Continuous Delivery Teil 2 – Ansible Tipps

30. Januar 2017 / 0 Kommentare / in Technisches / von Stefan Moises

Im ersten Teil dieser Blogserie [http://www.shoptimax.de/blog/allgemeines/continuous-delivery-mit-docker-gitlab-ci-und-ansible] haben wir den allgemeinen Entwicklungsworkflow bei shoptimax beleuchtet – in diesem Teil soll es um konkrete Tipps zum Deployment mit Ansible [https://www.ansible.com/] gehen.

Gitlab CI und Ansible kombinieren

Wie bereits beschrieben nutzen wir für automatisiertes Deployment unserer Projekte aus GIT heraus Gitlab CI [https://about.gitlab.com/gitlab-ci/] . Je nach GIT- Branch (bei uns sind das normalerweise „develop“, „stage“ und „master“) löst Gitlab CI unterschiedliche Aktionen aus, welche in „.gitlab-ci.yml“-Dateien [https://docs.gitlab.com/ce/ci/quick_start/#creating-a-gitlab-ci-yml-file] in den jeweiligen Projekten definiert sind.
In der Regel werden Änderungen in den „develop“-Branches direkt per Gitlab Runner auf interne Testserver deployed. Optional können hier (und/oder bei einem Git-Push in weitere Branches) Code-Style Prüfungen oder z.B. Codeception [http://codeception.com/] -Tests durchgeführt werden, die dann in einem Gitlab-internen Docker-Container automatisch gestartet werden.

Änderungen an „stage“- oder „master“-Branches hingegen werden in der Regel auf externe Server ausgespielt, sei es ein geteilter Staging-Server von shoptimax oder direkt ein Kunden-System. Hier kommt dann unser Ansible-Repository mit unterschiedlichen sog. „Playbooks“ ins Spiel.

Dazu wird im ersten Deployment-Step (welcher im Docker-Container abläuft) unser internes Ansible-GIT-Repository gecloned. Dieses wird am Ende des Durchlaufs in einem sog. **Gitlab CI-Artefakt** gespeichert, welches in weiteren Deployment-Steps dann ebenfalls zur Verfügung steht. Damit können nacheinander die unterschiedlichen Playbooks in den verschiedenen „Stages“ des Deployments aufgerufen werden. In einer „scripts“-Sektion der .gitlab-ci.yml wird also z.B. folgendes definiert:

```
- git clone --depth 1 --branch master git@repo:repo_group/ansible
```

Für das zu erstellende Artefakt wird weiterhin festgelegt, welche Verzeichnisse darin gespeichert werden sollen und wie lange es auf dem Gitlab-Server vorgehalten wird, z.B.

```
artifacts:
    paths:
    - build
    - ansible
    - deploy_scripts
    expire_in: 5 days
```

Diese Artefakte können vor Ablauf dieser Zeitspanne übrigens auch über die Gitlab-Oberfläche per Browser heruntergeladen werden.

In der .gitlab-ci.yml wird also definiert, wo das Ansible-Verzeichnis liegen soll und an welchen Stellen im Prozess Ansible Playbooks aufgerufen werden:

```
variables:
  SHOP_BUILD_DIR: "/builds/oxid/$CI_PROJECT_NAME"
deploy_stage:
  stage: deploy
  environment: deploy_stage
  tags:
    - docker
  script:
- cd $SHOP_BUILD_DIR/ansible/
- ansible-playbook -s ./deploy.yml --extra-vars "target=customer1
```

Timestamp-Datei für unabhängige Playbooks

Um eine gemeinsame Referenz für die unterschiedlichen Playbooks zu haben, die ja meist in verschiedenen Build-„Stages“ ablaufen (z.B. lädt das erste Playbook ein *.tar.gz hoch und packt es auf dem Server in Verzeichnis X aus, das zweite Playbook legt dann Symlinks auf dieses Verzeichnis an usw.), wird ein **gemeinsamer Timestamp** verwendet (z.B. auch für die Verzeichnis-Namen der Deployments). Ein solcher Timestamp wird bereits beim Clone des Ansible-Repositories erzeugt und in einer txt-Datei gespeichert:

```
- date +%Y-%m-%d_%H-%M-%S > ansible/timestamp.txt
```

In den Playbooks wird jeweils geprüft, ob es diese Datei gibt, falls ja wird der Timestamp ausgelesen und über diesen Timestamp dann das zugehörige Verzeichnis gesucht / weiter verarbeitet.
Gitlab CI cloned das Ansible Repository in einen projektspezifischen Docker-Container und speichert die Dateien im Build-Artefakt, inkl. der Timestamp-Datei.
Dadurch bleibt der Timestamp während des gesamten Deployment-Prozesses über die verschiedenen, zeitlich nacheinander ablaufenden „Stages“ quasi als „Referenz“ vorhanden und jedes Playbook operiert so auf demselben Verzeichnis.

