OrangeBook 2.0

Autistici/Inventati

Contents

1 Introduction **1**

1.1 Historical context 1

1.1.1 *The R plan* 1

1.1.2 The technical state of A/I at the beginning of 2017 3

2 Organizational principles **5**

2.1 Human and relational aspects 5

2.2 Technical aspects 6

3 Redefining infrastructure **9**

3.1 The technological substrate 9

3.2 Choosing a "level of service" 9

3.3 Towards an ideal service model 10

4 Implementation **13**

4.1 Distributed operating system 13

4.2 Building a bridge between *Configuration Management* and *Container Orchestration* 14

4.2.1 *Float* 15

4.3 On the opportunity of writing things versus using existing code 17

4.4 Source Control and Continuous Integration 18

4.5 The structure of our software 19

4.5.1 Common rules for artifact generation 20

4.5.2 Dependencies management 21

4.6 Secrets, meta-configuration and *environments* 22

4.7 A pragmatic approach to containers 23

4.7.1 Image relationships 24

4.7.2 Configuration 25

4.7.3 Users and permissions 26

4.7.4 Ports 27

4.8 Internal communication and RPC 28

5 Data and service structure **31**
5.1 Users database . . . 31
5.2 Data and partitioned services . . . 32
5.3 LDAP implementation . . . 33

6 Authentication and identity **35**
6.1 Authentication server . . . 36
6.2 Single Sign-On . . . 37
6.3 Cryptography . . . 39
6.4 Authentication mechanisms . . . 39
6.5 Workflow in detail . . . 41
6.5.1 SSO . . . 41
6.5.2 Non-HTTP services . . . 42
6.5.3 Third party services authentication . . . 43
6.6 A user database API (*accountserver*) . . . 44

7 System *observability* **47**
7.1 Monitoring . . . 47
7.1.1 Alerts . . . 47
7.1.2 Blackbox monitoring . . . 49
7.2 Logging . . . 50
7.2.1 Extracting metrics from logs . . . 51
7.2.2 Logs as structured events . . . 51

8 An example in detail **53**
8.1 The A/I static website . . . 53
8.1.1 The application itself . . . 53
8.1.2 Service configuration . . . 54
8.2 Change lifecycle . . . 56
8.2.1 Testing . . . 56
8.2.2 Conclusion . . . 58

9 Automation **59**
9.1 State machines . . . 59
9.2 Reconciliation . . . 60
9.3 Configuration generators . . . 61
9.4 Account and data operators . . . 62
9.5 Inverse reconciliation . . . 63

10 Only time will tell **65**

11 Outro **67**

1 Introduction

How does the infrastructure of autistici.org work? And more importantly, why is it designed to work like this?

With this document we intend to uncover and describe the design choices we made since 2019 that led to the architectural implementation of the Autistici/Inventati infrastructure we use today.

This document is not only a snapshot of the current inner workings, but also an attempt at storytelling on how out infrastructure ended up being as it is now. We believe that our work might be a positive example of a complex transition from a legacy environment with a lot of accumulated technical debt to a much more modern architecture, finally suited to adequately fit the requirements set by the problem it was originally designed to solve.

In other words, we intend to evaluate *how* we originally implemented our infrastructure, *what* solutions and code we have been carrying around for years, *how* we dealt with the "patchwork" approach typical of light maintenance and workaround-related processes. These are the most relevant aspects that contributed to the cognitive complexity and the consequent and inherent lack of robustness of the A/I infrastructure itself. And furthermore they represent an even bigger challenge when it comes to finding and training new people desiring to be involved in A/I activity, with the objective of empowering them and to allow them to actually contribute to the project.

We are trying to create a sustainable way for volunteers to effectively participate in the maintenance and development of our infrastructure by introducing some important guidelines and best practices from the software engineering industry, without turning everyone into a professional.

In this text we use Autistici/Inventati or A/I in reference to the collective, and autistici.org as domain in the various technical references, even if the infrastructure is designed to serve several domains at once: inventati.org, anche.no, noblogs.org, etc.

1.1 Historical context

1.1.1 *The R plan*

The A/I project has evolved in a relatively orthodox way considering similar anticapitalist and self-managed experiences, quickly moving, as demand grew, from a single-server and experimental approach to a multi-server,

geo-politically distributed and wider-audience capable setup, with the progressive introduction of different *configuration management* mechanisms. The service architecture has also evolved together with our improved understanding of strategical challenges posed to our projects: for instance the multi-layer structure became necessary in order to isolate the publicly exposed surface of the infrastructure, subject to legal discovery processes, from the end-user data. This implementation choice is not something we mean to discontinue, as well as the policy of distributing our servers in many different parts of the world, so as to minimize exposure to single country legislation and possible repression threats. At the same time we mean to keep considering each colocated servers as inherently insecure (we are in no position to physically certify their security) and consequently design our infrastructure as having no "irreplaceable" node and as many fallback nodes as possible.

At a practical level, we planned on acquiring access to server colocations in different parts of the world in order to avoid having one specific commercial provider as "single point of failure". This has proved to be a key strategy and has determined the need for additional processes: first of all, the need for an organizational structure capable of addressing the financial challenges connected to recurring expenses, keeping track of contract agreements and so on. Even when considering our relatively small scale, this is already far beyond the amateur level: it is simply not acceptable, for example, to lose a month of end-users' emails because an invoice has not been paid. At the same time, geographically distributing servers in different parts of the globe determines different techical requirements when compared with the technology used in single-server (or datacenter-local) setups, especially considering the difference of magnitude of the network latencies involved.

Specifically, we decided to **not** implement a global coordination layer, splitting services between *stateless* components, that can be replicated in order to obtain redundancy and high availability, and *stateful* ones where we instead adopt a partitioning strategy in order to scale horizontally and limit the consequences of an outage. These two are combined with very simple, mainly *client-side*, load balancing algorithms (round-robin DNS).

At a technical level, the technological evolution of the infrastructure over time has led to a two-layer solution: a public-facing frontend, or *reverse proxy*, receiving end-users traffic, and an internally routed destination, which owns the user data. The communication between these two layers is routed through a VPN. The user data is spread over several servers (every server only contains a different data "slice") in order to scale beyond the capacity of a single machine, and so that any possible problem can only affect a smaller portion of users. In this type of architecture, the public-facing machines do not contain any important data, and are easily replaceable.

As of the year 2022 this infrastructure still effectively satifies the original requirements we had set for our systems:

- Protecting user data from legal abuse (i.e. seizing), forcing legal requests to go through official procedures;

- an acceptable level of robustness, since a single server failure will be unnoticeable (in the case of a reverse proxy), or at least limit it to a subset of end users;
- the ability to easily scale in case of increased demand, by simply adding additional servers (*horizontal* scalability).

The original Orangebook[1] contains further details about the original implementation and the specific technical choices we made at the time. Architectures of this type are much more common nowadays and don't require too much of an introduction.

The requirements we just described (dating from 2005) are still largely valid, as well as the basic service structure of our architecture. Recently though, we did introduce some drastic changes in the way we manage our infrastructure. Starting in 2017 and up to its first complete deployment in 2019 we redesigned and refactored the whole technological implementation of the A/I infrastructure, condensating several years of experience in the effort of finding a better solution to our present day's challenges.

Let's see why and how.

1.1.2 The technical state of A/I at the beginning of 2017

As it happens to most projects with a long lifespan and contributions from several people, A/I's technical infrastructure ended up accumulating over time several layers of historical "junk": often the result of substandard and *ad-hoc* implementations. If the original high-level design from 2005 - the so-called *R Plan* which introduced anonymous, de-localized and partitioned services - served us well, its technical implementation started to present the typical symptoms of *technical debt*[2]. Over time, these symptoms can cast a shadow over the overall quality of the system. There are many contributing factors to the development of these symptoms:

- as a consequence of continuous improvement over time an old infrastructure design becomes less effective;
- the presence and tolerance of "known bugs" inherited by initial service prototypes never developed into more mature service implementations;
- the co-existence of multiple solutions to the same problem, as a consequence, for example, of incomplete service or data migrations;
- the increasing challenge of trying to solve a problem without introducing new ones, which was also compounded by a vaguely naive approach to system administration; a typical example could be a single deamon serving multiple functions, resulting in complex configurations (one Postfix instance for all mail, or only one Apache instance for all websites, etc) and making testing changes very hard;
- not being able to test a change on the infrastructure as a whole, leading to deploying quick and/or risky fixes.

[1] https://www.autistici.org/orangebook/
[2] https://en.wikipedia.org/wiki/Technical_debt

In 2017 the situation escalated to a critical level: we detected an intrusion in our systems, due to the compromised credentials of an administrator. The forensic analysis showed that the previously mentioned imperfect or incomplete implementations played a clear role into slipping the unauthorized actors in our system.

An event of this importance forcibly presented us with an existential question: even having solved all the single problems that allowed the intrusion to take place, was the collective self-confident enough about having all the necessary skills to prevent (not only fix) similar events in the future? It is implicitly clear that the problem is endemic and due to flaws that have not been addressed and solved mainly because of the previously described *technical debt*.

This document is the description of the solutions we have come up with, once we decided that it was worth to keep investing our energies in the project, in order to both mitigate the previously mentioned problems, and to try to avoid this type of incident in the future.

2 Organizational principles

When we gathered to begin considering collectively what to do, we tried to imagine the desired final situation, and at the same time the possible paths to get there. Our reasoning has been based on general principles that are useful to be presented here.

2.1 Human and relational aspects

How to structure a project so that it can survive the test of time? The human factor is a critical aspect in an organization exclusively composed of volunteers. We strive to model the technical aspects of the infrastructure on the humans that are going to deal with them.

The A/I organization is not only composed of technical staff with IT skills, such as programmers or system administrators. We have people that continuously have to follow up with the legal, financial, communicational and political issues, in addition to the super boring bureaucracy aspects of the project. This is particularly important since we need a robust organizational structure in place when our actions, and those of who benefit from our services and the hosted content, are under threat. Members of the collective who write new code, review *upstream* code, design the services and study the security of the infrastructure, are only a subset of the whole team. In addition to them there are some of us who take the first line of support, and by necessity are forced to have more of a generalist approach. To develop a resilient communication project with some solid antagonist principles we have to focus on the human factor.

Hence we prioritized our volunteers' time and dedication to the project, trying to follow the following principles when designing the infrastructure:

- whenever we consider any technical choice we always keep in mind the cognitive challenges the choice itself will bring, for this reason we will try to:
  - eliminate the avoidable complexity at any cost;
  - make the unavoidable complexity at least manageable, by providing a clear context for the choices taken, adding documentation at every level, in order to build systems that prioritize clarity in their interdependencies
- make it possible to delegate parts of the infrastructure to people with different levels of specialization and time availability, allowing service-specific *engagements* (e.g. "I will only work on Noblogs")

  - hence the necessity of a modular infrastructure with isolated components; having shared principles and shared infrastructure maximizes the chances of someone identifying and fixing a possible future problem

- minimize the effort needed to keep the project afloat, well aware of the challenges connected to both different time availability, and "human turnover" factors

  - this point ultimately motivates the choice of building automated, distributed "resilient" systems besides the need for scalability: when planning architectures in terms of decades, the initial investment tends to be justified by a much lower operational cost in the long run (both in time and stress/urgency related terms)

- design our infrastructure in such a way that its implementation can be shared, understood, and improved by other groups and generally other people besides ourselves.

2.2 Technical aspects

The requirements listed above fundamentally determined the technical choices we made:

- **Infrastructure-as-code** or in other words the ability to control our systems by describing them as *code* (configuration files). This choice brings in some pivotal consequences both on the technical and social aspects. First of all, this is a fundamental transparency issue for us: everything that is happening is visible to everyone and, at least in principle, is also documented. This is intended to be an antidote to the privilege someone could have by having more time available to follow the infrastructure changes. In heterogeneously participated projects it is a common pattern to have smaller sub-groups of people be able to be more often online and consequently having a higher understanding of the actual state of the infrastructure and its development, while everyone else struggles to tag along. By forcing the infrastructure state into a "single document" (code repository) we avoid these nefarious consequences.
- As soon as we are able to describe the entire state of the infrastructure as code, it is also possible to adopt one of the many existing **collaborative workflows** in order to work on the code together, making collaboration in distributed groups easier. Some systems that enable such opportunities are very common (i.e. Github), and a lot of people are already familiar with them, lowering barriers to contribution.
- The right abstractions need to be in place in order to manipulate and understand your services: in a context of *infrastructure-as-code*, how to understand exactly what the code does in order to manipulate it? A barely coordinated set of configuration files is often not easily identifiable as a "service". Instead, it is important to align the *control surface* with the mental model we are trying to build. This is traditionally the domain of *configuration management* tools, abstracting technical mechanisms and filling the gap between the implementation level (for example the configuration of a specific web-server instance) and the semantic level (the fact that this specific web-server instance is part of service X).

- It is necessary to be able to **replicate** the infrastructure as a test environment, in order to safely be able to experiment or verify an hypothesis and the consequent change. In doing so you gain self confidence about deploying changes and about your personal skills. This also empowers the group and opens up to the opportunity of including new people while giving them space to test and propose changes to the codebase, without the risk of compromising the stability of the production system.
- The previous point also enables to completely respawn from scratch all services (a **restorable** infrastructure), simply by bootstrapping the code contained in the repositories. This is especially useful during routine upgrades, and in case of security breaches or other catastrophic events.
- Finally, we also want to have an effective service **isolation**, so that services become modular and compartmentalized. Modularization makes component reuse easier and consequently the system gets easier to understand, while compartmentalization makes "developing code incrementally" possible. Strict service isolation is also the foundation of the infrastructure security model, regardless of the specific implementations.

These requirements pushed us towards the choice of versioning code using git, distributing configurations using a push model with Ansible, and achieving service isolation using *containers* (instead of e.g. virtual machines).

We are aware of the fact that many of these choices are common to the world of professional Information Technology: more than being a sign of ideological support from us, this similarity is to be considered as a manifestation of a convergence of pragmatical interests. Reducing the gap between internal collective practices and the "professional" ones, when necessary, also increases the chances for people to develop skills outside of the collective itself.

