scalac

HOW TO CREATE AN AUTOMATION PROCESS USING AWS, IAAS, IAAC.

WHY DO WE NEED ORCHESTRATION?

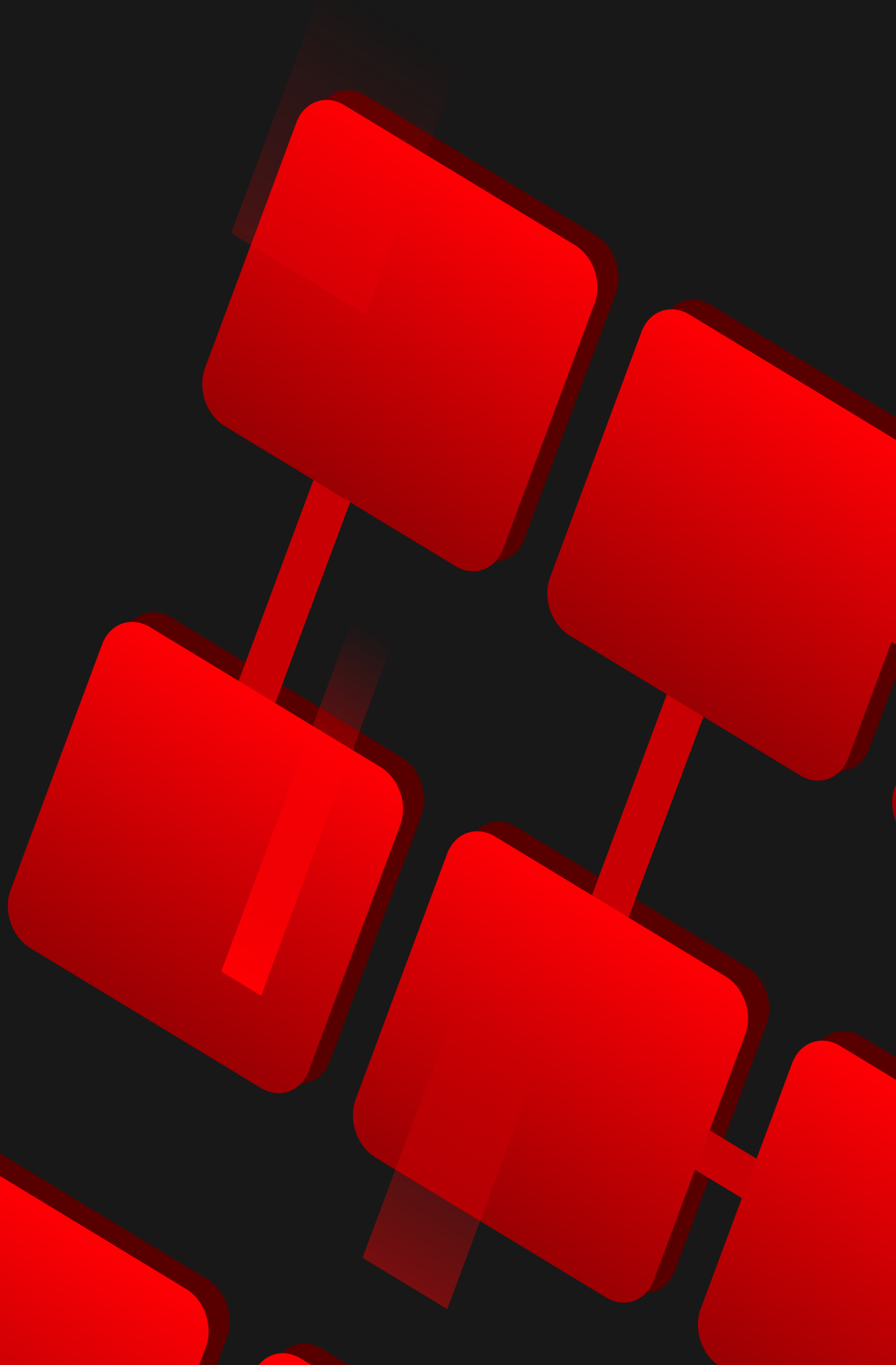


Table of Contents

Why do we need orchestration?

As Federico Garcia Lorca once said, besides black art, there is only automation and mechanization. And though for some of you, the title of this article might sound like black magic indeed, I will do my best to give you a sense of what these concepts mean.

