Observations and projections from half a year of using Osuny : “Co-construire un commun numérique” pour “Devenir un commun” (4/5)

yala **juin 2024**

After looking at **how Osuny works** and **how we used it at the IDN**, let’s take a last round and return to **the original hypothesis and question in focus**. We had last reformulated it in question form as:

> Is it possible for a computational commoning collective or Librehoster to run a replica of the Osuny application for the International Degrowth Network and possibly other communities?

In this concluding piece we depart from there to extrapolate some lines of activity and discourse, which in the long-term appear to be beneficial to the overall effort of maintaining Osuny *as* a Common *for* the Commons. Currently a few obstacles were perceived, when trying to find the Common in question. The text below contains proposals to widen the accessibility of its area to allow for easier contribution and reproduction.

> This is the ~~last~~last but one article in a series about **Ergonomy of development experience and common reproducibility questions around maintaining degrowth.net**.
> The previous article is **Osuny case studies: degrowth.net/ + explore.degrowth.net/ (3/5)**.
> The next article is **Observations and projections II (5/5)**.

What is a Common, anyway?

While certainly no single accepted definition of the term « the Commons » can exist, we can acknowledge for existing conventional strains by subscribing to and deviating from them. Starting from the Ostrom-line, we can choose to not just focus on the resource (means of production), but also its community, the processes, rituals and decision making procedures (modes of production).

When the collective performativity of generating the Common through Commoning is taken into the view, we are switching form a substance-based ontology to a process-based ontology. This comes with a shift in epistems (relationability. performative turn. **matrixial narratology.**) and perspectives that may not be entirely compatible with the cooperative money economy. Which helps us see the Common in its relatedness to other economic forces, by distinguishing it by its operational logic. With this shift we refocus on the regulatory principles that reproduce the Common.

To do that, we engage in active discourse and challenge each others assumptions in a productive way that provides constructive critique.

« There is only one next step »

When considering the practicalities involved in evolving and executing the regulatory system of a secure user-facing web application, we lend the term « Collective Responsibility » **from the DevSecOps movement** and **from Practical Philosophy**. Interestingly, both have independently come up with definitions of the term.
The movement goes back to the praxis of **blameless post mortems** at etsy, where the team took an approach of **forward-looking collective responsibility** to not incite shame on people, but to rather look for systemic responses of the collective and its product as a materio-semiotic whole.

Here we go, « failing forward ».

Reducing tight couplings and increasing development accessibility in times of reproducible builds

The application stack of Osuny is grown to many pieces in many places. They all relate to certain interfaces in one another and couple certain systems through them. In different areas, these systems and interfaces have become too accustomed with each other, which blurs their boundaries.

The behaviour of Osuny as a holistic distrbuted web application depends on clearly defined data paths for expected and exceptional user input. The layout of the data paths determines the state the system will capture, process and store. Code paths that increase the resiliency of the system are those, which fail gracefully and especially recover gracefully. Code paths that recover gracefully are those that are only loosely coupled. In a distributed, eventually-consistent system the error mode is an expected runtime state to deal with.

Reducing tight couplings between different parts of the Osuny stack helps to isolate error conditions and aid in eventual recovery.

When developing « on » Osuny is a recognised use case on **developers.osuny.org**, then developing « of » Osuny could be similarly recognised.

The development environment of an application provides Software engineers with the documentation and tooling necessary to build and run instances of the application. The accessibility of the development environment dictates, how quickly people can jump into testing a hypothesis with a quick fix. When people are easily able to test and implement ideas, more contributions can be shared between all and the quality of the Software increases.