Projektspezifische Kommandos und Symlinks in JSON-Dateien

Mit Ansible ist es sehr einfach, JSON-Dateien einzulesen und abzuarbeiten. Wir verwenden z.B. JSON-Dateien, um während eines Deployments individuelle Symlinks je nach Kundenprojekt anzulegen oder um auf dem Zielserver projektspezifische Kommandos (Shell oder auch PHP) beim Deployment auszuführen.
Z.B. kann hier ein Shell-Script aufgerufen werden, das eine Datenbank-Sicherung durchführt oder ein PHP-Script, welches das „tmp“-Verzeichnis des Shops leert, Datenbank-Views neu erzeugt oder bestimmte Module automatisch de-/aktiviert usw.
Damit müssen wir keine Fallunterscheidungen oder zusätzliche Variablen und Tasks in Ansible selbst definieren, sondern können in das Kundenprojekt einfach entsprechende JSON-Dateien in das GIT-Repository einchecken. Diese können sogar z.B. pro Git-Branch oder Zielsystem unterschiedlich sein, da die Pfade bzw. Dateinamen der JSON-Dateien im jeweiligen Deployment-Step in der „.gitlab-ci.yml“-Datei als Kommandozeilen-Parameter an das Ansible Playbook übergeben werden können.

```
- ansible-playbook -s ./deploy.yml --extra-vars "target=customer1
```

So kann es z.B. eine *„commands_live.json“* und eine *„command_stage.json“* im Projekt geben, in welchen jeweils unterschiedliche Shell- oder PHP-Kommandos definiert sind.

Um z.B. eine JSON-Datei mit einem Array von Kommandos einzulesen, verwenden wir im Ansible-Playbook folgendes:

```
commands_path: "{{ custom_commands_path | default('/files/cmds.js
commands_json: "{{ lookup('file','{{ inventory_dir }}{{ commands_
```

Die Variable *„custom_commands_path“* (Pfad zur JSON-Datei) kann per Kommandozeilen-Parameter projekt-spezifisch übergeben werden, der aufgerufene Jinja2-Filter [http://docs.ansible.com/ansible/playbooks_filters.html#filters-for-formatting-data] **„from_json“** liest das JSON aus der Datei in eine Ansible-Variable ein.

Das erhaltene JSON kann dann in einer Schleife (**„with_items“**, s.u.) abgearbeitet werden, die erkannten Kommandos werden dann ausgeführt. Hier ein Beispiel für eine solche JSON-Datei:

```
{
  "commands":[
    {"path":"/application/views/flow/", "ctype":"shell", "cmd":"l
    {"path":"", "ctype":"php", "cmd":"oxid cache:clear" }
 ]
}
```

Zudem kann man definieren, ob man ein Kommando „prerelease“, also vor dem eigentlichen „Aktivieren“ des Deployments (z.B. ein Backup erstellen) oder aber erst hinterher (z.B. den Cache löschen) ausführen möchte. Die Variable „host“ schränkt die Aktion optional z.B. bei einem Multi-App-Server System auf einen bestimmten Host ein.

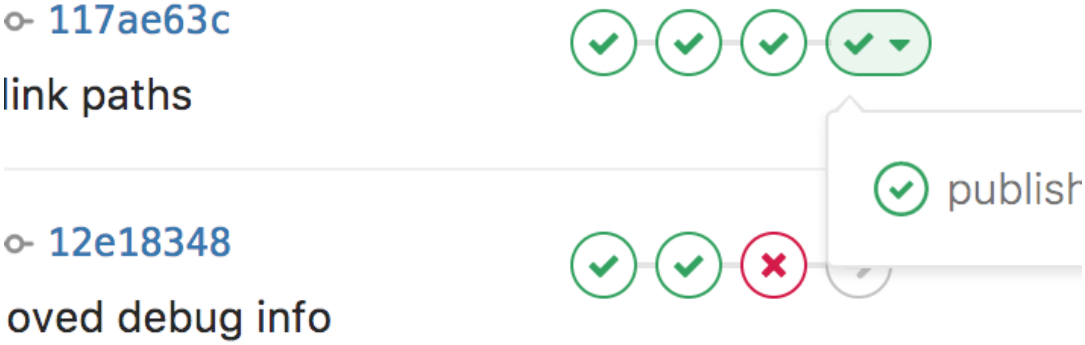
Hier ein Ansible-Task, welcher **„post-release“ Shell-Kommandos** ausführt:

```
- name: Additional Release Shell commands
  shell: "{{ item.cmd }}"
  args:
    chdir: "{{ release_dir }}/{{ release_subfolder }}{{ item.path
  with_items: "{{ commands_json.commands }}"
  register: command_result
  when:
    - item.ctype != 'php'
    - ((item.stage is defined) and (item.stage != 'prerelease'))
    - ((item.host is defined) and (item.host == ansible_nodename
```