3 Redefining infrastructure

3.1 The technological substrate

What we described in the previous section determined the technical choices we made. Having to conceive something that could last over time, we decided to focus our efforts on building an infrastructure that was easy to understand in its functionality.

Therefore we tried to define a high-level *service* abstraction that would solve our specific problems using an *infrastructure* of common functionalities, following these principles:

- clear service separation, in order to be able to make isolated and independent changes;
- a strict infrastructure definition limited to the bare interfaces (APIs, etc.), so that different parts of the implementation can be replaced over time without undesirable side effects.

Assuming that the right choices regarding the definition of the interfaces and the control surfaces are made (not an easy task!), such a model allows us to minimize long term maintenance to what is strictly needed, focusing primarily on the software updates required to keep the infrastructure up to date.

Any software related context involves a non trivial amount of work related to security updates and *bug fixes*. Isolating the services allows us to spread this activity over a longer stretch of time, and to let us proceed at our own pace: even upgrading the current *stable* Debian release becomes manageable without too much effort.

3.2 Choosing a "level of service"

Before taking any further technical decisions it is important to clarify what is our desired level of service. Even in the case of a volunteer-driven project like ours, where the end users clearly know that they are not offered a service with the same uptime guarantee as common commercial services, it is still important to give ourselves some expectations, in order to understand what kind and how much effort is necessary.

Let's take our mail services as an example. Since we are talking about communication, and email is still very important in people's online life, the required level of service is quite high, regardless of what we tell users or ourselves. Another way of formulating this concept is saying that there's not a lot of practical value in an email service that does not meet high reliability standards.

Therefore our objective here is to deliver the maximum possible level of availability for this service. Specifically, we are interested in a distributed service with a sufficient level of replication, resilient to single server failures (the most common operational problem we face), at least when it comes to the basic functionalities, like reading and sending mails. At the same time, if we take a closer look, some of the mail service components don't really require a high level of reliability: the main functionality of this service will anyway be delivered, even if in a degraded state. This means that for services like mail systems we can accept that some other secondary components could fail.

In general, by examining the features of all the services we offer, we have adopted two distinct architecture models, corresponding to two different levels of service criticality and quality:

- When it comes to *critical* services we chose distributed models with replication and redundancy (with specific features depending on the service, from pure replication for *stateless* services, to *leader election* mechanisms, and so on); this generally allows us to offer a better aggregated service level, in comparison to the one the single providers we rely on are offering us.
- When it comes to non-critical services we adopted a single-instance service model, with a manually managed *failover* mechanism (data is recovered automatically from backup, but the failover operation has to be manually activated). In addition to *stateless* services we also run a plethora of auxiliary services that collect different types of data, where it is perfectly acceptable to lose one or more days of data (since for example data will be periodically regenerated, etc): in these cases not developing any replication mechanism can simplify considerably the implementation of the overall infrastructure.

3.3 Towards an ideal service model

Considering the subset of criteria we outlined, and examining the service structure we offer and how they evolved during the last twenty years, we have an idea of what kind of abstractions we need:

- A service is composed by a well defined set of softwares, configurations and associated data. In our case we have a clear separation between the first two elements and the third: infrastructure automation is exclusively in charge of the first two, but *never* covers user data.
- We consider every service from a high-level point of view ("mail"), but services will be a construct made of multiple components, *daemons* and so on. This hierarchical structure is essential to keep the network of relationships between different components and services understandable to every person involved in the project.
- We strive to manage the *public* existance of a service (DNS records, certificates and so on) separately from the service itself. This is the perimeter of our legal interface, and we need to be able to manage it independently from the service implementation itself.
- We have public and "non-public" services, that have different requirements and characteristics. The purpose of non-public services is to be dependent to the public ones, implementing internal interfaces

and common functionalities, whenever this could make sense. Defining the (public/non-public) category of a service is a critical factor when it comes to managing the final complexity of the system as a whole: at a pragmatic level, the majority of the "non public" services we created (like the user panel, account admin, service manager) are dedicated to user account management, the most complex part of our architecture.

- We have quite relaxed requirements when it comes to decide what services should be assigned to which hosts: we want to have the opportunity to associate subsets of our hosts into groups (so that we have similar user data access between them, for instance), and to apply global criteria such as "N instances of this service should exist".

Describing services in this manner allows us to have a completely hardware-independent representation, so that the hardware can be treated purely as a *commodity*, merely to be considered in a quantitive manner. This perspective aligns really well with our model of hardware management, based on renting or leasing servers from commercial providers, a strategy inherited from the first R* plan.

4 Implementation

4.1 Distributed operating system

The choice of adopting a "distributed services" model stems naturally from the need of defining how to associate functionalities between services: some services are so common and general, and at the same time so tightly connected to the very opportunity of managing distributed services, that we could define them as a sort of *distributed operating system*.

Another way to look at it is to consider which functionalities could be useful from the point of view of who is trying to implement a service. For instance: my service probably generates logs and I want them to be managed and made visible in some way, without having to specify how to achieve this result separately for each and every service. Another example: my service has to communicate with other internal services and I want to be able to allow this communication to happen in a secure manner without having to define secure communication mechanisms for each pair of intercommunicating services.

These examples describe *horizontal functionalities*, potentially common to all services. Seeing them as a whole, these constitute the interface of our "distributed operating system". We will simply call it *infrastructure* in every case the context won't make this term ambiguous.

The complete list of these *horizontal functionalities* includes:

- a way to distribute code and configurations to the hosts
- a mechanism to associate services to various hosts (*scheduler*)
- UNIX separation of the services (one different user per service)
- automatically aggregated monitoring metrics
- an internal mechanism to allow services to find each other (*service discovery*)
- an internal mechanism to allow services to securely communicate with each other (we chose mTLS for this task)
- services should be able to receive traffic from internet and everything connected or required for the task should be set up automatically (DNS, SSL, etc.)

4.2 Building a bridge between *Configuration Management* and *Container Orchestration*

Combining the desire to keep every service compartmentalized and the necessity of describing them systematically altogether made us steer in the direction of modern *container orchestration*[1] solutions. At the time we started considering this decision, container orchestration had started becoming popular thanks to the *Kubernetes* project.

We immediately noticed a problem: the technical skills of the collective members were mainly focused on conventional system administration contexts. Further on we knew by experience that a shift in the technological paradigm could be extremely traumatic for a volunteer organization like ours, possibly contributing to exacerbate the internal gap among members in terms of competence level, time availability, learning possibilities, personal empowerment and satisfaction.

Hence we decided to start building a collective learning roadmap to help people move along the transition from the old paradigms to the new ones. This collective path was bound to adopt an incremental approach, introducing some fundamental concepts about container orchestration and distributed operative systems in the context of the more familiar one, that is, configuration management.

Since there was no existing product satisfying our requirements at that time, and in particular nothing that could give us the chance to moderate the steepness of the learning curve, we decided to write a tool of our own. This was a painful but unavoidable choice and the result is float[2].

Float is, in a few words, a (simple) service orchestration system built around Ansible, a rather popular and traditional configuration management tool.

There are a few comments to make about Ansible. To begin with we chose it mainly because of its *triviality*: without considering its various idiosyncrasies, it is basically a simple sequential tasks executor, not so very different from a shell script. This makes it very similar to our previous configuration management, to the point that rewriting the majority of our roles has shown to be a quite simple task. The other important Ansible peculiarity is that it follows a *push* pattern, where the changes are applied from the computer of the administrator connecting to the different hosts. This is the exact opposite of the *pull* pattern we had used until then, and was explicitly chosen as a form of adaptation to the new *threat model* in regard to the risk of security breaches.

We explicitly chose to consider float as a purely generic tool, usable without having to fully understand its implementation, as with any other open-source tool. A lot of time was then invested in keeping it minimal as well as in writing a clear documentation, producing tutorials and step by step guides, and whatever else was necessary to improve the internal learning process related to this tool.

[1] https://en.wikipedia.org/wiki/Orchestration_(computing)
[2] https://git.autistici.org/ai3/float

4.2.1 *Float*

The objective of *float* is to implement a "distributed operating system" such as the one we described above: float takes control over the *bare metal* hardware and transforms it in a service hosting infrastructure. While developing float the fundamental principle has been doing as little as possible to implement it as simple as possible: this in order to both reduce development time and to improve clarity in the final result. This explains why float could be best described as a "rough but efficient" tool.

Float combines a high level definition of the services with the opportunity of using Ansible to manage their configuration. In particular, float is very tolerant in its "service" definition, and allows us to include containers and normal systemd units (in this case presumably previously installed by Ansible). This flexibility has proved **fundamental** to migrate our existing services and to gradually transform them into containers.

Another important feature of float, possibly the one that makes it, all things considered, quite a *simple* tool, is the fact that it operates as an *offline* scheduler. Contrary to other more complex solutions, float doesn't have (!) an online component moving for example services from one host to another because of a hardware problem.

This might appear as a huge limitation, but it actually doesn't preclude the delivery of *high availability* services: we simply transitioned the responsibility of this task and delegated it to the service itself. Float can't do much for a single instance service on a single host, but when having multiple instances (by definition distributed on different hosts) the service remains operational even when a single instance is failing. We also prefer to be informed, rather than not, about a single instance failure.

The *offline* scheduler is also uncapable of applying other popular container ochestration patterns such as *autoscaling*, i.e. the dynamical change in the amount of running replicas of a service, following the same pace of the the service utilization. These are not so important features for us since our traffic patterns are very predictable.

Specifically, float is configured using a YAML file, traditionally called `services.yml`, containing the high level description of the service. The description of each service, in addition to the name that uniquely distinguish it, specifies which containers have to be configured, which ports have to be exposed, and other metadata related to the infrastructure service. An example:

```
archive:
  num_instances: 3
  containers:
    - name: http
      image: registry.git.autistici.org/apache
      env:
        APACHE_PORT: 8080
      volumes:
        - /var/lib/archive: /data
```

```
    port: 8080
  public_endpoint:
    name: archive
    port: 8080
```

This YAML snippet describes an hypothetical *archive* service, based on the container images *registry.git.autistici.org/apache*, configured to listen on port 8080 through the environment variable `APACHE_PORT` (assuming this is how this image accepts its configuration). The (server) directory */var/lib/archive* will appear as */data* inside the container (again assuming that is the path where the image expects to find its data).

The directive tells float to run 3 different instances of this service, that will be consequently distributed on three different servers. Other services will be able to connect to these instances, internally, resolving the service name ("archive") through DNS.

Further on, by defining a *public_endpoint*, we are requesting float to expose this service publicly through HTTPS (as a website named also *archive*), so float will take care of generating the valid SSL certificates, the different reverse proxy HTTP configurations and so on.

It is also possible to combine an Ansible role, when necessary, in order to perform tasks not covered in the possibilities offered by the metadata. Let's suppose our hypothetical service needs a specific file, we will consequently have an Ansible role *archive* with the following tasks:

```
- name: Create archive data directory
  file:
    path: /var/lib/archive
    state: directory
    owner: docker-archive
- name: Create archive index page
  copy:
    dest: /var/lib/archive/index.html
    content: "hello world"
```

To illustrate how the tool is indifferent to the traditional container / service dualism, let's consider an alternative implementation of this hypothetical service. Instead of using a container, it will use a Debian package (a equally hypothetical "archive-server" package) with an associated systemd unit. The description would then become like this:

```
archive:
  num_instances: 3
  systemd_services:
    - archive-server
  public_endpoint:
```

```
    name: archive
    port: 8080
```

And the Ansible role (assuming, just for the sake of making an example, that the software in the Debian package needs a configuration file):

```
- name: Install archive-server package
  apt:
    name: archive-server
    state: present
- name: Configure archive-server
  template:
    src: archive-server.conf.j2
    dest: /etc/archive-server.conf
- name: Start and enable the archive-server systemd unit
  systemd:
    name: archive-server.service
    state: started
    enabled: yes
```

Regardless of the differences in the implementation, this service is, from the point of view of float, identical to the one defined by the configuration we previously illustrated.

To sum it all up, float offers:

- a *scheduling* algorithm to distribute services to different servers
- a low level mechanism to run containers on servers
- a coherent and organized metadata structure to represent the services

Additionally float includes a serie of "infrastructural" services, collectively representing the previously mentioned distributed operating system: these are implemented using the same terms used in these last examples (they are "float services", configured using a YAML description and Ansible roles).

The last bit of the float configuration is the *inventory* (using the Ansible terminology), or rather simply a list of the servers that can be used as an input for the service assignment.

4.3 On the opportunity of writing things versus using existing code

This vision of the system architecture is not particularly original, but is well aligned with a series of *best practices* and cultural changes happening right now in the IT industry: this is not because we are keen on following the newest trends, it is more because adopting already widespread solutions is a form of warranty about long-term mantainance goals (even if one can never be so sure about the far or near future in technology).

However our situation is quite peculiar, especially since we want to be an *autonomous* and *independent* organization, using free software exclusively. We soon noticed how some of the bits and pieces necessary to implement our infrastructure vision didn't yet exist as such. Either that or they did not exist in a form useful to fix our specific problems.

We tried then to strip the problem down to the bare minimum requirements, trying to reduce as much as possible the amount of "missing bits", but still it was necessary to take a difficult decision: was it better to try to adapt an existing but imperfect solution, accepting the compromises it required, or was it better to write a new software from scratch, obtaining exactly the needed functionalities?

Unfortunately writing software is quite easy, while maintaining a software that has to keep working for many years is a much more complex and problematic task, and a responsibility towards our future selves (or future users). This is not a decision that can be taken lightly.

On the other hand, adopting existing software can sometimes represent an even bigger task: not only one has to maintain the integration code necessary to make it function in an environment the software itself is not designed to function in, but also the cognitive load over time has to be considered, a derivative of all the "special cases" not matching our architecture design. On the long run, the effect of this cognitive friction is to increase the system complexity and to decrease the amount of people able to maintain and modify it.