« Noesya is currently the only single Osuny vendor, who can provide feature, bug and security releases. From the perspective of the Osuny community, this dependency is a risk to prepare for. If Noesya dropped out of business, the application would cease to exist and existing websites break. Because no prior art for setting up independent instances exists, people can’t do much with their last data exports and are left without the ability to edit their perfectly fine static websites, if they don’t want to learn to write the Osuny data structure themselves, which still lives in the abandoned git, if not turned off already. »

This is an extreme example of how the accessibility of a deterministic build environment helps to directly influence the means of production everyone has available and can thus replicate. The self-propagation of its basic idea is the primary intent of a Common. Or to cite Christopher Alexander:

> "The most important properties which anything can have are those properties that deal with its stability. "
> picked from **Systems generating systems — architectural design theory by Christopher Alexander (1968) – Coevolving Innovations**

Mapping the road ahead

This page gives hypothetical speculations and empirically validatable steps, which invite for experimentation. The ideas below were grown from working with Osuny in the first two quarters of 2024. The intent to present them here is to discuss ways to further strengthen the computational foundations of Osuny:

1. There is a meaningful and technical readme in the website template
2. The Contributing workflow is documented within the repository
3. The amount of third-party dependencies to commercial systems is reduced and the usage of FLOSS replacements and civic hosting is exemplified
4. Replicability and reproducibility are included in the self-understanding of building Osuny, the Common and the application
5. A Software Bill of Materials (SBOM) exists to assess integrity and security
6. A `publiccode.yml` file exists in the repository that documents involvement in public funding
7. A technical developer documentation is available in English
8. The Osuny CLI is decoupled from the website code and theme repositories and is available as an installable package from a package registry
9. The Osuny CI is tested to run on FLOSS platforms and allows for customizations
10. The Osuny website git repository only holds references to its modules and doesn't contain them

---

Remarquer les pas

As a post-structuralist, I am not able to provide anyone with deterministic ontic evidence or forecast of the future, but just with a bulk of relations that describe the field of symbolic relations between the abstractions and material encounters between people. Make of that what you will.

That said, the language chosen for this section is technically-declarative in the headlines, which might sometimes sound like an imperative. It is not. The sentences name empiric evidence that can be checked for, in case a story in question has been addressed and needs to be verified. The method combines two techniques: One, the praxis of formulating acceptance criteria in the technical requirements of a story (accompanied by user stories in the functional requirements written above them) and two, the praxis of backcasting one's own expectations from the future: « What will have happened, so that we got there? » It was actively avoided to begin the phrases with verbs.

Now, after this disclaimer, let's take a few steps through this garden of ideas together. Check for yourself, which of them have a reasonable chance of implementation and which must remain in the icebox, waiting for an ideal future.

1. There is a meaningful and technical readme in the website template

As a developer, when looking at the source code repository of a technical project I am intending to build with, I will clone the repository to my local machine and open its Readme file to watch out for instructions. In its present state, it sends me away to another site. This other place also doesn't have the answers to my questions, because it is made for another purpose.

When searching for `install`, `setup`, `contrib`, `build` or `code source` on the **Plan du site | Osuny**, I get zero results. As a Software developer visiting the page for learning more about the magic behind the scenes, I am now confused and will move along.

It was also a bit disheartening to learn, that I can actually not just run another instance of Osuny without much reverse engineering effort, because its technical documentation is proprietary.

As a build engineer, I can continue to inspect the files that completely describe a running system and learn that I am talking to a Ruby on Rails application, which knows its `mice`, `elephant` and `whale`s. Yet even with this knowledge, I need to look up conventional runbooks for this kind of Software and infer its interactions with the 12 factors.

> **The Twelve-Factor App**
>
> A methodology for building modern, scalable, maintainable software-as-a-service apps.

The file can also describe the CI pipelines and eventual other automations that are integrated with the repository [Beware @osuny-bot!]. Additionally useful information may be how to build, setup and test with a development environment, or how to run a containerised instance with all dependencies.

2. The Contributing workflow is documented within the repository

Another convention that can help navigate the space of guiding an interested individual in learning about the workflows and practices used within a repository is the `CONTRIBUTING.md` file.

