You totally can test stuff, fix bugs and check out how to implement new features on production!

If you're a masochist, that is.

Usually a production system is a no-no for anything development related. Even if bugs happen on production, the best practice is to reproduce those bug on systems *other* than production.

These systems are usually called integration, development, acceptance or QA with various understandings from project to project.

But how can you reproduce a bug on a system that is different from production?
How can you bring the database, data and application files from production to a different system.

At my employer, **kps** we started doing that using Bash scripts. That worked quite well until we had more demands concerning speed, flexibility, complexity and – *let's be honest* – readability.

To conquer those we moved away from Bash scripts and to *Ansible playbooks*.

In case you lived among a flock of sheeps for the last ten years, **Ansible** is a software belonging to the „Configuration Management System" category. It is used to describe how a bunch of servers should be installed and configured. It's written in Python and describes itself as „agentless" (although it actually needs an SSH access to your system and an installed Python environment).

Ansible works using so called *playbooks* consisting of multiple *plays* against target systems. Plays have a bunch of *tasks* that are run on these systems.
There's a vast list of *modules* available providing tasks for every flavor. There are file system tasks, database tasks (for all sorts of databases) and so forth. Check out **their documentation** for details.

The premise

But I'm not talking about my professional life today. Instead, let me tell you that I'm active in an open air theater association called the **„Waldbühne Heessen“** (sorry, german only website). I started as an actor there, but you know the common way for people working in the IT business. So I'm now managing their IT stuff together with two fellow actors.

One of the systems I'm responsible for is a site the association uses for organizing auditions, shows and the other stuff that happens throughout the year. Additionally, it contains a bulletin board for discussing things.

The site is based on the **Elgg framework**, a platform for social networking sites. Elgg is a modern **php** framework, utilizes caching, a data directory and a database.

Now we needed to migrate from Elgg 2.3 to Elgg 3.2, which was a major jump with several breaking changes and breaking a lot of plugins. As a good SRE I knew, I had to build a proper integration system to prepare that move and to use it as a showcase for my fellow members.

Building a clone playbook

It's actually quite easy to build a clone playbook, because it simply mimics what you would do manually. So to build my playbook, I checked out what I had to do to clone my production data to an integration system.

From my experience with the clones from my employer I knew the basic steps usually required:

- Dump the database
- Augment the database dump file to match the integration system (This usually includes changing URLs or paths)
- Restore the database onto the target database host
- Copy the application files
- Copy the data files
- Augment the application configuration file to match the integration system (This obviously includes the database settings, paths and some minor stuff)

- Additional works (i.e. Setting an announcement banner to warn the user, that they're on the integration system)

As an additional topping, I wanted to include all stuff required to update the system to the new version as well. But I wanted to do that dynamically, so I could just switch it off when I had migrated the production system.

Transforming the manual steps into a clone playbook was mostly just checking for fitting modules in the Ansible docs and figuring out how they needed to be configured (which was taking the most time frankly).

For the database tasks, the mysql_db task was used. It supports dumping and restoring databases and also running SQL scripts. The file tasks mostly compress folders using the archive task and copies them around. Augmenting the database dump or application settings, I used the replace and lineinfile tasks.

Building the inventory

The playbook itself is rather static. It's basically just a list of tasks in a Yaml file, that Ansible runs from top to bottom.

Parametrizing the playbook is done using inventory files. They hold the connections to the target hosts, but also parameters like database username and paths.

The inventory files can be written in an ini and yaml flavor. After I initially started using the ini flavor because we used the same at my employer's I finally switched to yaml format because I needed to configure lists and object values for parameters.

That way I could add an external task file with the update tasks into the inventory, which is included into the main playbook using *include_tasks* Ansible task and could then clone the integration server and migrate in one step.

For sensitive data like passwords, Ansible brings a vault solution, that encrypts the values in the files using AES (or other cryptos you configure). You encrypt the values using a passphrase and simply use the same passphrase when running the playbook.

Bringing it all together

Of course, this all wasn't a one shot. I had to use several tests to finally finalize the playbook and have it working correctly. Depending of the size of the system, this can take some time. So prepare yourself with enough coffee for the task.

As a reference, if you're interested, I open sourced the whole stuff on **the association's github organization**. If you have questions, don't hesitate to ask.

Bonus! Building a local development environment

I don't know about you but I don't like to do stuff twice. To develop the site and plugins we're using a local containerized development environment based on **Docker** with mounted database and application paths. So why not use the clone playbook to clone production for the local environment?

Ansible supports us there quite well by accepting docker connections to target systems using the local docker daemon.

So I wrapped up a **compose file** that starts a database and an application server (we created our own **Elgg-compatible application server image**). This servers are then filled using Ansible and I can simply *tar* the mounted volumes, bring them to my server and start my development environment, also **using a compose file**.

Voilà.

(Cover Image: **Night. Clone by AdNorrel**)

*Originally published to **dev.to***