Hence we have chosen, in some cases, to write the necessary software from scratch, thinking that this would be a sustainable choice as long as we respected two preconditions:

- investing on *replaceability*, taking advantage of the modular structure in order to isolate the components that potentially could be replaced by a standard "stock" software in the future;
- again thanks to modularity, writing *simple* components, whose function can be understood without having to analyze the source code.

These principles have driven the implementation of all the *custom* software components that are mentioned in this document.

4.4 Source Control and Continuous Integration

A modern *infrastucture-as-code* system manages a conspicuous amount of information, authentications, configurations and software. Historically we always needed some personalized Debian packages, but after the introduction of containers the amount of this kind of artifacts has been significantly increasing.

A system to automatically generate and manage these instructions and the related variations became necessary: luckily there are several available nowadays, since Github made them popular. We chose Gitlab[3], since it offers a "groups" model that makes it possible to manage a great number of different projects. However it would

[3] https://en.wikipedia.org/wiki/GitLab

have been OK to choose any other existing software offering a *continuous integration* system (the possibility to execute scripts at every code commit).

We make use of these functionalities for a wide array of requirements, like regenerating the container images every time the associated git repository is changed, executing tests to verify code compliance, generating Debian packages to be distributed on our infrastructure and so on. Other components of Gitlab also play a key role in our infrastructure: for example the *container registry*, used to distribute the container images to the hosts.

Gitlab is at the moment a central component in our infrastructure: even if it is possible to generate all the container images manually, that would definitely not be very practical since we are talking about a quite considerable amount of images. Also to cut down the number of cross dependencies, Gitlab is managed separately from the rest of the infrastructure, so that we can consider it a "third party" service.
One of the least pleasurable consequences of Gitlab being so important to us is the fact that we have chosen not to make Gitlab available as a service to the end users, in order to mitigate the security risks involved. Unfortunately this decision de facto excludes end users from contributing to our infrastructure source code.

4.5 The structure of our software

It is now useful to discuss the structure, organization and origin of the software we deploy in production.

The importance of having a precise idea about where to get software, to have a proper *policy* about which sources to use, originates from two observations: first of all, it is useful in order to understand who are we allowing to run code on our infrastructure. Historically this role has been delegated to the Linux distributions, since they are maintaining, for good or bad, a defined quality and security level. In a public context where is very common to recompile the base containers from source this problem has become once again critical.

We appreciate a *trust* model offered by projects like Debian. Debian historically has always made us feel comfortable: the rate of updates, imposed by the release frequency allows us to keep ourself at pace with the newest changes, while we trust their developers and their infrastructure. To put it better, even if we are aware of the limited trust we have in distributions, we think is a reasonable compromise between security and usability.

However since this model of trust is difficult to apply to an ever-increasing-number of sources, we tried to keep their number strictly limited.

Another critical aspect is the need to be logistically prepared to manage these (probably multiple) sources in an organized manner: it is not so practical, for example, to have N different ways to import external code into our infrastructure, especially since powerful tools like Continuous Integration allow us to have a much better structured approach.

There are only two types of artifacts we are interested in producing, depending on the specific context in which every software operates:

- Debian packages. For ideological and familiarity reasons, all our systems and our containers (have a look at the *A pragmatic approach to containers* chapter) are based on Debian. In addition, the infrastructure requires the presence on every server of some components that by necessity have to be in the form of a Debian package.
- Container images, the fundamental "brick" used by our infrastructure. This second artifact type often includes the first: the vast majority of the container images we use are based on Debian, so we try as much as possible to install the necessary software using Debian packages.

4.5.1 Common rules for artifact generation

Since we have to fabricate a very limited number of artifacts, it makes sense to implement some shared rules about the way they have to be generated, through a standardized *build* process. Luckily Gitlab CI supports the possibility to include external *scripts* in the CI configuration. Consequently we created two different projects including rules to produce shared artifacts:

- pipelines/debian[4] to build Debian packages. The rules expect a *debian/* directory in the home directory of the repository, and compile the package in parallel for different Debian versions: generally *oldstable* and *stable*. However when Debian is about to release a new stable version they start to compile packages for that one as well.

  The generated packages are then published on an autonomous Debian repository, that we had to build from scratch (unfortunately), since Gitlab does not offer the opportunity to manage a Debian repository yet: it is nothing more than a simple VM with a web server, and a simple software[5] to manage package uploads using SSH.

- pipelines/containers[6] to build container images. These rules are much simpler (basically "docker build" only) but they feature other useful stages like a static analysis of the dependencies and vulnerabilities. The opportunity to add this *feature* to all our container builds by just modifying a single file shows how amazingly practical this approach is.

Centralizing the CI rules allows us to manage possibly complicated steps, like the release of a new version of Debian, in a gradual and systematical fashion.

[4] https://git.autistici.org/pipelines/debian
[5] https://git.autistici.org/pipelines/tools/urepo
[6] https://git.autistici.org/pipelines/containers

4.5.2 Dependencies management

An extremely important subject for us is *build reproducibility*: we want to be able to, when starting from the same code, to produce the same artifact. At any time, and independently from external systems. This is fundamental in order to be able to determine the update schedule, and consequently when and what to change, as opposed to being dependent on the rhythm imposed by the various software releases. Patching our systems following the typical patterns inherited from Debian-like distributions, where conventionally the packages are self-updating internally in the context of a *major release* can pose several difficulties: consequently we have decided that all packages of the same Debian major release are kept at the same "version". In other words, we consider having a dependency from *buster* or *bullseye* only, instead of from version X of a specific package to another version. This is acceptable since Debian has become quite good at not breaking software when releasing security updates.

In order to fill in the gap between the primary source and the final artifact we do have to manage the challenge represented by the upstream updates, a potentially infinite source of work. To minimize the possible impact of this problem we had to imagine some procedure that could be as automatic as possible, supporting the patching of software generally classifiable in four possible categories:

- *Stock* software, or software we use "as it is", fresh from the *upstream* Debian repos and without any modification. It constitutes the vast majority of all the software we use.
- Unmodified software that is not present in Debian. Here we have to distinguish between different scenarios, because even if the right answer would be to "integrate this software into Debian", this is actually beyond our reach (also because of time restraints). This means we have to look for a reasonable compromise:
  - When is easy to turn the software into a Debian package, or if it is part of the base Float infrastructure (for this reason not included in a container), we create a git fork of the original project and a *debian/* directory so that we can use our CI to build the package.
  - If the upstream source also features a custom Debian package repository, and if we consider that repository well managed, we reluctantly consider using those packages. Unfortunately not all repositories are well managed (even for very populare software), and in more than one occasion we had to "freeze" some packages by manually copying one specific version into our own Debian repository to avoid breaking our infrastructure.

  Both solutions require a moderate additional work load, so we try to keep these software package limited in numbers (the Gitlab group ai3/thirdparty[7]).
- Stock software on which we are required to apply our own small custom changes: this is the most annoying case since we have to trace the upstream changelog, so we try to avoid it as much as possible.

[7] https://git.autistici.org/ai3/thirdparty/

At the practical level this case is also managed with a *fork* of the original project in our Gitlab instance, where we then apply our *patches*. A macro example of this case is Noblogs compared to the main Wordpress package.

- Our homegrown software: in this case we simply generate our Debian packages directly, specific to the desired artifact.

Nowadays though there is a wide array of available automated tools that analyze the structure of different software projects dependencies and integrate them with the various source control systems like Gitlab. These tools make it possible to manage a considerable amount of both internal and external dependencies semi-automatically with a small effort, and make it practical to apply the "controlled version" model of explicit dependencies management. The advantage of this approach lays not only in the reproducibility of every build, but also in the explicit control of all the changes.

Specifically we use these tools:

- We wrote our own little tool[8] to regenerate the container images according to the hierarchy of their dependencies (Gitlab doesn't offer this functionality natively). In this way we can rely on the image hierarchy letting the automation take care of the upgrades.
- In order to manage the explicit dependencies, related to *version pinning* systems (common in Go, Python, Javascript and so on) we use a tool named renovate[9], that periodically checks the dependencies on Github and on other sources. This tool is designe to be integrated with the normal change revision workflow: it creates some *merge requests* on Gitlab, and it is possible to configure it so that it operates an automated *merge* if the code passes the tests successfully.

4.6 Secrets, meta-configuration and *environments*

The desire of easy "test environments" creation, so that is possible to experiment both privately or in groups, leads to some very significant consequences that we think are worth examining.

First of all let's ask ourselves: what is a test environment? It is a version of the A/I services themselves (or a subset of them, if convenient), running on a group of machines that is separated from the production ones – generally *virtual machines* created on the fly for the occasion. This test environment is different from production one basically because, in addition to using different hosts, it uses some different *secrets* and a different configuration. And you can run it locally if you want to.

The *secrets* in this case are objects like passwords or cryptographic keys, and the need for quick and easy test environments forced us to turn all of them into parameters: our service configuration does not contain any secret, but only variables referring to them, and the instruction needed in order to generate them.

[8] https://git.autistici.org/ai3/tools/gitlab-deps
[9] https://github.com/renovatebot/renovate

Similarly, the service configuration exposes other high level parameters relating to the *identity* and other specific objects: for example, our mail configuration describes how to build a mail service "like ours", but in the production configuration there is a specific parameter specifying that "this mail service name is called autistici.org".

To run all this we need four necessary components to define a specific setup (that we will call *environment* from now on):

- the generic services configuration
- a list of hosts (an *inventory* in Ansible terminology)
- a secrets subset, that can be automatically generated when necessary
- a list of high level configuration parameters, that we will refer to as *meta-configuration* being the configuration of the configuration

In practice the "live" version of "autistici.org" services is a combination of the generic configuration, the specific hosts (at least those with an IP connected to a "autistici.org" DNS record), and a subset of cryptographic credentials and variables (things like "use autistici.org as a domain name"). By varying these parameters it is therefore possible to create other installations, resembling the one in production.

This variety is reflected in our repository structure:

- ai3/config[10] contains the generic services configuration.
- ai3/prod contains meta-configuration and secrets for the production environments (we use Ansible to encrypt the secrets in the repository).
- ai3/testbed contains a script for the automated creation of local or remote testing environments (either on a local or a remote host), using Vagrant.

4.7 A pragmatic approach to containers

Having decided to deliver services using containers, we had to develop guidelines on how these should be built. We therefore focused on standardizing methodologies and approaches so that all our containers work pretty much in the same way.

The most widespread view today about how to structure a service in one or more containers follows the pattern "one container == one process". We look at it differently: for us it is better if a container represents a high-level aggregation (although maybe not exactly a *float* service), with a precise and complete interface to and from the rest of the world.

We don't see anything intrinsecally bad in having multiple processes (let' say, multiple daemons) in a single container if these are tightly interconnected with the service itself, and are for this reason meant to share it.

[10] https://git.autistici.org/ai3/config

Some practical examples can better clarify the issue:

- we find it superfluous to use two different containers for a daemon and the related Prometheus exporter - integrating them in the same container allows, among other things, to use "private" mechanisms such as UNIX sockets for process communication.
- Certain services like *lurker* and *helpdesk* include SMTP servers which are an integral part of the service API, and are meant to only communicate locally. In this case too it makes sense to run the *smtpd* process inside the container.
- In other cases services have components with clear internal lines of demarcation. In these cases it makes sense to split the service over several containers, for example *panel* and *memcache*, *irc* and its services. *Zipkin* for instance has a container for the web application and one for batch processing, etc.

Having multiple processes inside the same container requires an *init* daemon inside the container. The init daemon will manage the process' *lifecycle*: what to do when a process dies, perhaps due to an error? The following considerations are related to the specific implementation of *float*.

Due to the fact that the automation "control surface" is at the systemd unit level (for example to alert about services failures, etc) it is convenient for us that service related fatal errors emerge at this level: the container must terminate when there is a fatal problem with one of its main processes.

Beside the main processes there is a class of non-main processes: for example a mistake in a Prometheus exporter is not particularly interesting and the process can be restarted inside the container itself, without disturbing the main process.

A final requirement is the ability to run a script before the main processes are started: this is how we normally manage things like database updates etc.

Right now, the list of init daemons with the above features is unfortunately short:

- Chaperone[11], is the solution that we had chosen to use in the beginning. It worked fine but it had the disadvantage of being written in Python 3, which means that we pulled all of the Python 3 environment into each image. In any case, the project was abandoned a few years ago. And so did we.
- s6-overlay[12] is the current solution, based on s6, a minimal init system.

4.7.1 Image relationships

To reduce the resources necessary to deploy and operate these containers, it is convenient to organize services in a sequence of images gradually more and more specialized, in accordance with their generality. For example, Apache-based services have the following image hierarchy:

[11] https://chaperone.readthedocs.io/
[12] https://github.com/just-containers/s6-overlay

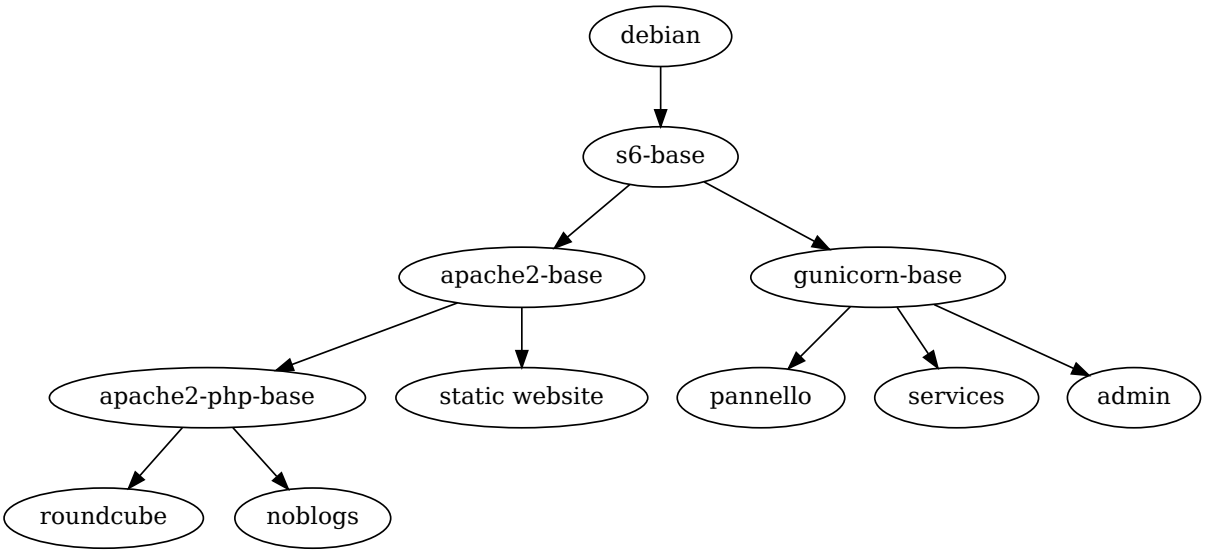

Figure 4.1: Partial hierarchy of container images from our Apache-based services

Specifically, we currently have the following images available, all based on a minimal version of Debian stable (*debian:stable-slim*):

- s6-base[13], Basic Debian plus a minimal *supervisor* (s6-overlay[14]), which it is used to run more than one process in the container.
- apache2-base[15], based on the previous image and including Apache; it can be configured at will as "children" images. For example: roundcube[16].
- gunicorn-base[17], an image with *gunicorn*, to run Python web applications.