Human nature is not as complicated as it may seem. In general, we are not keen on repeating things over and over again. Our laziness naturally bringths us closer to the automation of everything that might be automated.

> *..hey, are you asleep already? No, no, no... You can't! Because this is where the real story begins!*

Infrastructure in Dark Times

As Federico Garcia Lorca once said, besides black art, there is only automation and mechanization. And though When I joined the Scalac DevOps engineers crew my first assignment was to do some stocktaking and make a note of all the skeletons hidden in the closet. I was thunderstruck. We had two Jenkins instances – working simultaneously – some servers in OVH, some nodes in Microsoft Azure and the rest on a Desktop PC in HQ (also known as the Red Devil). Almost every service was launched on screen. Apart from that, one of the biggest surprises was the mix of production and dev environments on the same servers without any segregation or divisions. One on all, all on one.

Although we didn't have many productive applications, every procedure still resembled open-heart surgery without an anesthetist. In addition, some of the source code was stored on GitHub and some on an internally hosted GitLab. We would have had to keep two sources up and running, and that was not an option.

After a lot of team meetings, we made a strong decision to reorganize it all – codename: kill'em all. We wanted to have everything consolidated so that management, modification, and maintenance were definitively easier than the configuration that had been there from the very beginning.for some of you, the title of this article might sound like black magic indeed, I will do my best to give you a sense of what these concepts mean.

Human nature is not as complicated as it may seem. In general, we are not keen on repeating things over and over again. Our laziness naturally bringths us closer to the automation of everything that might be automated.

And the Phoenix rises from the ashes

For the code repositories, we chose GitHub because a lot of external tools can integrate with it like a charm, and thus the automation process becomes easier.

As we were struggling with different application setups, different configs and various versions all over the servers, Docker became the remedy we needed. With that, we were able to switch locations with the assurance that no matter what the host is, the applications will still work.

Even though our infrastructure was not the biggest, it still required a lot of time to keep it safe and sound. It also called for a lot of manual actions and to be honest, it was a bit like the blind leading the blind across a pedestrian crossing. We had no monitoring solution and no alerting tool which would have allowed us to pinpoint where it hurts the most. Which it did badly, I can assure you, many times. Grafana and Prometheus – these services became our guides or – referring to the previous metaphor – our eyes.

It's all in the code

One of the main ingredients for well-working orchestrations is to have an entire infrastructure in the code (IaaC). We chose Terraform from the list of provisioning tools as it is the most developed with regard to our needs and widely supported by the open-source community. In that way, we were able to encapsulate the whole infrastructure in the code. This part of the whole process took a huge amount of time, but it was definitely worth it. I will be telling you about that in more detail a bit later.

Ansible – without this Configuration Management Tool, we would still be doing updates, changes and implementations on the servers manually. Similarly to Terraform, to choose the correct implement, we were guided by the following facts:

- It is highly developed

- There is a wide open-source community

- It is agentless

- There are a huge amount of modules we could use

As with Terraform, I will elaborate on this later.

Packer – this tool is a time-saver because it lets us create identical machine images with all of the required packages installed. Thanks to this, we could use them in Terraform right away, without installing everything from scratch.

Last but not least – Jenkins. Continuous Integration and Continuous Delivery, extremely valuable and important to the deployment orchestration. We based our CI/CD on Jenkins because we had experience with this solution before, it was suitable for our needs (plugins, modules, integrations) and we didn't want to waste time looking for something new.

Having a toolchain chosen, we had to make a big call – where to splice it all?

...the cloud is all we need almost

In the last quarter of 2018, AWS dominated cloud solutions, which is why we decided to choose this provider. Having the largest number of services and resources and good dynamics when it comes to changes and updates gave us the confidence that we would not have to reconsider our decision next year if AWS were to suddenly stop developing.

Using AWS tools (Infrastructure as a Service) such as Spot Instances, Auto Scaling Groups, Launch Templates, RDS, S3, and many others, we started the migration process, so ensuring high availability and self-healing mechanisms if one or more of our basic elements were to unexpectedly be taken by evil spirits crawling from the shadows disappear.

The four members of the fellowship – Packer, Terraform, Ansible and Docker

Packer is an open-source solution for creating machine images with different providers, for example, AWS AMI. Packer is lightweight and can be configured easily. What is more, Packer allows the use of configuration management tools such as Ansible to install packages and any required software on a particular machine image during its creation. Because of this, when the image is prepared, we don't have to worry about installing everything from scratch on the operating system.