Patchen von .htaccess-Dateien

Beim Release eines Live-Shops will man in manchen Fällen den Shop kurzzeitig für Kunden „sperren“, damit die Agentur und/oder der Shopbetreiber den Shop testen, Testbestellungen ausführen usw. können. Dazu verwendet man in der Regel einen Eintrag in der „.htaccess“-Datei im Shop-Verzeichnis, welche den Zugriff auf die Website / den Shop z.B. nur für bestimmte IPs erlaubt und andere Benutzer auf einen sog. Baustellen-/Offline-Seite weiterleitet oder einen Passwort-Schutz ergänzt.

Unsere Ansible-Playbooks können optional in einem ersten Deployment-Schritt **den Inhalt einer frei definierbaren Datei in die .htaccess-Datei des Shops einfügen**, welche z.B. eine bestimmte IP-Freigabe enthält. Dann kann, nach Freigabe bzw. Aktivierung des neuen Deployments über das Gitlab CI Backend die Agentur bzw. der Shopbetreiber den Shop testen, um im Erfolgsfall schliesslich ebenfalls über einen Button im Gitlab-CI Backend das letzte Playbook zu starten („publish“). Dadurch wird der zusätzliche Eintrag wieder aus der .htaccess-Datei entfernt. Im Gitlab CI Backend sieht das z.B. so aus:



[http://www.shoptimax.de/wp-content/uploads/2017/01/Screenshot-2017-01-25-22.24.16.png]

Die vier Haken im Bild sind die Build-„Stages“, die in der „.gitlab-ci.yml“-Datei definiert sind.
In *Stage 1* wird der Shop in einem internen Docker-Container „zusammengebaut“ und ein Artefakt inkl. Ansible-Playbooks erstellt, ab Stage 2 wird Ansible aufgerufen („deploy“ – „release“ – „publish“).
In *Stage 2* wird das neue Release via Ansible auf den Server hochgeladen und ein Eintrag in die .htaccess-Datei eingefügt.
In *Stage 3* wird der Shop-Symlink auf das neue Deployment-Verzeichnis gesetzt und in *Stage 4* wird dann schliesslich der .htaccess-Eintrag wieder entfernt und der Shop ist damit wieder frei zugänglich.

Das **Einfügen in die .htaccess-Datei** sieht im Playbook wie folgt aus:

```
- name: Add custom content to .htaccess file
  blockinfile:
    dest: "{{ release_dir }}/{{ release_subfolder }}.htaccess"
    #insertbefore: BOF
    insertbefore: "#ANSIBLE_PLACEHOLDER#"
    block: |
          {{ htaccess_include }}
    marker: "# {mark} ANSIBLE BLOCK"
  when:
    - shop_htaccess_final.stat.exists == True
    - (custom_htaccess_include_rel_path is defined) and (htaccess
```

Die Variable *„htaccess_include“* enthält den Inhalt der zu inkludierenden Datei, sofern vorhanden. Der **Pfad zur Include-Datei** kann wieder auf der Kommandozeile übergeben werden:

```
- ansible-playbook -s ./deploy.yml --extra-vars "... custom_htacc
```

Entfernt wird der zusätzliche Inhalt am Ende wieder mit folgender Task-Definition:

```
- name: Remove included content from .htaccess file
  blockinfile:
    dest: "{{ release_dir }}/{{ release_subfolder }}.htaccess"
    marker: "# {mark} ANSIBLE BLOCK"
    content: ""
  when:
    - shop_htaccess_patched.stat.exists == True
```

Wir verwenden einen Marker („*#ANSIBLE_PLACEHOLDER#*„) in den .htaccess-Dateien, vor welchem die Datei dann inkludiert wird. Details dazu finden sich in der Doku zum Ansible „blockinfile"-Modul [https://docs.ansible.com/ansible/blockinfile_module.html] .

Das war es auch schon wieder mit unserem kleinen Ansible-Exkurs, in der nächsten und letzten Folge dieser Deployment-Serie schauen wir dann Gitlab CI genauer unter die Haube!

Schlagworte: ansible, ci, continuous, delivery, deployment, devops, git, gitlab, htaccess, json, oxid

Teile diesen Eintrag

0

ANTWORTEN