Such modularity warrants common implementation on similar services. It also implies having a single location where to change the Apache configuration for all of our services using Apache (thus preventing us from ever "forgetting" to update any of them). Lastly, since container images are *layered*, using this image hierarchy drastically reduces the amount of data that needs to be transferred to each server. This is very important for example during testing.

4.7.2 Configuration

For many services, the necessary configuration can be passed via the environment specified in *services.yml*: in this case the service within the image is configured to read its configuration from the environment.

[13]https://git.autistici.org/ai3/docker/s6-base
[14]https://github.com/just-containers/s6-overlay
[15]https://git.autistici.org/ai3/docker/apache2-base
[16]https://git.autistici.org/ai3/docker/roundcube
[17]https://git.autistici.org/ai3/docker/gunicorn-base

However, some services require a more complex configuration pattern from the one that can be specified in the environment: in these cases the configuration will be created in Ansible, in a dedicated directory under */etc*, which will then be mounted in the container at *runtime* (by adding the directory to the service mountpoints in *services.yml*).

Often it pays off to slightly tweak the application (or use a startup script to prepare the configuration) so that you can use a single directory for all the necessary data. This can be done in order to simplify the interaction between Ansible and the container. An example of the strategy for a PHP application with a complex configuration is in roundcube[18]. In this case we place symlinks inside the image to a directory */etc/roundcube* which does not exist in the image, but will be mounted at runtime.

4.7.3 Users and permissions

Normally *float* creates a dedicated user and group for each service and runs the containers with that user *within the container*. Since at image build-time the user is not known it must be passed at runtime. Due to this fact you have to be careful while building Docker images that can be run as any user. In practice this means:

- configure the service without setuid()
- if the service needs to write data to disk:
  - for ephemeral data, configure the service to use */tmp* or */run/lock*. These are automatically mounted for you by *float* as tmpfs with the right permissions
  - for data that must persist across container restarts, you need to create (in Ansible!) a directory with the right permissions, and mount it in the container
- user-related service data must be readable (and writable, depending on the needs) by the dedicated user, and also mounted in a suitable directory (eg */home/mail*).

Using reserved ports (< 1024) is not a problem since we can enable *capabilities* like `CAP_NET_BIND_SERVICE` at runtime.

Actually this system offers an additional level of service permission isolation. For example the system ensures that a service can not write its own configuration or even write inside the container image itself. Further on, it does not depend on having a *read-only* container image to work (however the *root* mountpoint is always automatically read-only under the hood).

Looking back, we had to do a fair amount of work to ensure that all our containers could run as an arbitrary user *within the container*. An alternative could have been to define a standard user, for example UID 1000 (so that we would not need to have an extra metadata field in the service description), then configure containers

[18] https://git.autistici.org/ai3/docker/roundcube

to use that within the container with the `USER` directive. Finally, tell float to map the internal UID with the *external* user.

4.7.4 Ports

The network layout in our infrastructure is deliberately simple: instead of adopting a complex strategy of isolated containers at the network level as it is done within advanced container orchestration solutions, like Kubernetes, we have decided to adopt a model closer to traditional system administration. In this model the containers share the network namespace with the host, as a consequence they respectively have to bind to different ports. We then rely on the system firewall to make sure these ports are reachable only through the internal VPN and not from the outside.

Among the consequences of this choice there is also the fact that it is not possible to allocate more than one instance of the same service to a host (it would lead to a port conflict). Another consequence has to do with how the container expects to run: Docker images normally advertise and expose the ports of services which they implement. The ports are then remapped at need at runtime. With our approach, however, we expect the container to bind to a port that we can specify at runtime through an environment variable, so that for Docker the port mapping is always 1:1.

To do this, the image should not map a specific hard-coded port for the service, but use an environment variable (preferably, since using the environment is easier to integrate in the service configuration) or another configuration mechanism. This makes some pre-packaged container images not directly usable on our infrastructure (not a problem, since we try not to use any anyway), and has required some precautions.

A couple of different examples:

- the image apache2-base[19] and all its derivatives use the `APACHE_PORT` variable to select the port to use (Apache2 kindly expands environment variables directly in its configuration files). The thing to note is that you must use `${APACHE_PORT}` in the site `<VirtualHost>` directives:

```
<VirtualHost *:${APACHE_PORT}>
    ...
</VirtualHost>
```

- the panel[20] images, which uses gunicorn-base, reads the environment variable `BIND_ADDR` directly from the configuration file gunicorn.conf[21].

[19] https://git.autistici.org/ai3/docker/apache2-base
[20] https://git.autistici.org/ai3/pannello
[21] https://git.autistici.org/ai3/pannello/blob/master/docker/gunicorn.conf

4.8 Internal communication and RPC

An architecture like the one we describe in this document involves a large amount of internal communication between services, in the form of RPC (*remote procedure call*). This traffic is different from the data stream directly generated by users with specific protocols such as HTTP for external requests or SMTP and IMAP for email.

There are many possibilities on how to deal with this technical problem, which is why we decided to standardize internal RPC calls in a unique solution. As far as the software written by us is concerned, the standardization has not been a problem since we can choose the implementation protocol.

While we're aware of more advanced possibilities like GRPC[22], we have instead decided to choose a simple and trivial protocol: HTTP requests and responses with a JSON payload. This is not a particularly clever nor performant protocol, however the level of internal traffic is relatively low. We have preferred a solution that was simple, easily understood and diagnosed with tools that everyone knows.

For this reason we have written two libraries, in Go[23] and in Python[24], the two languages we currently use to write software. These libraries implement the minimum of functionality that we think is necessary for an effective RPC layer in this context:

- a *load balancing* mechanism to distribute requests between several identical backends, in our case a *round-robin* algorithm is more than enough;
- a *back-off* system with exponential backoff to retry requests multiple times on temporary or network errors;
- a way to make sure that every RPC request has an associated timeout, to prevent "hanging" requests;
- support for service mutual authentication via mTLS;
- on the server side, a protection mechanism to limit concurrent requests. After the limit is hit the server returns a temporary overload error.

All these characteristics together, albeit very simple, allow a service to successfully communicate with another one even in case of broken or unreachable instances. In these cases you will notice an increase in latency by a fraction of the requests. By suitably choosing the timeouts you can decide to what extent this represents an inconvenience to the final user. It is a system that can be certainly improved in many ways: adding *circuit breakers* would solve the latency problem in presence of errors, for example. While better load balancing algorithms might take into account the server load, and performing preferential fallback for local backends could significantly reduce average latencies. However, the adoption of above mentioned principles gives us a sufficiently good service quality. In other words these characteristics alone are capable of taking effective advantage of the replicated elements of our architecture.

[22] https://grpc.io
[23] https://git.autistici.org/ai3/go-common
[24] https://git.autistici.org/ai3/tools/python-web-common

In any case, isolating RPC functionality into libraries allows us to switch to a different technology with the least possible effort, should we decide so in the future.

5 Data and service structure

5.1 Users database

As the name implies the *users database* contains all data relating to user accounts on the services that we offer. This is a rather vague definition however, and can be clarified by noting that there are two main data types:

- Proper *accounts*, i.e. a pseudo-identity including its authentication-related data. An account is identified by an email address.
- *Resources*, representing specific instances of the services that we offer (an email account, a website, a database, etc).

Resources and accounts are organized hierarchically and in most cases a given account "owns" resources in an exclusive fashion. There are however also "primary" resources, such as mailing lists, that do not necessarily have an exclusive owner. In this case we keep track of the association only where it exists, to be able to show "your mailing lists" in the user panel, and at the same time allowing mailing list management by external addresses.

The users database currently contains the following resource types:

- *email accounts*, mailboxes (and implicitly XMPP accounts since there is a 1:1 relationship)
- *mailing lists*;
- *newsletters*, a different type of mailing list configured for one-way communication, and managed by a separate infrastructure;
- *DAV accounts*, named after the protocol used for access: they are used to upload websites;
- *websites*, split into *domains* and *subsites* (custom domains and subdirectories of autistici.org and inventati.org, respectively);
- *MySQL databases* used for the website hosting service.

Some resources (DAV, MySQL) may themselves contain additional service-specific authentication data.

The relationship between DAV accounts and websites is unfortunately not hierarchical, because of historical reasons. We were not exactly sure how to arrange this relationship: one-to-one (one site per DAV account and vice versa), one-to-many (multiple websites for a DAV account), or many-to-many (free association of DAV accounts, for example limited to specific subdirectories, and websites). Historically we have been in the

latter scenario, and unfortunately we don't have another object in the hierarchy in order to be able to group things correctly. As a result, the grouping is done at rendering time, in the user panel, based on the resources *path*.

A data model like this, especially when implemented on top of a non-relational database, requires the manual maintenance of many non-variables, for example:

- given the decoupling between *identities* and *resources*, it is necessary that an *email* resource is present for each account matching the account name;
- some services require multiple resources (a website, and its associated sites for example), in which case it is necessary to ensure that all such resources live on the same server;
- every website has to have an associated DAV account, without it no directory is created on the filesystem;
- some related resources have *denormalized* attributes (i.e. identical across multiple related resources), simplifying automation enormously, but making it necessary to ensure that the attributes remain consistent over time.

Luckily all services use this database in *read-only* mode so this is not here we have to focus our maintenance efforts. This complexity is contained and isolated in a single *administration component* (accountserver[1]), offering a well defined API, and that's the only service allowed to create and modify database objects. The opportunity to contain all the historical complexity and the non obvious and probably sub-optimal consequences of decisions taken in the past (like LDAP and the specific scheme we used) in one single software has been the starting point for the process of system renovation.

5.2 Data and partitioned services

The majority of our services adopts a *scaling* strategy based on *partitioning* (or *sharding*): as soon as we have more than one server, all data from each account is associated to only one server. This is an alternative to *replication*, where the data of every account is copied on multiple servers.

The fundamental difference between the two approaches, at least considering the aspects that we consider most interesting, lays in their operational behaviour when facing a server failure: replication allows to continue delivering the services, while in the case of partitioning all users whose data lays on the failing server will loose access. Since replication is obviously offering superior quality, why not choosing it? For several reasons:

- a *technological gap*: at the time when we designed this architecture, a software product able to offer a reliable, performing POSIX-compliant storage did not exists. At least according to the competence and engagement we had at the time.

[1] https://git.autistici.org/ai3/accountserver

- a geo-distributed architecture: for a POSIX-compliant storage system, having a non location specific deployment represents a *worst case scenario*, since dealing with time latencies while maintain distributed integrity is demanding. This is a problem especially for *storage-heavy* applications like e-mail and implies that it's better to take advantage of the peculiarity these services offer, i.e. they normally access *local* user data (and every user has access only to her own local data).

Later on very reliable distributed storage systems emerged, bypassing all problems by giving up the POSIX compatibility: these systems are not leveraging access to a "remotely mounted filesystem", but are instead especially designed to use high level APIs.
Unfortunately this is not a possible option for our services: both for e-mails and for websites we are using software written by third parties (Dovecot, Apache) that cannot use these APIs, strictly requiring a "real" filesystem in order to work.

The way partitioned systems react in case of problems is quite peculiar (this is true for all distributed systems) and demands uncommon cognitive flexibility, both for the administrators and the end users. We lost a binary concept of operability ("service is running" or "service is down"), and the user experience presents itself differently: when a problem occurs, it's quite common that the service is perfectly functional for some users, while not at all for some other users. Therefore it becomes necessary to modify the monitoring and diagnostic tools so that they give answers in quantitative terms, i.e. no more "up" / "down" responses but "a certain X percentage of users is experiencing problems". This obviously implies a heavier cognitive load in order to be able to do troubleshooting, a problem though that can be mitigated with well chosen analytical tools: in our case we invested a good amount of time in the monitoring system functionalities.

5.3 LDAP implementation

Because of historical reasons our user database is based on LDAP. This would not probably be the choice we'd make if we started from scratch tomorrow: LDAP is an archaic and idiosyncratic technology that requires specialist skills, and many of the reasons that made us choose it in 2005 are not valid today:

At the time, LDAP offered some important advantages:

- it's very fast
- it features a robust replication mechanism that is easy to configure
- a broad client side support (true for SQL too)

Regardless, there are also some negative sides:

- modifying the scheme is a complex operation, requiring a deep understanding of the diverse types of LDAP objects, interactions and comparators and so on.
- Low level APIs that imply almost always the need of a middle layer to the database in order to make changes

- the database management complexity tends to be on the client side
- the lack of transactions leads to the necessity of a conservative attitude when introducing changes to the data: in practice this translates into a "light" database usage pattern (like choosing to also use some external databases in order to leverage more intensive data modification) and a strict *data ownership* policy, so that the relationship between LDAP attributes and the automation processes modifying them is unique (have a look at the *Automation* chapter).

