



Sicheren Puppet Code entwickeln und testen mit PDK

Thomas Krieger
Senior Sales Engineer DACH & CEE




Agenda

- PDK, was ist denn das?
- PDK in der Praxis
- PDK updaten
- PDK anpassen
- Sicherer Puppet Code



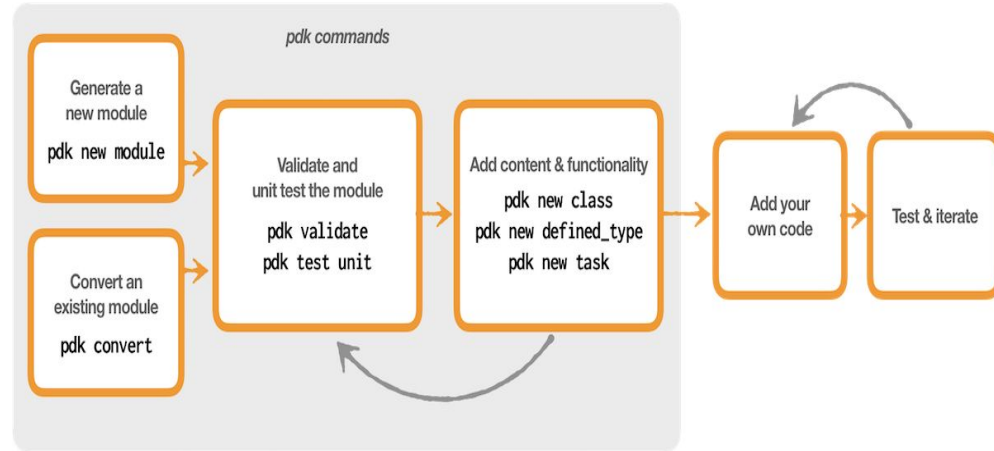
Thomas Krieger
Senior Sales Engineer,
DACH & CEE

PDK, was ist denn das?

- PDK = **P**uppet **D**evelopment **K**it
- Komplette Puppet Module mit Klassen, defined Types, Tasks, Facts usw. erstellen
- Puppet Code validieren (Syntax checks)
- Puppet Code testen (Unit tests)
- anpassbar an eigene Bedürfnisse (Templates oder .sync.yml)
- bereits existierende Module können mittels PDK konvertiert werden, um PDK nutzen zu können
- Erweiterbar, z. B. um Syntax Validierungen für sicheren Code
- PDK 2.4.0 wird im folgenden verwendet 

PDK Workflow

- Puppet Modul erstellen oder existierendes Modul konvertieren
- Code validieren (puppet-lint)
- Unit Tests ausführen (rspec-puppet)
- Klassen, defined Types, Tasks, Facts usw. generieren
- eigenen Code schreiben
- eigenen Code validieren und testen
- Rinse and repeat



Modul anlegen

```
pdk new module <Modulname> [--skip-interview] [--template-url]  
                                [--template-ref]
```

- erzeugt ein Modul im aktuellen Verzeichnis und erstellt dazu ein Verzeichnis mit dem Modulnamen. Dieses Modulverzeichnis darf nicht existieren.
- die Interview Fragen befassen sich mit der Lizenz, dem Autor, dem Forge Benutzernamen, den unterstützten Betriebssystemen usw. und können mit `--skip-interview` übersprungen werden.
- sofern kein eigenes Template angegeben wird, wird das PDK Default Template verwendet
- `metadata.json` überprüfen, Modul Abhängigkeiten und unterstützte OS Versionen eintragen
- `pdk new --help` listet alles Sub Kommandos auf

Modul konvertieren

```
pdk convert [--add-tests] [--noop] [--skip-interview] --template-url]
            [--template-ref] [--noop] [--add-tests]
```