In our case, all the servers are based on Ubuntu 18.04 LTS and this exact Linux distribution was used in the Packer configuration as well.

The Packer configuration includes two main blocks:

- packer_template.json

- Vars directory

The template file looks like this:

```
{
  "variables": {
    "home": "{{env `HOME`}}",
    "build_number": "{{env `BUILD_NUMBER`}}"
  },
  "builders": [
    {
      "type": "amazon-ebs",
      "name": "{{user `hostname`}}_{{user `os_name`}}",
      "ami_name": "{{user `hostname`}}_{{user `os_name`}}",
      "profile": "{{user `aws_profile`}}",
      "region": "{{user `region`}}",
      "source_ami_filter": {
        "filters": {
          "virtualization-type": "hvm",
          "name": "{{user `image_query`}}",
          "root-device-type": "ebs"
        },
        "owners": ["{{user `image_owner`}}"],
        "most_recent": true
      },
      "instance_type": "{{user `instance_type`}}",
      "iam_instance_profile": "{{user `iam_instance_profile`}}",
      "ssh_username": "{{user `ssh_username`}}",
      "ami_description": "{{user `git_branch`}}",
      "launch_block_device_mappings": [{
        "device_name": "/dev/sda1",
        "volume_size": "{{user `root_size`}}",
        "volume_type": "gp2",
        "delete_on_termination": true
      }],
      "ami_block_device_mappings": [
        {
          "device_name": "/dev/sda1",
          "volume_size": "{{user `root_size`}}",
          "volume_type": "gp2",
          "delete_on_termination": true
        }
      ],
      "tags": {
        "Name": "packer_{{user `hostname`}}_{{user `os_name`}}",
        "created_by": "packer",
```

```
      ],
      "tags": {
        "Name": "packer_{{user `hostname`}}_{{user `os_name`}}",
        "created_by": "packer",
        "environment": "{{user `environment`}}"
      },
      "run_tags": {
        "Name": "packer_{{user `hostname`}}_{{user `os_name`}}",
        "created_by": "packer",
        "environment": "{{user `environment`}}"
      },
      "force_deregister": "true",
      "force_delete_snapshot": "true",
      "spot_price": "auto",
      "spot_price_auto_product": "Linux/UNIX (Amazon VPC)",
      "security_group_id": "{{user `security_group_id`}}",
      "subnet_id": "{{user `subnet_id`}}",
      "ssh_keypair_name": "{{user `ssh_keypair_name`}}",
      "ssh_private_key_file": "{{ user `home` }}/.ssh/{{user      `ssh_keypair_name`}}",
      "associate_public_ip_address": true
    }
  ],
  "provisioners": [
    {
      "type": "shell",
      "inline": "{{user `install_ansible_script`}}"
    },
    {
      "type": "ansible-local",
      "playbook_dir": ".",
      "clean_staging_directory": "true",
      "playbook_file": "{{user `ansible_playbook`}}",
      "inventory_file": "{{user `ansible_inventory`}}",
    }
  ]
}
```

The basic values are mentioned in common.json in the Packer vars directory. So as not to send you to sleep, this file contains all the necessary parameters for Packer to work – plus some extra.

This file defines an AWS profile, the root size of the EBS volume, region, instance type, and a bit more:

```
1  {
2    "aws_profile": "packer-profile",
3    "root_size": "10",
4    "region": "eu-central-1",
5    "instance_type": "t3.medium",
6    "iam_instance_profile": "ci",
7    "ssh_keypair_name": "ssh-key",
8    "security_group_id": "sg-0459xxxx",
9    "subnet_id": "subnet-0797xxxxx",
10   "ansible_inventory": "packer_inventory.ini"
11 }
```

As we can see, besides the basic inputs (according to the Packer docs), there are also ones regarding Spot instances because each time Packer builds an image, it has to create an EC2 instance to configure the required components and execute the Ansible roles needed for any particular image. We used Spot instances because we wanted to cut the costs of using EC2 instances in general billings. When Packer finishes, the AWS AMI is created and the Spot Instance is terminated.

Another extra part is at the end of this template file – Ansible. As I mentioned above, Packer allows us to use a Configuration Management Tool and this is where Ansible comes into play.