With time we gradually isolated direct access to LDAP database more and more, while also implementing the adequate ACL, reaching the actual state:

- all authentication related database accesses have been delegated exclusively to the *auth-server* (see the related chapter for more details);
- all administrative accesses requiring write permissions, to create and modify database objects, are delegated to the *accountserver* service instead, exposing a high level API;
- some services have read only LDAP direct access, for queries like verifying the existence of an account, or determining specific service parameters (like for example the mailbox path);
- The automation mechanisms have a limited write access, just to allow changes to those necessary *state machine* attributes (see the *Automation* chapter).

In service architecture terms, we rely on LDAP replication whenever it's necessary to run a larger amount of queries: specifically we replicate LDAP on the frontends and the mail servers. The replication cost is pretty low (data rarely changes), and the database is small enough. In particular, it is small enough to fit in memory: the replicas serve the database from *tmpfs*, downloading a new copy (in less than 30 seconds) at every reboot.

6 Authentication and identity

Email addresses still play the role of *primary identity* for a physical person on the Internet, and considering our role as email providers, we felt the need to update our infrastructure in order to better protect these identities.

In particular there are two features we were especially interested in, so that we could get relatively closer to the state of the art for this kind of implementations:

- *Two-Factor Authentication* by hardware token. Even if tokens cost money, it is the only serious way to prevent *phishing*.
- The opportunity to freely create limited and de-powered passwords: limited because they allow access to a single service, and de-powered since they don't allow the execution of privileged operations (like password changes), preventing an account *takeover*. This provides the end users the chance of managing their risk profile at a single device level.

Naturally, since we had pre-existing users, it was necessary to maintain the familiar username and password workflow along the new one.

Finally a modern identity management system has to be slightly resilient to *brute-forcing* and other very widespread automated abuse attempt mechanisms.

In this scenario, the credential validation logic becomes quite complicated (just consider those necessary criteria like "allow a single password login only when no other service specific passwords are defined"), so it makes sense to manage it in an single place instead of spreading it between different applications. This and other concerns lead us towards the implementation of an autonomous and centralized authentication service which the other services delegate the user authentication to.

Also in this case the open source authentication systems that were available at the time did not feature the functionalities we needed, so we were forced to write some new software. This panorama is changing and constantly improving since many of these practices are becoming quite common, for this reason also this, like other parts of our infrastructure, will possibly be replaced with *stock* components in the future.

Let's then examine the various users authentication processes in detail, explaining the solutions we implemented one by one.

6.1 Authentication server

At the lowest authentication stack level lays the main authentication server, or auth-server[1]. The only aim of this service is to authenticate end users, using their data saved into the user database. Authenticating an end user or not is a decision based on the presented credentials (password, 2FA), and possibly other contextual information (like the IP address).

We are presenting it as a *centralized service*, but it is actually a service implemented using multiple instances of the auth-server, in order to avoid creating a *single point of failure*.

The authentication server is exposing a very simple text protocol, accepting though all the additional necessary information (beyond username and password) in order to complete the two-factor authentication. Unfortunately (again) we did not find an existing protocol supporting our needs while allowing a low cost implementation (we concluded that setting up a RADIUS server correctly was out of our reach).

There are two different types of client:

- Sophisticated clients, capable of managing a 2FA authentication workflow with the end user, that can fully use take the authentication protocol. For all practical aspects, the only way to implement a 2FA workflow is through HTTP, so the whole HTTP authentication protocol is managed via *Single Sign-On* (see the next chapter). There's only only one client of this type and it is the Single Sign-On service.
- The clients implementing an authentication exclusively based on username and password terms because of the inherent limits of the associated protocol. Basically all the non-HTTP services.

In order to integrate the authentication service with this second category of clients it has been necessary to write some integration "glue" that took the shape of a PAM module, the UNIX standard for service authentication. It has to be said that the existence of this PAM modules, written in C by necessity, is the main reason behind the choice of a *custom* protocol: it is much easier to use C to implement a line-oriented text based protocol than an HTTP/JSON client. As with many other choices, if we ever change our mind it won't be difficult to migrate away from this implementation to another one, thanks to service compartmentalization.

We wrote the authentication service trying to keep it as generic as possible (it does support SQL backends as well), and one of the interesting features it offers is multiple user database support. We use this feature to isolate completely the administrative users (the A/I admins) from the "normal" end users: while the latter are stored on the LDAP database, the administrative users are stored on a static file, distributed using configuration management. This allows us to still be able to use the administrative tools using web services, even if the LDAP database is having issues.

[1] https://git.autistici.org/id/auth

6.2 Single Sign-On

Single Sign-On is a relatively unprecise term that nowadays is often associated to a spectrum of technologies spanning from identity federation to data access delegation. We are most interested into unified identity control used by separated applications, delivered by a single organization. The domain is well defined: the end users are autistici.org users, and the applications are those offered by autistici.org. Specifically we are interested into the *web* applications: there is virtually no *SSO* support in clients used for other protocols, while web is sufficiently flexible in its structure to allow the implementation of arbitrary authentication workflows.

The need for this type of technology originates from two different but convergent requirements:

- the desire to merge, from the user experience perspective, two separate web applications: the account management panel, and the webmail. Since we use the management panel as a primary communication tool with the user (as well as to implement functionalities like the password change enforcement), we want to present it as the main "access point" to our services, while having a second authentication stage for the webmail would be redundant and worsen the user experience.
- The necessity of protecting all the various administrative web interfaces present on the infrastructure behind a single access control point for the admins. The high number of interfaces discourages the implementation of a separate authentication for each one of them.

We expect the following features from a modern Single Sign-On web based service:

- it has to be organized around signed *tokens* attesting a specific *service* relationship (identified by a URL), that has to become void after a strictly defined time period. These tokens are produced by a single *login service*, implementing the end user authentication interface using password and 2FA.
- It has to be possible to verify the validity of these tokens by simply owning the public key from the login service, without the need to coordinate with a centralized authentication service.
- In case different services have to communicate internally, a mechanism able to delegate authenticated user authority has to exist. For example, it has to be possible for a service A to obtain another token for a service B valid for the same user by presenting a valid user token. This in order to be able to transfer the verified user identity across the services stack. The list of the possible transitions A -> B has to be rigidly controlled.
- All the integrations we need have to be available as:
  - an authentication module for Apache, the web server we use for applications like webmail
  - binding and *middleware* written in the languages we use (Python, Go), to integrate with our web application using their own HTTP stack
  - a PAM module for the non HTTP services
  - a mechanism in order to integrate with more complex applications that support advanced authentications workflows (SAML, OAuth)

The aspect that makes a similar implementation "web centric", as opposed to similar low level systems like Kerberos[2], is namely the focus on the browser as a primary client: basically the possibility to transfer the previously mentioned tokens at the same time with the HTTP requests (by using cookies, for example) and to control the progression of the interactive workflow by using HTTP redirects.

Open source solutions offering these functionalities do exist today, but when we made our decision Single Sign-On still was an *enterprise* feature, almost exclusively used by professionals, so we had limited and barely functional options. After giving it a good and long thought, we decided to write a minimal but functional SSO implementation ourselves.

The result is ai/sso[3], that implements all the necessary features in a very simple manner, with a few *stateless* components, easy to be integrated in our infrastructure.

After many years we can say that at a general level this strategy has proven right: we are only recently starting to see *open* alternatives compatible with our requirements, and as for many other solutions presented in this document, our modular architecture would allow us to seamless replace *ai/sso* with another implementation (OAuth/OIDC, SAML, etc) with a limited effort.

Anyway it is important to keep in mind that cookies based SSO systems, even considering the multiple advantages, show some limitations as well:

- As with all the *bearer token* based mechanisms, they are subject to identity theft by token exfiltration: since it is part of the request, the token can end up in many non-controllable places (the local browser storage, HTTP log, cache, etc). One way to mitigate this risk is keeping the token validity period very short over time.
- *Logout* is a difficult operation to be defined and implemented, since it is based on the coordination of different HTTP requests in order to obtain the cookies removal on different domains, this being a quite fragile technique. A mechanism to force the session logout does not exist (beyond waiting for it to expire).
- Since the authentication mechanism interferes with the HTTP requests, by inserting some redirects, it tends to not work completely with POST requests, making complex applications apparently more "fragile" than necessary. Luckily the instinctive end user behaviour in these cases, that is reloading the page, solves the problem.

Finally we need to consider that even if the primary client of the Single Sign-On is the browser, there are some cases in which we want that other non HTTP based services support SSO. In our particular subset of services, we also have to consider the case in which a web service is internally making a call to a "non-web" service: the main example is the webmail, where the web application has to talk to the mail service by using IMAP protocol. In the Single Sign-On context though, the web application doesn't know an user's password: hence it's necessary that the IMAP server accepts and verifies signed tokens instead of passwords.

[2] https://en.wikipedia.org/wiki/Kerberos_(protocol)
[3] https://git.autistici.org/ai/sso

6.3 Cryptography

Another aspect that we absolutely wanted to implement was the possibility to encrypt the mailbox content of every account so that every end user with her password could read its content while preventing anyone else including A/I administrator from accessing the data. This is an important feature from a legal standpoint, also considering the various terabytes of mailbox data we have been given trust about.

One possible way to obtain this result is to use a cryptographic key derived from the password. Unfortunately this simple mechanism presents some problems whenever there are more than one password, or even when one simply wants to change password. What can be done then is to create a long-term validity cryptographic key, used to encrypt the user data, and use the password derived key to save an encrypted copy of it in the database. Multiple copies of the key will for this reason exist, each of them encrypted with a different password; while at moment of the password change it is only possible to save one single new copy of the key (encrypted with the new password) without having to modify the data in any way.

What happens though when a web application uses SSO? In this case the application does not have access to the password, and still we need it to have access to the cryptographic key. There are several solutions that can be applied here, but we chose to go forward in this way:

- We created a new service keystore[4], operating as a short term cache for the user cryptographic keys (decrypted live in computer memory!);
- the Single Sign-On makes a call to this service at login time, that is when holding a valid password;
- keystore decrypts the user key using the password and holds the result in its cache;
- the application behind SSO can make a call to keystore, presenting valid user credentials (the SSO ticket), receiving the key.

In this way (and setting the *time to live* of the cache to the same duration of the SSO ticket validity) we avoid the chance of having clear text keys in memory for longer than the effective utilization time by the application. Regardless, the application has to keep a copy of the unencrypted key in memory in order to be able to decrypt the user data and this represent a (small) security threat.

6.4 Authentication mechanisms

As you can probably imagine from what we described so far, our authentication system is designed to be able to support different user authentication methods.

- Simple password. Having already thousands of users with only one password, it would not have been possible to avoid allowing authentication without second factor.

[4] https://git.autistici.org/id/keystore

- OTP[5], or a semi-randomized second factor derived from a separate device (often a smartphone application, but also similarly a hardware token). This kind of two factor based authentication is still vulnerable to *phishing* but opens to the adoption of 2FA without additional costs (as long as the user owns a smartphone).

- Hardware token (using WebAuthN[6]), a second factor provided by a physical device, not vulnerable to *phishing*.

These mechanisms are directly supported by the central SSO authentication service, but they are not sufficient to cover all the requirements for all the services we offer, especially not when taking non-HTTP services in consideration. For these services, the adopted *best practice* is to be able to associate secondary passwords to one account (*application-specific passwords*), as long as each one remains valid for only one service at a time. This limitation makes it impossible, in case of leak, to achieve complete control over the account.

In addition it's the authentication server responsibility, for example, to inhibit authentication to the mail service by using the primary password when application-specific passwords are present and so on. There are more similar rules that enhance the user experience when it comes to authenticating.

[5]https://en.wikipedia.org/wiki/One-time_password
[6]https://en.wikipedia.org/wiki/WebAuthN

6.5 Workflow in detail

6.5.1 SSO

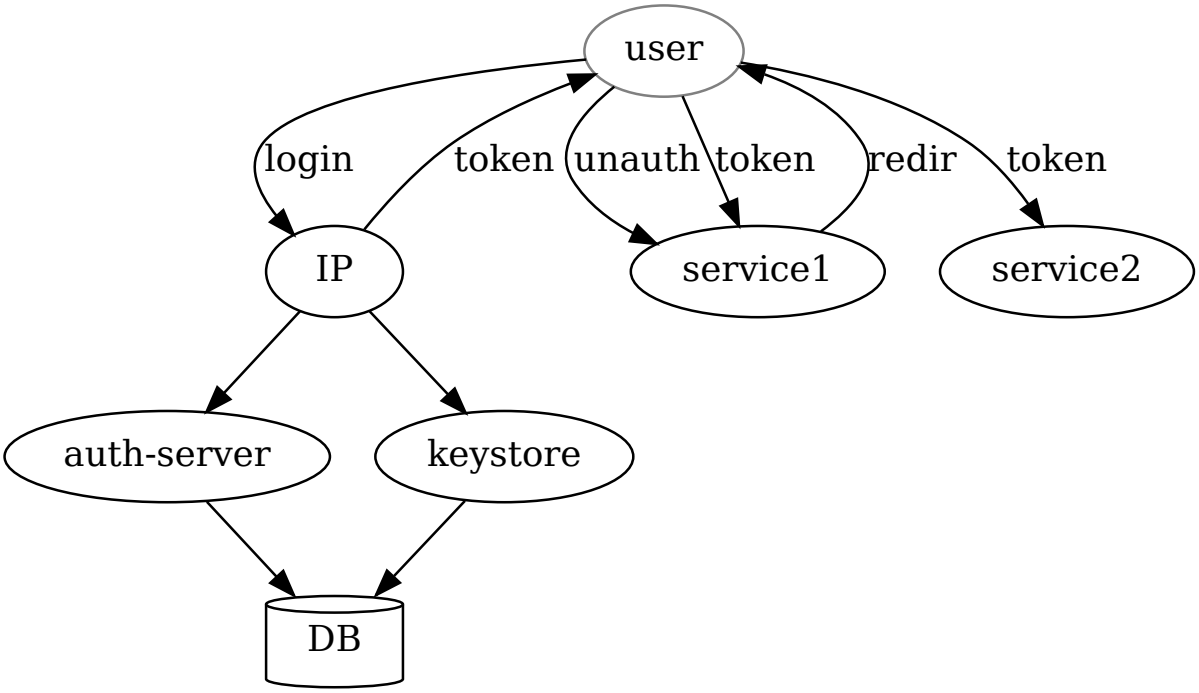

Figure 6.1: *Single Sign-On* Authentication flow

Let's follow an HTTP request in detail, as it is made by a user to a Single Sign-On protected service:

1. the user visits `https://service1/`
2. the server *service1* determines that the request does not contain any cookie with a valid SSO token, therefore returns a redirect to the login service, adding both the name of the service itself (service1) and the originally requested URL as parameters
3. the login service detects that the user does not have a valid session cookie, therefore shows an authentication form
4. the users authenticates using username, password and possibly a second factor
5. the login service delegates the authentication verification to the centralized auth server *auth-server* (that sends a request to the users database)
6. if the authentication is successful, the login service calls *keystore* in order to decrypt the specific user encryption key and keep it in its memory cache
7. the login service sets its own session cookie, so that there won't be a second authentication request, generates a valid SSO token for *service1*, passing a specific URL (`/sso_login`) with the new SSO

token and the original requested URL

8. the *service1* service receives the SSO token in the URL, stores it in a cookie, and redirects the user to the initial URL (always on *service1*)
9. *service1* now sees a valid SSO token in the cookie, and serves the request normally.

