configuration and exit to operational mode.

```
root@device# commit
root@device# exit
```

8. Run the application.

```
root@device> request extension-service start cmdline_args.py

Extension-service application 'cmdline_args.py' started with PID: 12345
jet
application
```

Example: Develop a Signed Python Package With C Dependencies

After you have set up the JET VM and acquired a signing certification, you are ready to create the development sandbox in the VM and start developing your signed, on-device application. Use this example to develop a signed Python package with C dependencies.

1. Check out the sandbox.

```
user@jet-vm:/user/jet-trial-apps/signed-python-with-c-dep$ mksb -n PyAppC
```

2. Create an application directory in the Python subdirectory.

```
user@jet-vm:/user/jet-trial-apps/signed-python-with-c-dep/PyAppC/src/python$ mkdir pyappc
```

3. A bitarray is an example of a relatively simple Python module with a C dependency. Download and extract the bitarray from https://pypi.org/project/bitarray/ into the Python application directory.

```
user@jet-vm:/user/jet-trial-apps/signed-python-with-c-dep/PyAppC/src/python/pyappc$ ls
bitarray

_bitarray.c  __init__.py
```

This is an example of a simple Python application that uses a bitarray module:

```
user@jet-vm:/user/jet-trial-apps/signed-python-with-c-dep/PyAppC/src/python/pyappc$ cat
bitarray_app.py
from bitarray import bitarray

a = bitarray()
a.append(True)
a.extend([False, True, False])
print a
```

If you are writing your application using Python 3, include the PASS keyword in the Exception block of the script.

```
    except Exception as tx:
        pass
```

4. Create the JSON configuration file that references the external source files. See "Package JET Applications" on page 41 for more information.

```
user@jet-vm:/user/jet-trial-apps/signed-python-with-c-dep/PyAppC/src$ cat pyappc.json

{
    "app-name" : "PyAppC",
    "app-path" : "python/pyappc",
    "language" : "python",
    "main-scripts" : ["bitarray_app.py"],
    "app-type" : "standalone",
```

```
    "sign" : "yes",
    "os-type" : "bsd11",
    "target-arch" : "i386",
    "description" : "Simple Python App with C dependencies",

    "srcs" : {
                "python/pyappc/bitarray" : ["__init__.py"]
    },
    "extn-srcs" : {
                "python/pyappc/bitarray" : ["_bitarray.c"]
    }
}


user@jet-vm:/user/jet-trial-apps/signed-python-with-c-dep/PyAppC/src$ jet-pkg-gen.py -I /
user/jet-trial-apps/signed-python-with-c-dep/PyAppC/src/pyappc.json -p /user/jet-trial-apps/
signed-python-with-c-dep/PyAppC/src
```

5. Run the following command to create the necessary makefiles and the manifest file that locates the files on the Junos device when the package is installed.

```
/user/jet-trial-apps/signed-python-with-c-dep/PyAppC/src/pkgs/PyAppC/contents.manifest


user@jet-vm:/user/jet-trial-apps/signed-python-with-c-dep/PyAppC/src$ cat ./pkgs/PyAppC/
contents.manifest.orig


/set package_id=31 role=Provider_Daemon
%TOPDIR%/python/pyappc/bitarray_app.py store=%INSTALLDIR%/bitarray_app.py mode=555
program_id=1
%TOPDIR%/python/pyappc/bitarray/__init__.py store=%PYTHON_MOD_INSTALLDIR%/PyAppC/bitarray/
__init__.py mode=555 program_id=1
%TOPDIR%/python/pyappc/_bitarray.so store=%PYTHON_MOD_INSTALLDIR%/PyAppC/_bitarray.so
mode=555 program_id=1
```

6. To locate the bitarray module on a Junos device, add the **/PyAppC/** path for the **__init__.py** file and the **bitarray/** directory path for the **_bitarray.so** file.

```
user@jet-vm:/user/jet-trial-apps/signed-python-with-c-dep/PyAppC/src$ cat ./pkgs/PyAppC/
contents.manifest


/set package_id=31 role=Provider_Daemon
%TOPDIR%/python/pyappc/bitarray_app.py store=%INSTALLDIR%/bitarray_app.py mode=555
program_id=1
```

```
%TOPDIR%/python/pyappc/bitarray/__init__.py store=%PYTHON_MOD_INSTALLDIR%/bitarray/
__init__.py mode=555 program_id=1
%TOPDIR%/python/pyappc/_bitarray.so store=%PYTHON_MOD_INSTALLDIR%/bitarray/_bitarray.so
mode=555 program_id=1
```

7. Build and package the application.

```
user@jet-vm:/user/jet-trial-apps/signed-python-with-c-dep/PyAppC/src$ mk-i386,bsdx PyAppC
```

> **NOTE**: Starting in Junos OS Release 20.2R1, if you will be running your JET
> application on a ACX710 device, you can use the Clang toolchain for ARM-based
> compilation of JET applications written in C, Python, or Ruby. Use the command `mk-arm,bsdx` instead of `mk-i386,bsdx` to use the Clang toolchain to compile your application.