When a `CONTRIBUTING.md` is given, it can contain information about how to contact the developers, where to file bugs, how to find the community chat and discussion channels, and also about how to contribute code to the repository. The file can for example explain which branches there are, how they are used and what to do when wanting to contribute

back to the repository.

Interestingly, it is the service-oriented fee schedules, which designate valid roles for contributors.

> Les personnes physiques sont qualifiées de « contributrices » dans les cas suivants :
>
> - ouverture d'issues documentées sur Github
> - contribution à la documentation
> - contribution au design
> - contribution au code
> - contribution aux outils de formation
>
> **Conditions d'utilisation | Osuny**

> **Réciprocités**
>
> Nous proposons d'utiliser OaaS sous le principe de réciprocité. Les personnes physiques peuvent contribuer (ouverture d'issues documentées sur Github, contribution à la documentation, au design, au code, aux outils de formation…) et bénéficient dans ce cadre d'un accès sans contribution financière.
>
> | | |
> |---|---|
> | Personne physique contributrice | gratuit |
>
> Le choix d'un fonctionnement contributif est déclaratif, pour un an. Si la personne ne contribue pas pendant l'année, elle s'engage à revenir au fonctionnement non contributif.
>
> **Osuny as a Service | Osuny**

These lists could be fleshed out into a complete contribution guide, which helps people where to seek advise, where to locate interventions and how to test them.

The Contributing file is also a good place to link to other technical documentation to look up in different cases and to offer different levels of contribution, depending on the developer hat a visitor may carry. Here we may allow us to group those audiences roughly into application- and platform-oriented people.

3. The amount of third-party dependencies to commercial systems is reduced and the usage of FLOSS replacements and civic hosting is exemplified

The dependencies on commercial service offers for the commercially used live instance(s) of the Osuny Common as the logiciel libre it is pose a risk for other Commoning initiatives wanting to use Osuny within the Commons, as persistence of sufficient conditions cannot be guaranteed without working outside the Commons logic.

With the existence of many FLOSS alternatives for those components, it is detrimental for Commoners to realise that this is the only tool they have, if they genuinly want to propose their different economical programme outside the world of the reciprocity principle.

If some services cannot be hosted on one's own, there are always options for federating operations with Librehosters/CHATONS, like Point1 EnCommuns showed. This not only produces more resilient networks with contributing to commons cross financing, but also produces trust by showing the validity of alternative economies.

I'm seeing a lot of effort on **Transparence — Osuny** of the cooperative economy behind the project. I'm also seeing that the **Budget | Osuny** has prefiguration for accounting of different kinds of usages. But the page also reminds me, that I am allowed to use the software « for free », but that I need the commercial offer to get the whole deal.

As I'm seeing EnCommuns on that list and that you are working with DeuxFleurs gives a little confidence, that you are aware of the problematic and already try to « divest » your attention and resources where appropriate. And if a single Librehoster/CHATONS cannot help you to validate your use case, maybe multiple of them together unisono could?

Maybe this is also a chance to go through the stack and see, which components could be marked redundant, for example for smaller scale deployments. Why are there so many CDNs, for example? I'm counting three: Website, Images, Assets.

Word has it, that assets are no longer in a good place on third-party domains, if one wanted to provide strict CSP rules for increasing an application deployment's security.

> **Public CDNs Are Useless and Dangerous**
>
> Once upon a time, loading common scripts & styles from a public CDN like cdnjs or Google's Hosted Libraries was a 'best practice' - a great way to instantly...

You are always free to host your own dependencies, as you are effectively vendoring them, to bring your own caching layer and to benefit from multiple parallel connections in the same HTTP/2+ stream.

4. Replicability and reproducibility are included in the self-understanding of building Osuny, the Common and the application

Osuny is a web application that follows distributed systems engineering principles by implementing an event-driven microservice architecture based on multiple dependent components. The data flow is basically unidirectional and represents a pipline that pictorially respresents an editorial workflow of a website.