If the user, in the course of the same session, decides to visit the *service2* service, the steps are simpler since the login service recognizes its own session cookie and can for this reason skip the authentication phase, directly generating a new SSO token and continuing to the redirect. From the user point of view the whole process appears as a serie of redirects being for this reason "invisible". The same applies if the user returns to *service1* after the SSO token expires: how frequently the user is asked to re-authenticate is exclusively dependent on the validity of the login service session cookie duration.

6.5.2 Non-HTTP services

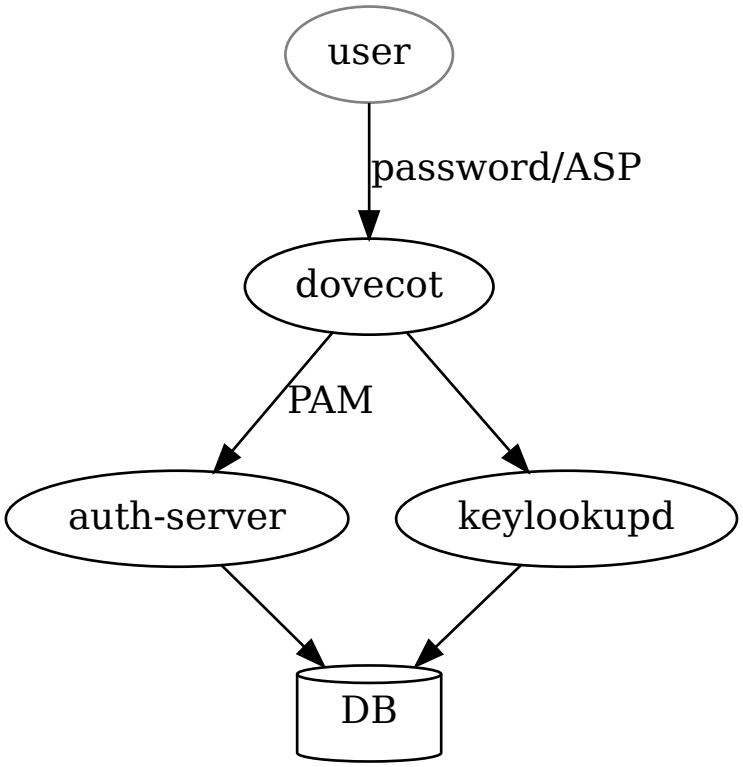

Figure 6.2: Dovecot authentication workflow

The case of a non web service is much more simple because in this case there is no second authentication factor, or a client capable to maintain a state as the browser does with cookies. If the user has 2FA activated, this is the case in which application specific password are used (ASP), and from the service point of view this

is a simple single factor authentication using username and password.

Let's have a look at *dovecot* (the IMAP service), that is interesting since it uses the specific user cryptographic credentials:

1. the service receives an authentication request with username and ASP
2. using PAM, the standard UNIX service authentication mechanism (with the *pam_auth_server* module), Dovecot delegates the authentication verification to *auth-server*, that compares the received credentials with those already present in the database
3. if the authentication is successful, Dovecot makes a call to the *dovecot-keylookupd* service, that reads the cryptographic key from the database and decrypts it using the received password.

6.5.3 Third party services authentication

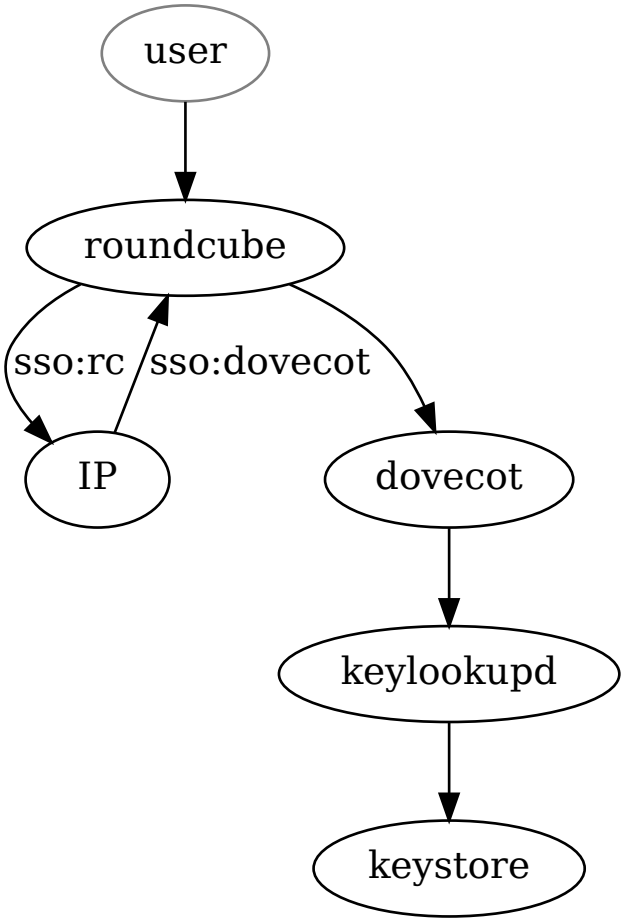

Figure 6.3: Example of authentication transfer between Roundcube and Dovecot

A case in which the earlier described authentication workflows are mixed is for example that of a service having to authenticate with *another* internal service, for this reason propagating the user credentials (as much as in addition to the *service level* authentication between the two). For example you can consider *webmail*, where the web application (Roundcube) has to authenticate with the mail service impersonating the user.

1. the user visits the webmail
2. the webmail is a HTTP service protected using Single Sign-On, so Roundcube owns a valid SSO token for the *webmail* service
3. Roundcube operates a *token exchange* request to the login service, presenting its own valid SSO token and requesting a valid one for the *dovecot* service
4. the login service verifies that the request is included in the list of "authorized exchanges", in other words verifies that a transaction between *webmail* and *dovecot* is allowed, and in case it is, returns a new SSO token for the original user, but valid for the *dovecot* service
5. Roundcube makes an IMAP request to Dovecot, using the SSO token as a password
6. Dovecot uses another PAM module (*pam_sso*) in order to verify that the SSO token is valid
7. Dovecot calls the *dovecot-keylookupd* service passing the received "password", that in fact is a SSO token
8. an SSO token is not enough in order to decrypt the cryptographic key present on the database. But in the description of the "SSO" authentication workflow we previously looked at we have seen how in one of the steps the login service saved the **decrypted** key in the *keystore* service. Here is when it gets used: dovecot-keylookupd invokes *keystore* to obtain the key and return it to dovecot.

6.6 A user database API (*accountserver*)

As explained in a previous chapter, our user database is structured in a complicated manner, in particular it requires a quite considerable amount of validation logic external to the database itself, both to enforce the "business logic" of our services, and to maintain the huge mass of denormalized data.

The *accountserver* is our solution in order to provide a centralized and standalone interface to be used for changes on the user database, implementing all this logic in a sole component.

This choice of architecture is a product of our long (and slightly frustrating) experience with the previously used strategies:

- the necessity of a high level API, above the database itself, to isolate the applications and the database structure, and to provide abstractions like *account* and *resources* instead of objects nested into LDAP or SQL tables;
- the desire of having a standalone implementation of the *business logic* where every change in the database has to pass through this logic;
- the necessity of decoupling the logic from the presentation, in order to have multiple and separate interfaces (for example it is useful, in terms of privileges and separation, that the "user" and "administration"

panels are two separate applications).

The service is implemented as a RPC service. We think that this approach presents several advantages:

- thanks to centralization, it is easier to aggregate information from multiple backends without the need to duplicate complex logics on the client side, with a limited proliferation of RPC workflows;
- privilege separation: accountserver is the only service with a write authorization into the database (controlled by LDAP ACL), all other services have only *read-only* credentials;
- a centralized service represents a more manageable target to logging and monitoring;

The RPC interface is quite extensive and includes various methods specific to account management (credential change, account recovery through secondary passwords, etc), but in general it allows the manipulation of mainly two types of objects: *account* (*user*) and resources (*resource*), contained in the account. The authentication parameters are associated to the account, while the resources represent specific accounts connected to the offered services, depending on their type.

Every type of resource also features different attributes. Every account has a primary identifier that is an email address - as a consequence, coherently, for the account taken into consideration at least one *email* type resource with that address has to exist. Other possible resources can be websites, databases, lists and so on.

Basically we transitioned from the need for a client software to know the LDAP schema to produce a valid query (including all the correct filters apt to define complex concepts like "the account is active"), to the implementation of an RPC API:

```
/api/user/get (username, sso_ticket)
```

returning, when presenting valid SSO credentials for the specific user, a JSON object with user information and all the associated resources, for example:

```
{
  "name": "user@example.com",
  "lang": "it",
  "uid": 19477,
  "status": "active",
  "shard": "3",
  "has_2fa": false,
  "has_otp": false,
  "has_encryption_keys": false,
  "resources": [
    {
      "id": "mail=user@example.com,uid=user@example.com,ou=People,dc=example,dc=com",
      "type": "email",
      "name": "user@example.com",
      "status": "active",
```

```
      "shard": "3",
      "original_shard": "3",
      "created_at": "2002-05-07",
      "usage_bytes": 412866884,
      "email": {
        "aliases": [
          "alias@example.com"
        ],
        "maildir": "example.com/user/",
        "quota_limit": 0
      }
    }
  ]
}
```

The write operations are backend processes to explicit APIs, for example:

```
/api/user/change_password (username, sso_ticket, cur_password, new_password)
/api/resource/email/add_alias (resource_id, sso_ticket, alias)
...
```

It's extremely easier for the application itself to operate on this type of data as opposed to needing to act on the result of LDAP queries, and this approach allows to "hide" all the horrible database implementation details behind a more elegant and maintainable API.

Since even the read-only workload is quite high on accountserver (every access on the user panel implies a request for the account data, in order to show the user a list of her resources), and since we still have a local LDAP database for every server, accountserver implements a *leader/follower* model. We basically execute multiple instances of accountserver, but only one of them is responsible over the write operations on the database: the other instances offer a read-only service, and possibly forward write requests to the *leader* instance.

Finally an application *accountadmin* exists, taking care of accounts management and reserved to administrators, providing the execution of various high level operations (closing / opening of accounts, moving accounts between servers, setting options and so on).

7 System *observability*

The ability to monitor, measure and analyze in detail the behavior of a system is essential to reduce the maintenance work required by the system itself. Specifically, maintenance is attention and time. These two resources are by far the most scarce, and therefore they should be invested with extreme attention. We have adopted a few approaches to address this problem:

- Building *resilient* systems, that don't need specific attention to temporary events since they autonomously return to a stable state. For example, by having multiple *front-ends* to handle traffic we don't have to worry about network problems at our providers.
- Having enough coverage provided by monitoring systems so that we get notified when there is a real problem. Especially useful for building the confidence that when there are no alarms everything is working fine.
- Having analytical tools to quickly identify the causes of problems and lead us to their resolution.

These last two roles are performed, in cooperation, by the *monitoring* and *logging* services.

7.1 Monitoring

By *monitoring* we mean the analysis of real-time performance characteristics of the system, derived both from the observation of internal system parameters (*whitebox* monitoring), and of "external" behaviors (*blackbox* monitoring).

The goal of the monitoring system is to collect and store these metrics, and offer a mathematical language apt to manipulate them. The goal is double folded: on one hand, monitoring has to allow us to evaluate metric trends over time for planning and troubleshooting purposes. On the other hand, it allows us to define anomalous states or errors as analytical expressions (rules) over the metrics. These rules are called *alerts*: their definition is subtle and hence deserving further explanation.

7.1.1 Alerts

Good *alerting*, i.e. the process that notifies admins when something is not working, is in our experience very difficult to achieve but also extremely important.

For a volunteer project, that does not have an on-call rotation to address problems day and night, the first thing to do correctly is to explicitly define the expectations of our end users. You have to be honest and transparent and carefully communicate the level of responsiveness users can reasonably expect (which will be presumably quite low).

Part of the solution is to make it generally not necessary to intervene quickly in the first place, for example by deploying redundant services.

There is another aspect to consider: in a context where nobody gets paid to investigate and solve problems, it is very important that the monitoring system does not waste people's time. For this reason is necessary to have alerts that are:

- *Accurate*, meaning that they are generated only when there is actually a real problem with immediate implications for the quality of the services offered. In other words the alerts should have a high signal-to-noise ratio.
- *Informative* i.e. they should quickly point the admin in the correct direction of the problem.
- *Comprehensive* in the sense that they should offer as much coverage as possible over the services offered. It goes without saying that it is impossible to detect a problem there where no one is looking.

Failing these objectives results into a useless and/or frustrating monitoring system, that in turns will sooner or later end up being ignored. Luckily technology can help in building a successful system, one that minimizes both disruptions to users and admins efforts.