8. Copy the built package onto the device running Junos OS.

```
user@jet-vm:/user/jet-trial-apps/signed-python-with-c-dep/PyAppC/src$ scp /user/jet-trial-apps/signed-python-with-c-dep/PyAppC/junos-jet-sb-obj/ship/
PyAppC-x86-32-20180829.211252_user.tgz root@device:/var/tmp/
```

9. Configure the application's provider's ID, license type, and deployment scope on the Junos device, if necessary. Use the same provider license that you used to package it.

```
root@device# set system extensions providers 12345 license-type juniper deployment-scope
commercial
```

For more information, see Configuring the JET Application and its License on a Device Running Junos OS.

10. Install the package on the device running Junos OS.

```
root@device> request system software add PyAppC-x86-32-20180830.031354_user.tgz
```

Once the package is installed successfully, the dependent Python module and the C shared library are installed on the device in the standard Python module path as specified in the manifest file.

```
root@device:/opt/lib/python2.7/site-packages # ls bitarray/__init__.py    _bitarray.so
```

11. Add the application in configuration mode.

```
root@device# set system extensions extension-service application file bitarray_app.py
[edit]
root@device# commit
commit complete
```

12. Run the application

```
root@device> request extension-service start bitarray_app.py

Extension-service application 'bitarray_app.py' started with PID: 12345
bitarray('1010')
```

Change History Table

Feature support is determined by the platform and release you are using. Use Feature Explorer to determine if a feature is supported on your platform.

Release	Description
20.2R1	Starting in Junos OS Release 20.2R1, if you will be running your JET application on a ACX710 device, you can use the Clang toolchain for ARM-based compilation of JET applications written in C, Python, or Ruby.

Package JET Applications

IN THIS SECTION

After application development is complete, write the JavaScript Object Notation (JSON) file describing the content to build and package the application before deploying it on the device. JSON is a lightweight data-interchange format. It is easy for humans to read and write, and also easy for machines to parse and generate. For more details, see https://www.json.org.

JSON files consist of a collection of attributes are included inside a set of curly braces. Attributes use two structures:

- A collection of key-value pairs.

- An ordered list of values.

Read further to learn about each of the attributes contained in the JSON format for application packaging.

Main Section Attributes

The top block of the JSON file is the main section of the file. It consists of mandatory and optional attributes.

Mandatory Attributes

Table 2 on page 43 describes the mandatory attributes that all JSON files for application packaging must have in the main section. The following is an example of a simple application JSON file containing only the mandatory attributes:

```
{
  "app-name": "testcapp",
  "app-path": "bin/test-c-app",
  "language": "c",
  "app-type": "standalone",
  "sign": "yes",
  "os-type": "bsd10",
  "target-arch": "i386",
  "description": "C Test Application",
  "srcs": {
    "bin/test-c-app": ["test_app.c"]
  }
}
```

Table 2: Mandatory Attributes in the JSON File Main Section

Attribute	Description	Example Values
`"app-name"`	Specify the name of the application.	`"sample_pyapp"`
`"app-path"`	Specify the path to the application's implementation directory. All paths should be relative to sandbox **src**.	`"python/sample_pyapp"`
`"language"`	Specify the language used for developing the application.	`"python"`, `"c"`, `"c++"`
`"main-scripts"`	This is a list attributes. Specify the filename or filenames of the main script or scripts that run on the device (do not specify the module here). The main script files will be deployed under the **/var/db/scripts/jet** path on the device.	`["foo.py", "bar.py"]`
`"app-type"`	Specify whether an application is to be a standalone program or a daemon.	`"standalone"` or `"daemon"`
`"sign"`	Indicate whether the application is to be signed or unsigned.	`"yes"` or `"no"`
`"os-type"`	Specify whether the application is to be deployed on legacy Junos OS (bsd6) or Junos OS with upgraded FreeBSD (bsd10).	`"bsd6"`, `bsd10`, or `"bsd11"`
`"target-arch"`	Specify the target architecture on which the application is to be deployed.	`"i386"`, `"powerpc"`, `"octeon"`, `"xlr"`, or `"arm"`
`"description"`	Write a brief (one-line) description about the application. This will be displayed in the `show version` operational command output.	`"Simple Python test app"`