- Backup des Moduls anfertigen, da die Konvertierung Dateien ändert
- Befehl muss im Modulverzeichnis ausgeführt werden
- die Interview Fragen befassen sich mit der Lizenz, dem Autor, dem Forge Benutzernamen, den unterstützten Betriebssystemen usw. und können mit `--skip-interview` übersprungen werden.
- sofern kein eigenes Template angegeben wird, wird das PDK Default Template verwendet
- `--add-tests` fügt Unit Tests in das konvertierte Modul ein
- `--noop` gibt aus was die Konvertierung tun würde (Unit Tests werden hier nicht ausgegeben, diese werden erst bei der Konvertierung erstellt)

Klasse anlegen

```
pdk new class <Klassen-Name>
```

- Befehl muss im Modulverzeichnis ausgeführt werden
- die `init.pp` Klasse erhält man, indem die Klasse genauso wie das Modul benannt wird
- es wird eine `<Klassen-Name>.pp` Datei für die Klasse angelegt
- neben der Klasse `<Klassen-Name>.pp` wird auch eine Datei `<Klassen-Name>_spec.rb` für die Unit Tests im `spec/classes` Verzeichnis erstellt
- Klassen in Unterverzeichnissen werden über den Namensraum erzeugt, z. B.

```
pdk new class misc::myclass
```


Fehlende Unterverzeichnisse werden automatisch angelegt (auch für Unit Tests)
- Unit Tests werden nur rudimentär angelegt (das Modul muss nur fehlerfrei kompilieren)

Defined Type anlegen

```
pdk new defined_type <Typ-Name>
```

- Befehl muss im Modulverzeichnis ausgeführt werden
- es wird eine Datei für den defined Type `<Typ-Name>.pp` angelegt
- neben dem defined Type `<defined_type_name>.pp` wird auch eine Datei `<defined_type_name>_spec.rb` für die Unit Tests im `spec/defines` Verzeichnis erstellt
-

Fact anlegen

```
pdk new fact <Fact-Name>
```

- Befehl muss im Modulverzeichnis ausgeführt werden
- er wird eine Datei `<Fact-name>.rb` für den Fact im Verzeichnis `lib/facter` angelegt
- neben dem Fact `<Fact-name>.rb` wird auch eine Datei `<Fact-name>_spec.rb` für die Unit Tests im `spec/unit/facter` Verzeichnis erstellt
- Fehlende Unterverzeichnisse werden automatisch erstellt

Function anlegen

```
pdk new function <Function-Name>
```

- Befehl muss im Modulverzeichnis ausgeführt werden
- er wird eine Datei `<Function-name>.pp` für den Fact im Verzeichnis `functions` angelegt
- neben dem Fact `<Function-name>.pp` wird auch eine Datei `<Function-name>_spec.rb` für die Unit Tests im `spec/functions` Verzeichnis erstellt
- Fehlende Unterverzeichnisse werden automatisch erstellt

Provider anlegen

Diese Funktion ist noch experimentell!

```
pdk new provider <Provider-Name>
```

- Befehl muss im Modulverzeichnis ausgeführt werden
- er wird eine Datei für den Provider und den Type angelegt, fehlende Unterverzeichnisse werden angelegt
- der generierte Provider nutzt die Resource API
- neben dem Provider und dem Type werden auch eine Dateien für die Unit Tests im `spec/unit/puppet/provider` und `spec/unit/puppet/type` Verzeichnis erstellt

Anpassen der Datei `.sync.yml` ist erforderlich.
Folgendes muss vorhanden sein:

```
Gemfile:
  optional:
    ':development':
      - gem: 'puppet-resource_api'
```

Ausführen von `pdk update` um die Änderungen zu übernehmen

Task anlegen

```
pdk new task <Task-Name>
```

- Befehl muss im Modulverzeichnis ausgeführt werden
- es wird ein Skript `<task-name>.sh` und eine Datei mit Metadaten `<task-name>.json` für die Task im Verzeichnis `tasks` angelegt
- für Tasks werden keine Unit Tests erzeugt