For each host/server, we have defined particular Ansible roles to be applied to the image. The hosts that the roles are applied to are described in the inventory_file, which in this case is packer_inventory.ini, and for example look like this:

```
xwiki  ansible_host=127.0.0.1 ansible_python_interpreter=/usr/bin/python3
```

In Packer vars, we have two modes: build and ci. For the purposes of building machine images , the build mode is used. This mode references the Ansible file with details about the specific roles to be applied, for example xwiki host. Ansible_host is set to localhost because of the “ansible-local” type (mentioned in the code snippet above). Of course, a remote address is possible as well.

```
# ROLES FOR XWIKI
- hosts: xwiki
  any_errors_fatal: true
  roles:
    - configure_os
    - os-update
    - base-packages
    - docker
    - ssh
    - fail2ban
    - aws-cli
    - xwiki
  become: yes
```

The last part regarding Packer is a base image definition (ubuntu-18.04.json), and it goes as follows:So each time an image for Xwiki is built, Packer applies for the above roles on the AWS AMI. Thanks to this, when a new server with Xwiki is spawned (new EC2), it contains all of the necessary configurations, settings, and files.

```
{
  "os_name": "ubuntu18.04",
  "image_query": "ubuntu/images/*ubuntu-bionic-18.04-amd64-server-*",
  "image_owner": "099720109477",
  "ssh_username": "ubuntu",
   "install_ansible_script": "until [ -f /var/lib/cloud/instance/boot-finished ]; do sleep 1; done,sudo
   DEBIAN_FRONTEND=noninteractive apt-get update,sudo DEBIAN_FRONTEND=noninteractive apt-get -y
   install python3 python3-setuptools python3-dev python3-pip; sudo ln -s /usr/bin/python3 /usr/bin/python; sudo
   ln -s /usr/bin/pip3 /usr/bin/pip; sudo pip install ansible==2.8.3"
}
```

Install_ansible_script – this parameter defines instructions to be executed on this image before any others. We are using python3 as a default executor, which has to be installed and set as default by creating symlinks to the proper binary. Also, as we wanted to take advantage of Ansible, this had to be added because it was not a pre-installed software on OS. We base AWS AMIs on Ubuntu 18.04 and during the building process, Packer uses this particular image version mentioned in the image_query. This value can be found in the AWS EC2 section once a new instance has been spawned manually.

Finally, to build an AWS AMI for Xwiki, we have to execute (with respect to the proper location of these files):

```
xwiki.json:
{
  "hostname": "xwiki",
  "environment": "dev"
}
```

```
build.json:
{
  "ansible_playbook": "internal_hosts.yml",
  "ci_mode": "True"
}
```

..please do not faint now, do not fly away to /dev/null with your brain.

..yet more amazing things to be discovered.

..overcome your doubts and be rewarded with great knowledge.

Terraform is a tool for creating, changing, and managing infrastructure defined in configuration files. It supports and works with well-known Cloud providers such as AWS, Azure or Google Cloud Platform.

After every action in configs, Terraform can show what will change in the infrastructure and is then able to execute a plan to reach an appropriate state.

When a small modification is made, Terraform will do only that, without rebuilding everything from scratch. This is why Terraform is based on state files.

Terraform has well-described documentation with examples and code snippets, which can easily be adjusted to individual needs.

As we decided to choose AWS as a major Cloud provider, we started to migrate the existing resources (created manually) such as IAM, S3 Buckets, VPC and Route53, etc. to the Terraform configuration files using the terraform import command.

But this was just the tip of the iceberg. With a lot of resources, like all the servers spread across hosting providers, Auto Scaling Groups, Launch Templates, EBS, ALB and more, we had to develop from 0 state to have our infrastructure as a code.

All of this gave us the mobility to use the Terraform code from wherever we wanted, using laptops or Jenkins. Even if one or more components were to be taken by evil forces, we would be able to recreate them easily and flawlessly.

This process took over six months and is actually still ongoing but now we can just add new bricks to this wall, without worrying that it is about to fall on our heads.

..done talking, showtime!

This article is for an audience that already has some experience with the aforementioned tools, so I will not present basic stuff such as AWS CLI configuration or how to install Terraform.

When I previously described Packer, I based all of the examples on Xwiki. So let's stick to that in this part as well.

In this paragraph, my plan is to show you how we built an infrastructure using the IaaC concept and how it is a part of the entire orchestration process.