Optional Attributes

describes the optional attributes you can include in the main section of the JSON file for application packaging. The following is an example main section with mandatory and optional attributes:

```
  {
    "app-name": "sample_pyapp",
    "app-path": "python/sample_pyapp",
    "language": "python",
    "main-scripts": ["foo.py", "bar.py"],
    "app-type": "standalone",
    "sign": "no",
    "os-type": "bsd6",
    "target-arch": "i386",
    "description": "Simple Python test app",
    "c-compiler-flags": "-DFOO  -DBAR",
    "c++-compiler-flags": "-DAPP_CHECK  -DSOMETHING_ON",
    "linker-flags": "-lstdc++ -lfoo"  }
```

Table 3: Optional Attributes in the JSON File Main Section

Attribute	Description	Example Values
`"c-compiler-flags"`	Specify the list of C compiler flags, if any. Compilation flags can be defined for the main section, dependent libraries (dep-libs), or dependent Python modules (dep-py-modules).	"*flag1 flag2 flag3*"
`"c++-compiler-flags"`	Specify the list of C++ compiler flags, if any. Compilation flags can be defined for the main section, dependent libraries (dep-libs), or dependent Python modules (dep-py-modules).	"*flag1 flag2 flag3*"

Table 3: Optional Attributes in the JSON File Main Section *(Continued)*

Attribute	Description	Example Values
`"linker-flags"`	Specify the list of linker flags, if any. Use these flags to specify additional libraries to link to or additional link-specific flags that are required during linking, You can define linker-specific flags either in the main section or in the dep-py-modules section.	`"flag1 flag2 flag3"`

Source Attributes

shows two source attributes you can use to specify source files for the application package. The following is an example Python application with additional module files to be deployed, along with the main script file:

```
{
  "app-name": "sample_pyapp",
  "app-path": "python/sample_pyapp",
  "language": "python",
  "main-scripts": ["foo.py", "bar.py"],
  "app-type": "standalone",
  "sign": "no",
  "os-type": "bsd6",
  "target-arch": "i386",
  "description": "Simple Python test app",

  "srcs": {
    "python/sample_pyapp": ["a.py", "b.py"],
    "python/sample_pyapp/temp": ["temp1.py", "temp2.py"]
  },

  "extn-srcs": {
    "python/sample_pyapp": ["foo.c", "bar.c"],
    "python/sample_pyapp/temp": ["1.cpp", "2.cpp"]
  }
}
```

Table 4: Source Attributes You Can Use in a JSON File

Attribute	Description	Example Values
`"srcs"`	Specify the list of additional source files. For Python applications, these source files are the additional module files. For C or C++ applications, these source files are the source files to be compiled to generate **lib/binary**. Each entry should be a key-value pair, where the key is the path of the source files and the value is an array of source filenames.	```"srcs": { "python/sample_pyapp": ["a.py", "b.py"], "python/sample_pyapp/temp": ["temp1.py", "temp2.py"] }```
`"extn-srcs"`	This section is applicable only for Python. Specify the list of C or C++ module files to be compiled. Each entry should be a key-value pair, where the key is the path of the source files and the value is an array of source filenames.	```"extn-srcs": { "python/sample_pyapp": ["foo.c", "bar.c"], "python/sample_pyapp/temp": ["1.cpp", "2.cpp"] }```

Dependent Libraries

You must compile any dependent libraries available in the dependent libraries (`dep-libs`) section. The library generated from this JSON code is packaged with the application. The `dep-libs` section is an array of multiple library dependencies, each composed of the following key-name pairs:

- `"lib-name"` is the name of the library.

- `"lib-path"` is the path of the library source code in the development sandbox.

- `"srcs"` is a key-value pair in which the path is the key and its value is a list of source files.

The following is an example of a `dep-libs` attribute:

```
"dep-libs": [
  {
```

```
      "lib-name": "xyz",
      "lib-path": "lib/xyz",
      "srcs": {
        "lib/xyz": ["foo.c", "bar.c"]
      }
    }
    ]
```

Dependent Python Modules

The dependent Python modules (`dep-py-modules`) attribute is used only for Python applications. This attribute contains any dependent Python modules that need to be compiled and packaged with the application. The `dep-py-modules` attribute is an array in which you can specify multiple Python module dependencies. Each dependency is composed of the following objects:

- `"py-module-name"` is the name of the Python module.