We have based our monitoring on the *symptom-based alerting* principle, i.e. on the idea that alerts should necessarily correspond to an evident and demonstrable disservice to users. This approach is generally in opposition to "causal" alerting, where you check internal parameters assuming that they correspond to potential causes of problems. The symptomatic approach recognizes that in a complex distributed system, this inference is often impossible and generates a significant amount of noise.

Let's look at an example to clarify the difference. Consider a web application: we can have alerts for conditions such as "CPU usage is 100%" or "disk space is full" (and many other conditions of this kind) implying that if one of these conditions is true, presumably the application is not working properly. The symptomatic approach instead prefers to check conditions like "the site is serving more than 1% HTTP errors" or "the 90th percentile of latency is greater than 100 milliseconds", which are *directly* related to user experience. This approach assumes that if these conditions are false all the rest of the system is, by definition, working as it should. The advantage of focusing on what is directly visible to users is that there is no risk of "forgetting" causal conditions that could potentially induce problems without us knowing.

Therefore let's define two types of alerts, with distinct purposes:

1. Those describing a symptomatic problem and requiring immediate attention from an admin. We identify them with the severity level *page*, and activate the notification process that results in sending emails/messages/etc.

2. Alerts that simply report an anomalous state, without any notification associated. These cover a lot of what is normally considered "causal alerting", and they are used for the debugging phase. We have a dashboard that reports the overall system status and can help to locate the source of a problem, once a notification of a symptomatic alert has been received.

Another important objective of the alerting system is to be as non-verbose as possible while still being effective: in theory we would like to receive a single alert for each "incident". By not paying attention to this aspect there's the risk of overloading admins with useless information and noise, even in the presence of an actual problem, contributing to cognitive fatigue.

The alerting system we have adopted implements a standardized alert hierarchy that takes into account knowledge of possible *failure domains*: the result allows us to ignore, for example, "local" alerts in the presence of "global" alerts (e.g. we are not interested in knowing separately that a specific server is broken if all are), or alerts that refer to a service on a specific server if the server in question is unreachable.

7.1.2 Blackbox monitoring

One of the investments that have "paid" the most in terms of usefulness has been writing a collection of tests for various services, in fact a *probe* for blackbox monitoring. This software is called service-prober[1] able to manage all the protocols related to services that we offer (HTTP, POP, IMAP, SMTP, etc.) and run custom tests.

This allows us to keep the following resources under continuous monitoring:

- the user panel and webmail functionality, including verifying the capability of reading / sending mail from the webmail itself;
- the ability to receive and send mail from and to external servers;
- mailing lists operativity

and other "complex" deliveries. We are using dedicated test users (both internal to our services, and externally) obtaining a wide coverage of the functionalities of our services. Having the ability to keep these under observation allows us to quickly react to any report of failure and its impact. The tests are designed to be written using terms that are immediately relevant to users.

Further on, the software keeps complete logs of each transaction made with the supported protocols: when an error occurs a detailed log is already available.

[1] https://git.autistici.org/ai3/tools/service-prober

7.2 Logging

With *log* we refer to the wide set of standard outputs of the processes running on our infrastructure. This is a relatively large aggregation of different data types, from event logging (such as HTTP requests) to diagnostic messages. Some of this data is structured according to predefined schemas, other is simply free text. Whereas monitoring presents an aggregated view of the system performance ("number of HTTP requests received"), logs represent the detailed tracing ("*these* are exactly the HTTP requests received").

Precisely because of this detailed information they contain, logs tend to include potentially sensitive data relating to our users' privacy, for this reason they are treated with particular care and respect, as it was radioactive material: isolation, rapid disposal, and above all an explicit and controlled "chain of custody".

At the same time this detailed information is often essential when attempting to solve a problem, therefore there is a tension between the need to keep logs for diagnostic purposes, and the desire to delete them since they constitute a risk and a burden. Beyond that, management of logs and the relative data is covered by legislation, so there are legal implications to consider.

We decided to work around this tension by implementing *anonymization*, obtained by applying different measures at different levels:

- Where possible, problematic information simply is *not included* in the logs in the first place.
- Logs are rewritten on the fly, eliminating unwanted data, *before* being stored on disk.
- We established a *time horizon* by deleting logs older than a certain (rather short) period of time.

This decision obviously bring consequences, first and foremost the inability to debug problems that have occurred beyond the established time horizon. In practice, however, this turns out to be a less significative culprit than one might imagine: ultimately, it is more important to be able to solve a problem that is occurring *now* rather than a problem that has happened in the past and has resolved itself since. Similarly, for example, the lack of IP addresses in our logs has never really constituted a problem, simply because usually knowing these addresses does not provide additional diagnostic value. Clearly there are mechanisms that have knowledge of these IPs internally, for example request rate-limiting systems, but they keep logs exclusively *in memory*, and without being able to establish any correlation with requests made by users.

Therefore the logging system we have adopted has the following characteristics:

- Offers a global view of all logs generated by our infrastructure – we implement this with a centralized system that collects logs from various server processes into one database (we use Elasticsearch for this purpose).
- It offers us a language for complex analysis and research, suitable for the multiple types of data present in the logs. This is a case where we voluntarily use a *complicated* tool (Kibana). Simple tools like *grep* and *awk* undoubtedly cover 90% of the analytic cases, but for scaling related reasons we find ourselves in need of being able to make more productive use of more advanced features.

The implementation is pretty straightforward, as a consequence of the above reasons, and relies on *journald* to collect logs from all processes in execution, which then forwards them to a local *rsyslog* server on each host. Hosts then forward their logs to a central collection *rsyslog* server. Here they are subsequently archived into Elasticsearch. The only feature perhaps worth of note is that in this process the logs do not touch the disk until the last step, important aspect since our disks are slow and very busy.

7.2.1 Extracting metrics from logs

Sometimes it is useful to be able to extract *real-time* metrics directly from logs, for example when it comes to software that is not in able to export metrics directly. Two common examples are NGINX and Postfix, which do not export metrics but produce logs with a detailed trace of each transaction.

In this case we use mtail[2], just one example of many similar programs, which reads logs line by line, and using regular expressions is able to derive metrics. For example from a normal NGINX access log entry it can calculate a "number of HTTP accesses" metric. We also track additional fields such as the host, method, status of the request, etc.

When defining specific fields for these metrics, you should take into account their cardinality (i.e. the number of distinct values they can have): monitoring systems, because of their nature, generally perform poorly with high cardinality fields. In such cases it is better to rely on the structured log database, discussed in the following section.

7.2.2 Logs as structured events

All logs produced by our systems have a structure, at the very least the minimal *syslog* schema, with attributes for host, *facility*, priority, process, etc. that enables us to perform accurate searches.

Other logs extend this schema: our logging system supports the ingestion of "structured" events, nothing more than JSON objects (we use a somewhat obsolete but extremely simple standard called Lumberjack to transport them transparently via syslog).

Many of our logs, on the other hand, are structured differently: they report information in plain text but with a fixed structure (e.g. HTTP logs). In this case it suits us to transform this fixed structure into an explicit schema, that the log search engine understands. Thanks to a series of regular expressions, upon final log collection we then transform things like

```
noblogs.org - - [03/Nov/2020:10:59:18 +0000] "GET / HTTP/1.0" 200 ...
```

in (for example JSON):

[2] https://github.com/google/mtail

```
{
  "vhost": "noblogs.org",
  "method": "GET",
  "uri": "/",
  "status": 200,
  ...
}
```

The fundamental purpose of this data, beyond facilitating searches, is to be able to generate complex aggregate reports using metadata (for example using web *dashboards*).

It is worth underlining how the monitoring system and the database of structured logs differ from each other. Especially in the case of metrics derived from the logs, we are faced with a spectrum of possibilities. The monitoring system provides data in *realtime*, suitable for an alerting system with an overall vision of system performance, cheap to elaborate and to store. The logging system instead provides an analytical database suitable to answer complex queries, which are expensive to calculate.

Let's take the example of the HTTP logs mentioned above, and consider in particular the fact that the URL field (with high cardinality, controlled by the user) is *not* stored by the monitoring system: in this case the system can tell us quickly, in real-time, what fraction of incoming requests are returning a 500 error. While the logging system is better suited to analytical queries such as "the list of 10 URLs that gave the most errors in a given period of time".

8 An example in detail

8.1 The A/I static website

The main A/I website (https://www.autistici.org/) is a simple website serving seldom changing static contents. The source code is a list of Markdown files, and an HTML template. The total size of the content is quite limited (< 1 GB). In addition there is also an associated small search motor as a stand-alone application written in Go.

The HTML pages generation process though is quite elaborate, since some HTML templates have to be applied to the Markdown pages, the Javascript code has to be compiled, and the search motor index has to be generated.

Because of these features, we decided to move the whole complexity to the container image *build* process. This process uses a multi-stage build to obtain a final and monolithic image including both the contents and the necessary services to serve them. The only public interface in the container is a HTTP port (plus an auxiliary port for monitoring). This makes it easier to test the container even locally without any special setup.

8.1.1 The application itself

The web application "sito di A/I"[1] it's slightly more complex than a static website: to begin with we need an advanced web server (like Apache or similar) in order to redirect the users requests to the content in the correct language, in addition there is a dynamic component delivering the search results.

By using already available components we can build a container featuring the following processes:

- an Apache server to serve the static contents via HTTP (we need Apache in particular because the website uses *content-negotiation* and *URL rewriting* extensively) so we start from the *apache2-base*[2] image;
- a Prometheus exporter for the Apache metrics, also present in the *apache2-base* image;
- the search engine server, listening inside the container from a separate HTTP port, receiving the requests from Apache for the `/search` URL.

The container has in turn to be compatible with the float *runtime environment*, in particular:

[1] https://git.autistici.org/ai/website
[2] https://git.autistici.org/ai3/docker/apache2-base

- has to tolerate the execution by a non-root user
- has to tolerate the execution of a base read-only image

Luckily for us, the *apache2-base* image already offers these features, so we don't have any especially challenging task here. Besides, this image offers the opportunity of configuring some parameters through environment variables, like for example the web server port (`APACHE_PORT`). This feature is quite neat since we then can skip using container-external configuration files.

The `Dockerfile` for this project will be similar to:

```
# We omit the various compliation stages for the static files,
# the search engine compilation and index generation,
# and just present the "COPY --from" commands for the relative artifacts

FROM registry.git.autistici.org/ai3/docker/apache2-base:master
COPY --from=gobuild /go/bin/sitesearch /usr/sbin/sitesearch
COPY --from=build /src/index /var/lib/sitesearch/index
COPY --from=assets /src/assets/templates/ /var/lib/sitesearch/templates/
COPY --from=precompress /var/www/autistici.org/ /var/www/autistici.org/
COPY docker/conf/ /etc/
RUN a2enmod headers rewrite negotiation proxy proxy_http \
    && a2dismod -f -q deflate \
    && chmod -R a+rX /var/lib/sitesearch /var/www/autistici.org
```

The only file present in the *docker/conf* directory is the configuration of the default Apache2 virtualhost.

8.1.2 Service configuration

The container has a very small *footprint*, both in terms of size, and in terms of necessary resources in order to run it in production (since we are talking about a static website, Apache is doing relatively little work). In addition we are talking about a completely *stateless* application so we can run multiple instances in order to achieve *high availability* without the need of special setups, since the infrastructure we have is capable of load balancing HTTP traffic on multiple instances.

From the '*orchestrator*' point of view, we don't really need to know any special details on how the container works internally. All we need to know is:

- the container speaks HTTP from a port we can choose by controlling the environment variable `APACHE_PORT`;
- the container exports some metrics (via HTTP, on the `/metrics` URL) on a different port, by default `APACHE_PORT` + 100.

The service description in the format used by *float* is quite simple:

```
web-main:
  scheduling_group: frontend
  containers:
    - name: http
      image: registry.git.autistici.org/ai/website:master
      port: 8081
      env:
        APACHE_PORT: 8081
  monitoring_endpoints:
    - port: 8181
      scheme: http
  ports:
    - 8081
```

We have chosen the port 8081 manually, knowing that it was available (not used by any other service).

Here we lack a *public_endpoint* definition that would have generated a NGINX configuration automatically: since we host websites as subdirectories of autistici.org and inventati.org, this special configuration is generated by our automation scripts (have a look at the *Automation* section). But if this wasn't the case we could have added a section:

```
web-main:
  ...
  public_endpoints:
    - name: www
      port: 8081
      scheme: http
```

in order to automatically reach the website as *www.autistici.org*, with the correct DNS records and valid SSL certificates in place.

After applying the above configuration in production, we expect to find some systemd services (on the hosts belonging to the *frontend* group) named as *docker-web-main-http*, and in fact:

```
# systemctl status docker-web-main-http
* docker-web-main-http.service - web-main/http
     Loaded: loaded (/etc/systemd/system/docker-web-main-http.service; enabled;
     Active: active (running) since Sat 2022-09-03 15:51:25 UTC; 1 weeks 0 days
   Main PID: 2046356 (conmon)
      Tasks: 103 (limit: 19095)
     Memory: 109.8M
        CPU: 1h 7min 24.733s
     CGroup: /system.slice/docker-web-main-http.service
```

```
|-2046294 /usr/bin/podman run --cgroups=disabled --replace --sdnot
|-2046356 /usr/bin/conmon --api-version 1 -c 6e4babea0b901a109c24b
|-2046359 s6-svscan -t0 /var/run/s6/services
|-2046385 s6-supervise s6-fdholderd
|-2046464 s6-supervise sitesearch
|-2046465 s6-supervise apache2
|-2046466 s6-supervise apache-exporter
|-2046469 /usr/sbin/sitesearch --index=/var/lib/sitesearch/index -
|-2046470 /usr/sbin/apache2 -DFOREGROUND
|-2046471 /usr/bin/apache_exporter -scrape_uri http://127.0.0.1:80
|-2046545 /usr/bin/logger -p daemon error -t apache
|-2046559 /usr/bin/logger -p local4 info -t apache
|-2046560 /usr/sbin/apache2 -DFOREGROUND
`-2046561 /usr/sbin/apache2 -DFOREGROUND
```

8.2 Change lifecycle

In order to understand how we interact with this kind of system in practice we can have a look at the lifecycle of an hypothetical change to the service we just considered (the main website).