During brainstorms within the DevOps team, one of the conclusions we came to was to ensure High Availability and Self-healing mechanisms for all Scalac's applications.

AWS comes with such solutions natively in the EC2 section and they are called:

Launch Template

Auto Scaling Group

The configuration for the first regarding Xwiki is as follows:

```
data "aws_ami" "ubuntu" {
  most_recent = true

  filter {
    name   = "name"
    values = ["xwiki_ubuntu18.04"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }

  filter {
    name   = "root-device-type"
    values = ["ebs"]
  }

  owners = ["self"]
}

resource "aws_launch_template" "xwiki-lt" {
  name_prefix = "lt-xwiki"
  image_id    = data.aws_ami.ubuntu.id

  block_device_mappings {
    device_name = "/dev/sda1"

    ebs {
      volume_size = 50
      volume_type = "gp2"
    }
  }

  instance_type = "t3.small"
  key_name      = data.terraform_remote_state.global_variables.outputs.keypair-scalac
  user_data     = base64encode(data.template_file.user_data.rendered)

  network_interfaces {
    device_index                = 0
    associate_public_ip_address = true
    security_groups             = [aws_security_group.sg-xwiki.id]
  }

  iam_instance_profile {
    name = aws_iam_instance_profile.xwiki-instance-profile.name
  }

  tag_specifications {
    resource_type = "volume"

    tags = {
      Name        = "xwiki-root-volume"
      environment = "prod"
    }
  }
}
```

xwiki_ubuntu18.04 defines which AWS AMI will be used. This was created by Packer as pictured above in the Packer section. This config creates a Launch Template and all the resources related to it.

Later on, the AWS EBS is configured, the Instance type on which Xwiki will be installed, and the IAM instance profile to be attached to the EC2 instance, security groups and tags.

User_data describes pre-configured actions to be taken, such as: checking if the EBS is mounted, associating always the same Elastic IP, creating the proper FS on partitions.

The Launch Template is strictly connected to the Auto Scaling Group, so the config goes as follows:

```
resource "aws_autoscaling_group" "xwiki" {
  name             = "xwiki"
  min_size         = "1"
  max_size         = "1"
  desired_capacity = "1"
  force_delete     = false

  launch_template {
    id      = aws_launch_template.xwiki-lt.id
    version = "$Latest"
  }
  vpc_zone_identifier = [
    data.terraform_remote_state.internal-prod-vpc.outputs.network_public_id[0],
  ]
tag {
    key                 = "Name"
    value               = "xwiki"
    propagate_at_launch = true
  }
tag {
    key                 = "environment"
    value               = "prod"
    propagate_at_launch = true
  }
tag {
    key                 = "project"
    value               = "xwiki"
    propagate_at_launch = true
  }
}
```

ASG is also used here as a Self-healing mechanism, which is also a big keyword for the orchestration process. If it gets terminated somehow, it will be recreated one more time based on the above settings. ASG ensures how many instances will be spawned, which Launch Template will be used and in our example it also adds some tags describing resources.

The previously mentioned resources regarding Xwiki are not the only ones we have in the Terraform config files. We also have Route53, Security Groups and EIP defined as IaaC, but it would be boring to show you all of these as they are the same as in the documentation, only just adjusted to our internal needs.

Orchestration means possessing an Infrastructure which is safe and sound and also ready to be resurrected from the dead when required without any human interaction, or at least with less effort.

Ansible is a configuration management and application-deployment solution designed by Red Hat. It works as anagentless model and requires only an SSH established connection and Python as an interpreter on the target machine.

Similarly to Terraform, it also has a declarative language that is used to describe roles and tasks to be executed.

Ansible can be run on one target or more simultaneously.

In our case, the structure is as follows:

host_vars – variables used in roles and tasks for specific hosts.

group_vars – variables used for all hosts enclosed in the groups.

internal_hosts.yml – Ansible playbook.

inventory.ini – file containing all targets on which we execute Ansible.

So if we would like to launch this tool, we hit the terminals with the following command:

```
ansible-playbook -i inventory.ini internal_hosts.yml
```

If a certain object needs some extra variables included in the role or task, they are defined in the host_vars directory. On the particular target, it executes the roles and tasks described in internal_hosts.yml using the SSH connection.

Based on Xwiki in my previous examples, I will do the same here.

