# Declarative Configuration Management

stefan.steinert@t-systems.com

v1.0.4, November 2018

## Contents

## 1 Preamble

The idea of declarative state management was not invented by me and has been around for some time. I propose this as an idea because I think it not well understood on average but can bring a lot of value to the table if widely promoted and adopted.

## 2 State on a computer system

The benefits of a declarative approach to configuration management might not be immediately obvious and mainly requires an abstract understanding of the term *state* in the context of managing computer systems.

State-related topics are an elementary challenge in handling computer systems and are commonly found in their architectures. Some systems (e.g. databases) are termed *stateful* while others (e.g. application servers) are referred to as *stateless*. However this is not an either/or classification. Rather it is a gradual difference and mainly refers to the value of the state. This value is an expression of the effort required to reproduce it. I.e. if state is very hard (or even impossible) to reproduce then the system can be described as *(very) stateful*. Stateless systems are generally easier to handle because several - often complicated - topics do not apply to them. State always has to be taken care of, leading to additional requirements to an architecture like consistency, reentrancy or reproducibility.

State is commonly found in most major operating systems [1]. It is useful and necessary to define and customize the computers behaviour. On an abstract level a significant part of the computer can be thought of as a *collection of states*. This state is not easily reproducible and thus requires tight management to ensure consistent behaviour.

At the same time it is important to recognize that everything in software is constantly changing. Anything can be improved virtually forever. The question is rather how much value a modification adds versus how much it costs (i.e. how much effort is required to perform the modification). It is hence important that

[1] Other approaches like `https://www.nixos.org` exist. It aims to avoid state as much as possible on a low level. Although this is a promising idea it is not very mature yet and has its own set of challenges.

applying change is an inexpensive operation. Otherwise a system cannot evolve to its optimum.

Important requirements towards the state of a computer system are:

- it should be robust and predictable
- but it should also be easily modifiable with minimal effort so that it can evolve over time
- it should be consistent throughout a group of related systems no matter the point in time a system is added to the group
- it should be stageable (i.e. "change non-prod first, then change prod in the exact same way") with minimal potential of error
- it should be testable (i.e. "What will be the effect if I apply this modification?")

State is also a common organizational topic in software development where essentially a group of people works together on a set of files to build an application. The more people are involved in this process the more complex the collaboration becomes, largely due to conflicting changes to the files. For decades this has been supported through the use of version control systems. They guarantee a consistent state where conflicting changes can be precisely tracked and resolved. It is a good example of how state problems can be alleviated by using managed state instead of unmanaged state. It seems to be a good idea to apply similar mechanisms to configuration management.

Not all events in the lifetime of a system are beneficially described as state. For example, repeating deployments of new versions of customer applications are not a good candidate to be described in this way. Instead these are recurring modifications with non-identical outcome (i.e. with varying state). Such activities are better described in ways which do not emphasize on maintaining state but instead on describing the (constant) *transition*.

## 3 Approaches to managing configuration

Apart from manual activities (which are inherently prone to human error) there are two fundamentally different approaches to managing (i.e. creating, altering and maintaining) the configuration state of a computer system.

The first - more widespread and traditional - way to think about state-change is to write down the steps required for the transition. "Create this directory, create a file with that content, ...". This approach can be termed "imperative" as it is a list of commands required to modify the state from A to B. Typical examples for such an implementation is the use of scripts written in languages like Bash or Python. This technique often feels more natural because it resembles the way in which we are used to interact with the world in general. We execute an action. We receive a result.

The second way to think about state is to create a formal model describing the *should-be* state that the machine is *supposed* to be in. And let the machine figure out the required steps to reach that state at run-time by itself. Instead of describing the transition it describes the aim. This approach is commonly called "declarative" as it is based on a formal description of the desired state. The current state of the computer is then (typically in intervals) compared against and adjusted to that state. Examples of tools which implement this strategy are Puppet or Salt.