Syntax checks ausführen

```
pdk validate [-a][--parallel]
```

- Befehl muss im Modulverzeichnis ausgeführt werden
- PDK führt alle vorhandenen Validatoren aus. Werden Fehler festgestellt, werden diese ausgegeben
- einfach zu korrigierende syntaktische Fehler, wie z. B. falsche Einrückungen können vom PDK automatisch korrigiert werden (`pdk validate -a`). Die Korrekturen werden ausgegeben, die betroffenen Dateien geändert. PDK kann auch Ruby Code (z. B. Unit Tests) formatieren. Kann PDK nicht automatisch korrigieren, gibt es Vorschläge aus, wie z. B. Ruby Code verbessert werden kann, um sich an geltende Konventionen zu halten.
- `--parallel` führt die Validatoren parallel aus

Unit Tests ausführen

```
pdk test unit [-v] [--parallel]
[--tests]
```

- Befehl muss im Modulverzeichnis ausgeführt werden
- PDK führt alle Unit Tests aus (ohne --tests)
- -v erzeugt einen detaillierten Output
- --parallel nutzt alle zur Verfügung stehenden Ressourcen, um so viele Unit Tests wie möglich parallel auszuführen
- --tests <spec-file> dient dazu, nur bestimmte Unit Tests auszuführen

Output mit -v:

```
meetup::service
  on ubuntu-20.04-x86_64
    is expected to contain Service[meetup] with
ensure => "running" and enable => true

    on centos-8-x86_64
      is expected to contain Service[meetup] with
ensure => "running" and enable => true

    on redhat-8-x86_64
      is expected to contain Service[meetup] with
ensure => "running" and enable => true

    on ubuntu-18.04-x86_64
      is expected to contain Service[meetup] with
ensure => "running" and enable => true
```

PDK in Aktion

PDK in der Praxis:

- Konvertieren eines bestehende Moduls
- Erstellen eines Moduls `'meetup'` für Redhat like und Debian like Betriebssysteme
- `metadata.json` und `.fixtures.yml` anpassen, einbinden von Puppet Modulen wie `Stdlib` und `Concat`
- Erstellen einer Klasse `init.pp` und einer Klasse `service.pp` mit Unit Tests
- Erstellen eines defined Types `sec_test` mit Unit Tests
- Syntax Validierung
- Unit Tests schreiben (rspec) und ausführen



PDK updaten

Was kann ich tun, wenn mir PDK sagt, dass das Modul mit einer älteren PDK Version erstellt wurde?

- Voraussetzung, eine neue Version des PDK wurde bereits installiert
- verwendete PDK Version steht in der Datei `metadata.json`:

```
"pdk-version": "2.4.0",  
"template-url": "pdk-default#2.4.0",  
"template-ref": "tags/2.4.0-0-gfa6b6d2"
```
- `pdk update [--template-ref=<Branch oder Tag>]`
nimmt alle definierten Anpassungen aus der Datei `.sync.yml` beim Update vor. Wurde ein Template verwendet, kann ein neuer Branch oder ein neuer Tag angegeben werden. Die Änderungen werden in diesem Fall aus dem Repository gezogen (siehe `template-url`)

PDK anpassen

Zwei Möglichkeiten, das PDK anzupassen

- Datei `.sync.yml` im Modulverzeichnis anpassen
 - Anpassungen für ein einzelnes oder wenige Module mittels `pdk update`
- eigenes PDK Template verwenden
 - dazu kann das Puppet PDK Template Git Repository in Github benutzt werden
<https://github.com/puppetlabs/pdk-templates>
 - Sinnvoll als Unternehmens-Standard oder als Standard für eigene Module
 - `pdk new module xyz -template-url=https://<repo url> -template-ref=<Branch oder Tag>`

Sicherer Puppet Code

Schwachstellen und böse Überraschungen vermeiden



Sicherer Puppet Code

Wie schreibe ich sicheren Puppet Code?

Traue niemandem, der Dein Puppet Modul oder Deinen Puppet Code verwendet!