For example, host_vars/xwiki looks like this:

```
hostname: xwikix
wiki_test_pass: !vault |
        $ANSIBLE_VAULT;1.1;AES256
633033383465346139386238643638366633613439356332333334623338643536616631363566653939353765343
613335333964383353431663132376130346538623166353366636236363132643864
certbot_domains:
  - wiki.scalac.io
nginx_generic_domains:
  - { domain_name: 'wiki.scalac.io', dest: 'http://localhost:8080', cert: 'wildcard', action: 'proxy_pass' }
```

This is a good time to explain the !vault existence in the above code as this is what we use very often in files with variables. The hostname is matched with the SSH config; if the entry is here, Ansible starts the connection, if not, it will throw an error – hostname does not match – and will stop the execution.

To protect important and crucial data such as passwords, tokens, access and secret keys, Ansible provides a tool called Ansible-vault. It is integrated within so there is no need to install it separately.

All we have to do is just to put the important data in plaintext in some temporary file and then run:

ansible-vault encrypt this_temp_file and fill the Vault password on the terminal prompt. Black Magic happens and the data is encrypted.

To decrypt:

ansible-vault decrypt this_temp_file and fill in the Vault password which was used to encrypt.

We use Ansible roles for Nginx and Certbot, directives which are also included in the host_vars/xwiki file.

It has already been mentioned in the Packer section what internal_hosts.yml for Xwiki looks like, so there is no need to repeat it once again.

The information in the form of tasks to be executed on Xwiki hosts is defined in the Xwiki role. For this service actually there is only one:

```yaml
---
- name: Run xwiki container
  docker_container:
    name: xwiki
    image: xwiki:11.3-postgres-tomcat
    state: started
    pull: true
    restart_policy: always
    published_ports:
      - 8080:8080
    volumes:
      - /data/xwiki/xwiki:/usr/local/xwiki
    env:
      DB_USER: xwiki-test
      DB_PASSWORD: "{{ xwiki_test_pass }}"
      DB_DATABASE: xwiki-test
      DB_HOST: testenv-postgres.fnijfgtbdlyn.eu-central-1.rds.amazonaws.com
```

This runs a built-in docker container module with all the needed parameters, builds and finally launches Xwiki service aka wiki.scalac.io in a container with the mapping ports on the designated target. Database credentials (encrypted by ansible-vault) are injected as environment variables.

In addition to the above role, we also have different ones:

Role name	Short description
configure_os	for configuring OS with setting up hostnames in /etc/hosts and disabling system services
os-update	for keeping OS always up to date
base-packages	for installing commonly used packages like gzip, htop, git, etc.
docker	for installing docker on every target
jenkins	for installing and configuring Jenkins and its components
ssh	for applying and conducting ssh keys and configs for every host
fail2ban	for installing and configuring fail2ban – server security tool
chrony	for installing and configuring one NTP server on every host
exporters	for applying Prometheus exporters on every host
aws-cli	for installing and configuring AWS CLI
backups	for applying backup scripts and cron configs on every server
certbot	for installing, setting up and generating certs using Certbot
nginx	for installing and configuring Nginx
logs-filebeat	for installing Filebeat on every host for ELK

In this entire AWS orchestration process, Ansible plays a big role. Having the infrastructure in a code (thanks to Terraform) does not solve all the problems though. Without a configuration management tool we would still be doing OS-related stuff manually and this would consequently lead to repetitive mistakes.

..hey, are you asleep already? No, no, no... You can't! Because this is where the real story begins!

Docker was invented and developed to facilitate the entire deployment process. The software maker can easily package up an application with all its components such as libraries into one container and ship it all out to the desired target.

Containers are separated from each other and thanks to this, you are able to run multiple applications on one server without having any problems with dependencies.

During the discovery phase on old infrastructure, we decided to package every possible application into the Docker containers. Having done that, we were able to build it, ship it and deploy it onto AWS EC2 instances using Docker Hub as a harbor for container images.

To prepare an application package as a Docker image, we had to create Dockerfiles which are required in the building phase.

This process is strictly related to Jenkins, as he is the captain of this orchestration ship, so it will be described in a bit more detail in the Jenkins section, where we are heading right now.

Jenkins – to gather them all

What do automation and orchestration processes look like without proper Continuous Integration and Continuous Delivery? There is no such thing.

CI/CD is the first fiddle in the whole deployment methodology.

In our story, we used Jenkins for this purpose.