## 4 Advantages of declarative state

While the differences between these two approaches might seem minor at first glance there are important aspects which make the declarative approach superior in several ways:

- A declarative model *guarantees* a certain state on the system with no guesswork required. In an imperative approach state B cannot be guaranteed because state A is not guaranteed. A lot of effort today goes into trying to emulate this guarantee by applying restrictive access rules and relying on traditional change-management methodologies. However that approach continues to fail every single day.
- The declarative state can be easily and continuously adopted and improved by changing the declarative model. The model will stay valid and guaranteed for all managed systems no matter their age. In the lifetime of the model it becomes increasingly simple and decreasingly frequent to change it. In

contrast the imperative approach becomes increasingly hard to maintain over time due to the increasing number of possible states and lack of a common, homogeneous, descriptive model.

- The overall system state can be composed from modules. By cleverly cutting the declarative model into separate modules, the overall state of the system can be compiled from individual components. These modules can be independently created and maintained by different subject matter experts or different teams. Other technicians can use the provided modules without requiring them to have much detail knowledge of that subject. They can act as consumers. In an imperative approach such consumable modules are almost impossible to build because it usually requires a consumer to either understand the subject matter experts code or dig through (often poorly maintained) documentation. Most typically it even requires both which then quickly leads to re-implementations on the consumers end.
- A declarative approach encourages collaboration among people and teams because conflicts in state do necessarily pop up. If team X tries to manage the same file as team Y then such overlaps (which are the direct result of a lack of communication) result in computational run-time errors complaining about duplicate declaration. This in turn forces the parties to communicate and find a common solution, be it technical or organizational. In an imperative approach such conflicts typically go unnoticed until someone detects that his components are not working as expected. While finally this probably yields the same end result it is far more difficult to detect and creates an atmosphere of distrust and the desire for restrictiveness.
- No additional documentation regarding the system state is required. The declarative model already is a precise and complete description of the relevant system state. Implementation and documentation no longer happen to be two (technically) unrelated entities. They become a single entity.
- The declarative model enforces a ”document first” approach because the state of the system can only be changed by first changing the model (i.e. by changing the documentation).
- A declarative model can serve as a broadly useful, semi-technical documentation. It is expressive and precise to technicians as well as to computers.

But it can be of value to non-technicians, too. By applying transformations the model can always be converted into less detailed and more presentable / more descriptive forms anytime if desired. This does not hold true for an imperative approach where the code which performs the state transformation cannot be converted into anything meaningful in an automated and generalizable fashion.

## 5 Lessons learned

We are applying this idea internally to lots of our systems since long and achieved legit time savings and synergies with it. By managing the declarative model within a commonly accessible version control system (a GitLab instance in our case) all of this evolving and modularized state is precisely documented and historicized from the very beginning until the present moment and down to the smallest detail. Among other benefits this is a great source to learn from and to avoid making architectural mistakes twice.

Staging of configuration changes has become robust and intuitive as it is performed by simply merging branches in the version control system. Changes are perfectly reproduced throughout stages. To extend this further we are about to attach a CI-controlled Docker build-and-test pipeline which enables testability without even touching any real system.

To us this approach has proven to exceed the informative value of other - typically manual - ways to document the process (like hand-written documents and generic change processes) by far.

I think the approach would specifically excel if applied to a broader base of people, teams and systems. Internally (i.e. for T-Systems own purposes) as well as externally (i.e. for customers).

## 6 Proposal

This brings it to the point of my proposal. I think the above advantages make this a perfect choice for a real-world, internal- as well as customer-facing service.

Offering computer instances managed based on declarative configuration management will erect a transparent, traceable and reproducible way to define a good part of what a system should provide. The state (and thereby much of the functionality) of a system could be composed of readily available T-Systems modules in combination with purpose-built customer specific modules created and maintained collaboratively by T-Systems- and customer-engineers, all perfectly documented using version control systems.

I'd say many customers (read: the customers technical staff) will see this kind of transparent and reproducible stewardship as a value-add. Especially if we consider that the problem of state management keeps increasing significantly due to the ever increasing number of deployed systems.