The first step consists in the creation of the change in question, and the upload on the code management platform:

- Someone makes a change to the website source code: for example exchanging the collective logo with an animated GIF showing a cub. This change gets committed in a new *branch*, let's call it *cat*. Contextually to the branch, a *merge request* gets created on git.autistici.org so that we allow other collective members to review, write comments and eventually approve the change.
- The CI system creates a Docker image with the source code from the new branch, that consequently gets the name `registry.git.autistici.org/ai/website:cat`

8.2.1 Testing

The most important part in the management of system changes is verifying the correctness of the changes themselves. The great majority of the software we use has an associated *test suite* so that is possible to verify that it is functioning correctly. This is though not even remotely sufficient in order to be able verify that combining all these different pieces of software, and their configurations, will satisfy our expectations in terms of functional services. For this reason it is necessary to execute some integrations tests at system level, so that we can verify exactly this aspect.

For this reason we worked on the possibility of creating test environments at will, by spinning up virtual machines (normally locally on our own PC) configured with all or just a share of our services and a configuration similar to the one in production. The modular nature of the configuration makes this setup possible: the system automatically creates all the necessary credentials, so a minimal amount of example configuration is sufficient to create an autonomous version of our infrastructure.

Beside the configuration, it is also important to include some test data, in order to have something (user accounts, websites) to test! In our case we have some test user accounts, with a standard password, and some other specific data relative to different services like websites, noblogs etc. Noblogs is maybe the most representative case of how useful this type of work is: even if it has been necessary to write specialized tools to obtain a completely anonymized dump of the data from some specific blogs, we can now manually test new changes to Noblogs in a particularly extensive and complete way. Therefore it's easy to compare the behaviour of our software after a new change with the one actually in production.

When it comes to the specific case we examined earlier, once this new `:cat` image of the container exists, we need to verify its correctness, not only indipendently but also in the context of the whole infrastructure: this is actually not so difficult in the case of this service without dependencies, but is easy to imagine how important this process can be in more complex cases.

To specify we want to use this new image together with all other services, we have to modify the service configuration (*ai3/config*[3]) so that we refer to the image:

```
web-main:
  ...
  containers:
    - name: http
      image: registry.git.autistici.org/ai/website:cat
      ...
```

Right after can we choose between different further steps:

- It is possible to manually create a test environment, locally, from this new configuration, and then execute all the manual tests we might want.
- The CI brings up a seie of VMs. The ai3/config repository is configured so that it can automatically execute a serie of automatic simple base functionality tests (mail, web) at every commit. A test environment with the new changes gets installed, and the integrated test suite is run. The tests don't go extremely in depth, but they allow to quickly identify possible big mistakes, verifying there are no alerts (so that all services are started correctly) and the possibility to authenticate with the various services is tested.

[3] https://git.autistici.org/ai3/config

- Remote test environments are created, without launching the testbed locally on the laptop: this can be useful for travellers, or for anyone with a less powerful machine. It is sufficient to create a MR of the repository ai3/config: in this case, Gitlab will automatically create a test environment on some temporary VMs (that will be nuked after a couple of hours) in order to execute manual tests. It will be possible to connect to them by SSH or using a dedicated SOCKS proxy to test the service state.

8.2.2 Conclusion

In order to trim the changes made, by using an iterative process, it's enough to update the branch and run once again the desired test environment automation. These steps tend to be reasonably quick even when they are not locally executed (we anyway rely on git.autistici.org in order to distribute the new version of the container).

When the change is approved, the branch gets included in the main branch (*master*), and the next *push* of the configuration will update the software in production.

Later on the temporary test environments are not necessary anymore and can be destroyed.

It is important to acknowledge both the importance of being able to rely on a sufficiently mature system managing the code and the CI, (in order to facilitate collaboration, and to have a container-build-and-distribution platform) and the fundamental role of the test environments to evaluate the effect of the changes *before* they reach production. Since the test environment is intrinsically similar to production (although it requires occasional manual labor in order to have sufficiently representative *data*) this process saves us a lot of time and effort, and gives us more self confidence at the time of action.

9 Automation

This chapter is about automation processes relative to the user accounts: once we have an infrastructure capable of offering multi-user services and a database containing the list of the accounts on these services, how do these two relate? For many services the answer is quite trivial and requires a direct connection: the service can simply look for the configuration parameters directly in the database. Not all services work this way though, and in any case the plot thickens when we consider also the accounts associated *data*.

What we actually end up having to do, more often than not, is *syncing* different sets of information: on one side the user database, on the other side some other internal service specific database, or alternatively (very often) the filesystem.

Many possible approaches exist to the problem of synchronization, and the one we chose, using it since 2005, reflects partially the technical skills we had at the time, and partially the technologies that were available at the time: we had a fast and *easy to replicate* database, while there were no well widespread *log-based* technologies. In any case we still think that this approach is the best one, because of its robustness.

9.1 State machines

Let's compare the account automation tasks to a model, saying that they can be seen as a serie of *finite state machines*[1], where the states are memorized in the LDAP database, and the transitions are periodically executed by a serie of *scripts*. Normally a script is dedicated to a certain change, sometimes restricted to a particular type of account (mail, web, etc). The account states are independent from one another, so the automation can progress on every account independently, and the mechanism is naturally apt to be parallelized.

The execution states for these scripts correspond to the "beats" of the whole change automation frequency, revealing the speed of the synchronization process when distributing the changes from the database to the different services. Because of how our automation is structured, this does not represent a limit (once explained both to admins and end-users that no change is immediate and a non zero propagation time is to be expected).

Let's consider in detail, for example, the migration process of the account data from server A to server B for a partitioned service, where the account data is associated to a specific server. This workflow uses two account associated object attributes from the LDAP database, *host* and *originalhost*:

[1] https://en.wikipedia.org/wiki/Finite-state_machine

- At the beginning, the account has both *host* and *originalHost* set to A
- The workflow is started (manually, or by an automatic process) changing the value of the *host* attribute to B.
- A periodical script on server B notices an account with *host* set to B but *originalHost* different from B, and starts copying the data from the host specified by *originalHost*, that is A.
- When this scripts terminates sucessfully, sets the value of the attribute *originalHost* to B.
- A periodical script on server A notices that the account has both *host* and *originalHost* set to a different value than A, so it deletes the account associated data.

The account has therefore transitioned through these following states:

- *host=A*, *originalHost=A*
- *host=B*, *originalHost=A*
- *host=B*, *originalHost=B*

The account is in a well defined state at every step: depending on the specific service it will be possible to choose to serve the account accordingly to the *host* attribute (the account will temporarily appear as "empty" and it will be populated with the backfill of data from server A), or the *originalHost* attribute (the data will remain accessible but the changes happening during the migration will be lost), or simply suspend the service for the account during the period in which *host* and *originalHost* return different values (an approach that is useful in order to maintain data integrity in respect to modifications, at the cost of a temporary unavailability).

It has to be noted that the implementation of state machines described above is efficient as long as the tasks we need to complete are simple, but it rapidly becomes unpractical if the amount of states multiplies.

9.2 Reconciliation

With the simple state machines described above we implement an approach to the synchronization of data known as *reconciliation*: assuming that the database represents the primary authoritative copy of the data, or to better say expresses the *expectation* from the service configuration, we compare the database with the effective state of the system, and we execute operations scoped in order to eliminate the differences.

This approach is often compared to the *task queue* mechanism, where the result of the operation meant to be executed on an account is represented by a *task* that will be executed in its entirety and as soon as possible.

The reconciliatory approach presents some advantages that makes it more desirable in our case:

- It's an intrinsically robust and *fault-tolerant* infrastructure: machines can be unreachable, processes can fail, and the system will eventually converge to the desired state anyway, making progress whenever possible.
- The script activity is completely local and isolated, so scripts can be very simple and easy to understand.

Obviously some disadvantages do exist, starting from the fact that an *eventually consistent* approach implies the existence of intermediate states that are potentially *inconsistent*: since we manipulate very simple states (a mailbox either receives mail or it doesn't) inconsistency manifests mainly as a propagation delay between the expression of the intention (database change) and its effective change distribution, without introducing service related problems. Other inconsistent states can appear whenever coordination may be necessary: for example between the HTTP reverse proxy configuration and the configuration of the associated service, since these are configured independently. This also doesn't end up constituting a big problem: since this type of transitions engage either when a problem occurs or at creation time for a new resource, a temporary unavailability of the associated resource is difficult to notice.

Lastly every automation iteration requires a complete scan of both the database and the actual state of the service, an operation that can potentially be expensive on big volumes (but not yet for us, and not for a couple of magnitude of users account more). One of the ways to avoid this problem, if it ever becomes necessary, is to reduce the execution frequency of the scripts, but that of course would imply a corresponding slowdown in the propagation of the changes.

The main disadvantage of this approach is the confusion created by the length of the *feedback* loop between an action and its effect: the fact that after having for example modified the user database absolutely nothing happens for tens of minutes suggesting that there is a problem in what has been done. Training becomes very necessary in order to internalize this mechanism, representing a big obstacle, although this is compensated by the notable robustness offered by the reconciliatory approach.

9.3 Configuration generators

We mentioned how some services require an explicit configuration for every account, while some others don't. Let's consider two specific examples to clarify what we are talking about:

- At one extreme of the configurability spectrum we have services like *web hosting*, where we have to create a separate configuration file for every site present on our database. Apache is a very flexible and powerful software, with an extremely wide configurable surface, requiring consequently a very verbose description of what has to be done.
- At the other extreme there are services like IMAP and XMPP, where instead the only user specificity lays in her identity: here we can easily get along with a global service configuration, and a direct connection to the user database.

A certain amount of our automation scripts is for this reason dedicated to the pure generation of complex services configurations, generically using specific *templates*. When these scripts detect a change in the generated configuration, they automatically restart the associated service.

The trade-off between a static configuration and a dynamic one is also sometimes a matter of complexity

and performance: to make an example, it is in practice possible, with the help of particular modules, to configure Apache in a "dynamical" way, taking the list of the hosted website directly from a database. This in turns introduces a direct dependency from the database, to the point where the web service cannot function without it, while in the static case a failure at the database level only gets in the way of changes propagation. It has also to be considered that a dynamic Apache configuration requires access to the database for every HTTP request, becoming a problem with a high traffic volumes. Whenever both types of configuration are possible, from time to time it becomes necessary to consider also:

- the configuration change frequency, and the cost of *reloading* a service - if the frequency is very high a dynamic configuration becomes preferable;
- the adaptability of the data models: if some operational logic is necessary in order to adapt the data from the database to the type accepted by the application, can be easier to express it by using a *template*.

9.4 Account and data operators

Since in the majority of cases there is *data* associated to our database resources, a significant share of our automation is dedicated to data management, that is synchronizing the resource state (from the database) with the filesystem (or other service specific database).

In particular these management tasks can be categorized in three groups:

- Service *creation*: some services (for example web hosting) do need that the relative filesystem directories exist, with specific permissions, to be able to correctly operate. For others (for example mailing lists) it is necessary to create some entities in the specific service database.
- *Deletion* of the resource not anymore necessary: when a resource gets deleted, we want also the associated files to be eventually removed from the filesystem.
- *Moving* of data from a partition to another. This script category implements the previously described state machine for the transition of data between servers, copying data from the "old" server and modifying the database. For all practical purposes, when there is no specific service protocol available (for example MySQL in order to copy the database) the copy is executed using an internal *rsync* service dedicated to this task.

The first two tasks are generically incorporated in a script that uses a trivial *subset difference* algorithm, in order to find the accounts that need to be created and those that need to be removed in a single pass:

- list the effectively on-system present accounts
- list the database present accounts
- create the accounts that are in the database but not on the system
- delete the accounts that are on the system but not on the database

Data removal is a dangerous process since it's irreversible, so when it comes to public resources like websites we pay special attention: before operating deletion of *public* data from the filesystem, we first archive the data on a dedicated backup service (namely *cold storage*) that preserves a long term copy.

9.5 Inverse reconciliation

There are cases where the user database *is not* authoritative for all the resource parameters. This happens when a service has its own interface able to modify these specific parameters, and we could not or did not want to modify the software in order to make it communicate directly with our database. There are few of these cases:

- The list of mailing list administrators can be modified by the same administrators through the Mailman web interface. This is a Mailman native workflow and we want to keep supporting it, in order to avoid confusing the users too much (or worse, having to maintain *patches* to the Mailman code).
- The Mysql user passwords can be changed from inside Mysql itself. Less common case, but it would be good to continue supporting it.

To manage this situations we have some automation operating an "inverse reconciliation": the specific data is read from database only when *creating* a new resource, while it gets periodically synchronized in the opposite direction, from the service to the database.

The important aspect, in these cases, is to acknowledge that the data is present in the users database only to provide informative description, and in *read only* mode: to go back to the mailing lists example (Mailman), having the administrators of a list present in the database does not mean that we cannot change them in the database itself, it instead allows us to show the lists in the user panel, delivering an illusion of a *transparent* service from the end user point of view.

10 Only time will tell

A few years have already passed since the A/I collective first adopted the infrastructure described in this document, so at least a bare minimum of retrospective analysis is already possible. In general we can say that our efforts have sorted out the desired effects: the necessary maintenance level has proven to be low as we had planned, the "forceful" operations (for example the upgrade to a newer stable Debian version) have been completed with independent deadlines and without rush and so on.

The separation of secrets from code has finally allowed us, after many years of only-desiring-to-do-it, to publish *all* the code and our service configuration, and hopefully this has been useful for other groups doing similar things.

We have benefited greatly from the opportunity of testing arbitrary changes on the infrastructure, significantly improving Noblogs, and empowering some of us to develop specific skills in that field.

Slowly we are also noticing how some "professional" technologies are aligning to the scopes that are most interesting to us, so that we will be able to drop several *custom* tools and solutions in favour of free and open-source equivalents: we are finally close to the point in time when we will be able to find a substitute even for *float* itself.

11 Outro


```
"Socializzare saperi, senza fondare poteri." (share knowledge without establishing power)
-- Primo Moroni
```

[1] https://creativecommons.org/licenses/by-nc-sa/4.0