Jenkins is an open-source automation server. All non-human actions during the deployment activities can be managed by it. This CI/CD solution contains an enormous amount of packages and plugins to be used in job configurations. It can also very easily integrate with third-party applications to ensure a better and more efficient delivery flow.

Jenkins reminds me of a little man with big and bare feet – Frodo Baggins when he was gathering the Fellowship of the Ring to make his colossal commitment.

Using it in our AWS orchestration, we were able to join all the pieces of the jigsaw together in one platform.

This whole article is based on an Xwiki example. Actually, in this section, that could be a little inaccurate because building and deploying Xwiki has already been done by Ansible as mentioned a few paragraphs ago. But what goes around, comes around.

Jenkins, in this case, is responsible for launching two jobs == tasks.

- *devops-packer-xwiki*

- *devops-refresh-servers*

First of all, it downloads our repository from Github, runs the script for building a Packer image and later on executes the Terraform apply command on the AWS Launch Template to switch the ID's of the xwiki_ubuntu18.04 image in AWS AMI.

```
#!/bin/bash
set -x
set -euo pipefail

docker login -u $build_user -p $build_passwd

# build ami
./build_images.sh xwiki

# update launch template
cd terraform
./apply scalac/internal-prod/xwiki
```

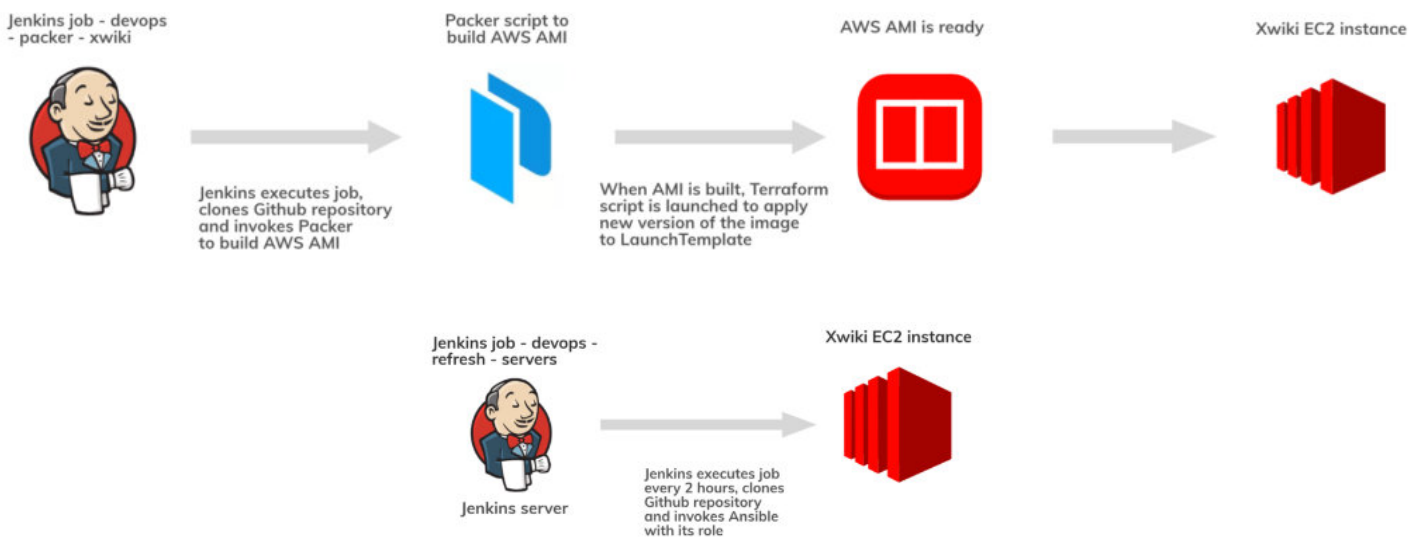
This job runs every 2 hours every day, but if the service is down before another execution, Prometheus alerts are delivered to the Slack channel about downtimes and we have the power to launch this job manually from the Jenkins UI. The second one is in play because it invokes the Ansible playbook with the roles defined in the internal_hosts.yml file matching the particular host from the inventory. In this example, it is the tasks for the Xwiki target (previously mentioned in the Ansible section of this article).

Failure example:

If on the server where Xwiki is hosted, the docker daemon or the docker container with the service stops working, devops-refresh-servers will relaunch the Xwiki Ansible role and restore the container to an up-and-running state.