- Parameter validieren und auf Plausibilität prüfen, bei Fehlern den Katalog mit `fail` abbrechen
- Parameter wenn möglich über Standard-Typen (Integer, Boolean, Enum) oder selbst geschriebene Typen absichern (spart Code zur Validierung)
- https verwenden wo immer möglich, Apache oder Nginx nur mit https einsetzen (wenn möglich)
- bei Dateien und Verzeichnissen User, Group und Rechte sinnvoll und vorsichtig setzen (lean rights management: wer muss darauf zugreifen, welche Rechte braucht er)
- Rule of Thumb: ca. 70% - 80% des Codes entfallen meiner Erfahrung nach auf Prüfung von Parametern und Daten, Plausibilitätsprüfungen, Fehlerbehandlung

PDK mit Security Validierung erweitern

Weiteres Lint Modul einbinden

Das Gem `puppet-lint-security-plugins` (verschiede Sicherheitsprüfungen) mit Hilfe von `.sync.yml` in das Modul `meetup` einbinden:

- Das Gem wird während `pdk validate` ausgeführt und prüft auf Schwachstellen im Code. Werden Schwachstellen gefunden, wird der `pdk validate` Lauf als fehlerhaft betrachtet

- Die Datei `.sync.yml` wird erweitert:

```
---
Gemfile:
  optional:
    ':system_tests':
      - gem: 'puppet-lint-security-plugins'
```

- Mit `pdk update` werden die Änderungen im Modul vorgenommen
- Syntax Validierung und Unit Tests ausführen um auf vorhandene Schwachstellen zu prüfen
- Erstellen eines defined Types `sec_test` mit einer Schwachstelle in unserem `meetup` Modul
`pdk new defined_type sec_test`

Warum Unit Tests wichtig sind

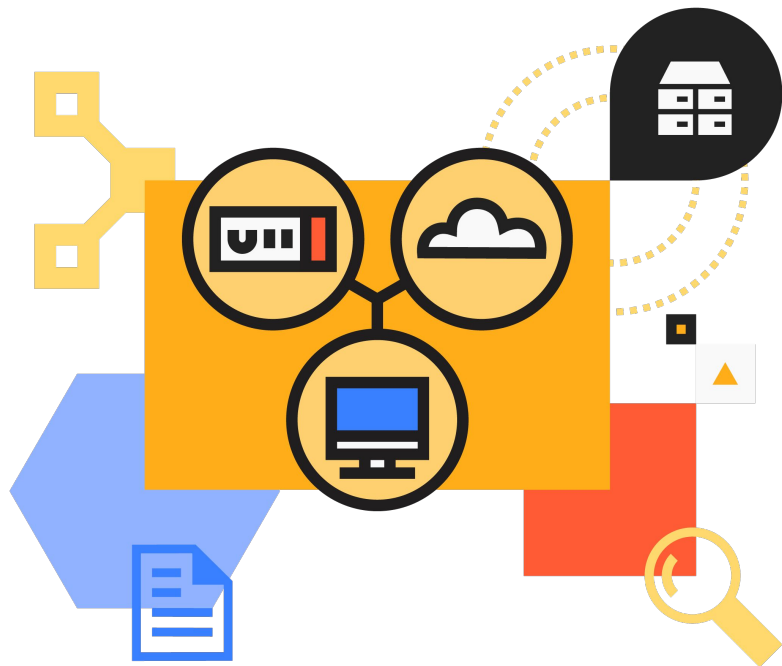
Eine kleine Story, wie mir Unit Tests viel unnötige Arbeit erspart haben



Puppet Continuous Delivery for Puppet Enterprise

Pipelines mit Syntax Checks und Unit Tests:

- frei definierbare Pipelines, eigene Jobs können definiert werden
- Impact Analyse
- Code Deployment in PE



Nützliche Links

Oder wo finde ich was?

Puppet Lint Security Plugins:

<https://github.com/floek/puppet-lint-security-plugins>

Puppet PDK Template Repository:

<https://github.com/puppetlabs/pdk-templates>

PDK Dokumentation:

<https://puppet.com/docs/pdk/2.x/pdk.html>

Puppet Modul mit vielen Unit Tests:

<https://github.com/voxpupuli/puppet-dhcp>

Meetup Repository:

<https://github.com/pugdach>



Vielen Dank



thomas.krieger@puppet.com



+49 175 20 20 297



@TomK / Puppet Community Slack