- `"py-module-path"` is the path of the Python module source code in the development sandbox.

- `"srcs"` is a key-value pair in which the path is the key and its value is a list of source files.

- `"extn-srcs"` is a key-value pair in which the path is the key and its value is a list of Python extension source files.

The following is an example of a `dep-py-modules` attribute:

```
"dep-py-modules": [
    {
        "py-module-name": "module_a",
        "py-module-path": "python/module_a",
        "srcs": {
          "python/module_a": ["foo.py", "bar.py"]
        },
        "extn-srcs": {
          "python/module_a": ["foo.c", "bar.c"],
          "python/module_a/sub_mod": ["lmn.cpp"]
        }
    }
    ]
```

Debug JET Applications

IN THIS SECTION

Use this topic to debug JET applications.

Debugging Tips

- For debugging applications on a device running Junos OS, you can configure the trace file option with the `edit system services extension-service traceoptions` statement. You need to enable this statement on the Junos OS device before writing the sample applications.

- The Junos service process (jsd) is supported only on the Routing Engine running in primary mode. It is not supported on the backup Routing Engine.

- To eliminate any firewall issues, use an on-device application to test.

- For notification applications, verify that your client IP source address (the address from which the connection is established) is added to the list of allowed clients in the jsd notification configuration.

- Ensure that the maximum number of notification connections does not exceed the number configured on the device. Use the following command to see the clients:

```
netstat -a | grep 1883
```

How to Invoke the Debugger During Install

For non-daemonized applications that run on the router, you can invoke the debugger at the same time that you install the application. To load your application along with the debugger:

1. Use the Junos OS CLI to invoke the debugger and install the application at the same time.

```
user@device> request extension-service start invoke-debugger cli application-name.py

Extension-service application 'application-name.py' started with pid: 12345
```

2. Enter `help` to display a list of the supported commands.

```
(Pdb) help

Documented commands (type help <topic>):
========================================
EOF    bt         cont      enable  jump  pp       run      unt
a      c          continue  exit    l     q        s        until
alias  cl         d         h       list  quit     step     up
args   clear      debug     help    n     r        tbreak   w
b      commands   disable   ignore  next  restart  u        whatis
break  condition  down      j       p     return   unalias  where
Miscellaneous help topics:
==========================
exec  pdb
Undocumented commands:
======================
retval  rv
```

3. Use the debugger commands as needed by typing **help <topic>**.

Issue: Cannot Connect to jsd

Use this procedure if your application cannot connect to jsd.

1. Check whether jsd is up and running on the Junos OS device using the following command:

```
ps aux | grep jsd
```

2. If jsd is not up, restart jsd. Choose from the following options:
   - gracefully—Gracefully restart the process.
   - immediately—Immediately restart (SIGKILL) the process.
   - soft—Soft reset (SIGHUP) the process.
   - |—Pipe through a command.

```
user@device# restart jsd <gracefully | immediately | soft>
```

3. If jsd is up, verify the configuration is present on the device using the following command:

```
user@device# show system services extension-service
```

You should see the configuration in the output. If you do not, redo the configuration.

4. If the configuration is present, verify jsd is listening on configured port 51051:

```
nestat -a | grep 51051
```

You should see a matching entry.

5. If you do not see a matching entry, restart jsd.

```
user@device# restart jsd <gracefully | immediately | soft>
```

RELATED DOCUMENTATION

3

CHAPTER

Additional Resources

Additional Resources

- Expert Advice: Junos Extension Toolkit (JET)

- FAQ: Learning About JET Part 1—Python on Junos OS

- FAQ: Learning About JET Part 2—JavaScript Object Notation (JSON)

- FAQ: Learning About JET Part 3—JET APIs

- FAQ: Learning About JET Part 4—Fast Programmatic Configuration

4
CHAPTER

Configuration Statements and Operational Commands

Junos CLI Reference Overview | 54

Junos CLI Reference Overview

We've consolidated all Junos CLI commands and configuration statements in one place. Learn about the syntax and options that make up the statements and commands and understand the contexts in which you'll use these CLI elements in your network configurations and operations.

- Junos CLI Reference

Click the links to access Junos OS and Junos OS Evolved configuration statement and command summary topics.

- Configuration Statements

- Operational Commands