In addition, this particular job also protects us from arranging any manual interactions on the servers. Even if such a thing happens, it will restore the defaults written down in the host/group vars.

The picture below shows the flow for these two Jenkins jobs:



The above examples were based on bash scripts we have created for our internal needs. Almost every project has Jenkins jobs because in every one there is a build and deployment phase. As we were trying to avoid manual steps as much as possible when starting the orchestration , we used our CICD to build Docker containers for applications as well.

Build phase using Jenkins:

```
docker login -u $scalac_user -p $scalac_pass

export PROJECT_VERSION=`grep -E "^version" build.sbt | sed -e "s/.*= \"\(.*\)\"/\1/"`

# clean up previous build artifacts
sudo rm -rf hire-help-$PROJECT_VERSION target

# build the code
docker run --rm -i \
    --name $JOB_NAME \
    -v $WORKSPACE:/src \
    -v $WORKSPACE/.ivy2:/root/.ivy2 \
    -e PROJECT_ENV="dev" \
    -e PROJECT_DB_PASSWORD \
    -e PROJECT_DB_NAME \
    -e PROJECT_DB_USERNAME \
    --cpus="1" \
    hseeberger/scala-sbt:8u181_2.12.8_1.2.8 \
    /bin/bash -c 'cd /src && ./build.sh'

# unzip the package
docker run --rm -i \
    --name $JOB_NAME \
    -v $WORKSPACE:/src \
    -e PROJECT_VERSION \
    -e PROJECT_DB_PASSWORD \
    -e PROJECT_DB_NAME \
    -e PROJECT_DB_USERNAME \
    -w /src \
    busybox \
    unzip /src/target/universal/project-$project_VERSION

# build and push the container
docker build -t our-dockerhub/scalac-project-backend:dev .
docker push our-dockerhub/scalac-project-backend:dev
```

The deployment phase using Jenkins:Building the code and unzipping happens inside the Docker containers so not to pollute the server space where Jenkins's jobs have been executed.

```
APP_NAME=project-backend-devel

for i in `docker ps -a | grep -E "\s$APP_NAME\s*$" | awk '{ print $1 }'`; do
    docker kill $i || true
    docker rm $i
done

docker pull scalac_docker_hub/scalac-project-backend:dev

docker run -d \
    -e PROJECT_S3_ACCESS_KEY \
    -e PROJECT_S3_SECRET_KEY \
    -e PROJECT_ENV="dev" \
    -e PROJECT_DB_PASSWORD\
    -e PROJECT_DB_NAME \
    -e PROJECT_DB_USERNAME \
    -p 10000:10000 \
    --name $APP_NAME \
    --restart always \
    scalac_docker_hub/scalac-project-backend:dev \
    /src/start.sh
```

Without Jenkins, we would still have one foot in Dark Times, despite having IaaC, IaaS and other puzzles that fit in with modern orchestration.As I have already mentioned, we were trying to build and deploy every application of ours in the same way to standardize the entire process. I think we have managed to achieve 90% of that so far. But some work is still yet to be done.

With GitFlow we all flow

Having CI / CD tools combined is one thing, but using them wisely is another – not only from the infrastructure point of view but also from the perspective of the software .

In a situation where several developers work on an application code, there's no room for using only one branch to direct all of the commits, without any order.

Each stage of the project should have its own specific tree structure in the repository. GitFlow is a tool that in a well-defined way allows you to enter the life cycle of the entire project. Regarding this subject, we have a dedicated article where the entire flow we use is presented.

To wrap up

At the beginning of this article, I declared I would try to explain all of the concepts in more detail. I hope I have managed to achieve that, despite the scope. Of course, our solution cannot ensure that all the problems will just disappear. This article gives an insight into the problems we were facing and the methods we picked to figure them out.

Now you can see what orchestration and automation looks like, with regard to infrastructure and software.

If you still have any doubts or questions, do not hesitate to leave them in the comments below.

Author profile

Błażej Obiała

I am a DevOps Engineer passionate about automating processes. In the course of my career, I have gained three AWS certificates, and thanks to them, I can support customers by answering their questions regarding AWS architecture in their projects. Recently I also became a Certified Kubernetes Administrator.

Privately I am keen on reading historical and philosophical literature while listening to good vinyl records. I used to play bass guitar in a band called Mad Teacher during my studies, so I've been on both sides of the music world.