As a Software engineer, it has shown practical in the past for development, deployment and (unit and integration) testing purposes to make use of deterministic runtime environments, which are used to execute reproducible builds of the application. The runtime environments are expressed in a declarative language, for example a configuration file. The builds are distributed as immutable artifacts.

As a platform engineer, who deploys FLOSS on a regular basis, I need to have access to immutable build artifacts and declarative runtime environments, in order to replicate instances of a given Software easily.

To be able to build and run all components of the stack on my own, they must be Free Software. This comes with the need for not relying on proprietary components in the stack, which can potentially be replaced with libre alternatives.

Similar as people engaged in reproducible data science, scheduling a long-running job for a computational experiment on a deployment system shares many properties of an executable data pipeline. The only practical difference is, that it is not expected to exit.

1. Reproducible builds
2. Immutable build artifacts (containers, packages)
3. Declarative runtime environment (containers, Nix)

Providing these components to the development environment can increase the confidence in the work, because things fail much more expectably and build artifacts are the same between runtime environments. An early experiment could be to test building an Osuny page from within a reproducible **Development Container**.

For the Osuny Common to be replicable, it is necessary for its technical components to be

reproducible.

5. A Software Bill of Materials (SBOM) exists to assess integrity and security

The application builds contain individual pieces, either from self-written code or from vendored package dependencies. When listed across implementations, these pieces provide a global overview of in-house and public components in the stack. Such a list is called a Software Bill of Materials (SBOM).

It is often expressed in a vocabulary suitable for this kind of metadata. I had written about the subject during research for the new eco:bytes website, and would like to invite you to skim the piece:

> **Licensing the whole stack of an application under a Free and Open Source License**
>
> When building Free Software in the computational common, it is important to know the materials used for building the Software. The moment I add a dependency to my project, it becomes my responsibility to care and cater for it. This procedure is...

When this component catalogue exists, itself also expressed in a declarative language, its presence is also benificial for determining insecure packages and updating them more timely. You may know similar kinds of security scanning from CI workflows. They upload their results as CI artifacts for subsequent processing.

Creating an SBOM from immutable, eventually signed artifacts generated through a reproducible build pipeline has the benefit of giving the guarantee that all instances of the same version of the artifact will share the same dependency tree, no matter where they are deployed. This allows us to draw substantiated conclusions about the attack service of a given version of the application, just from looking at its metadata.

6. A `publiccode.yml` file exists in the repository that documents involvement in public funding

There is more metadata which can be useful to put inside or alongside the repository. Looking at the record of a substantial contribution from public funding sources in the document of the strategy commitee reminded me of `publiccode.yml`, which is recently

making its rounds in European FLOSS projects.

It is a metadata Standard, which allows to identify and describe publicly-funded, freely-licensed open source projects and has grown from the **Public Money, Public Code** initiative

The public `publiccode.yml` file is a lightweight pattern, which aids accountability and discovery of artifacts from progressive public funding agencies and also increases their confidence in working with FLOSS.

Osuny could be a nice example for carrying this voluntary label to document these kinds of human affairs in machine readable form.

7. A technical developer documentation is available in English

When it comes to documenting technicalities of a project, nothing beats a technical service manual which explains *developing*, *building*, *running*, *testing*, *debugging*, *installing* and *operating* a code base in question. Even more so, when it also contains a technical reference of the API surface of its models and methods, human- and machine-readable schemas, e.g. in form of an Entity-Relationship-Diagram (ERD) and an OpenAPI definition of the routes present in `admin`, example deployment diagrams or root administration tutorials.

People who are new to the project will have *many* questions. And having answers readily available in written form and openly accessible increases the likeliness of a contribution from a stranger.

Documenting the API service also allows for easier modification and extension of the code base, because the interworkings of the Abstract Syntax Tree don't need to be manually reparsed in the head, when reverse engineering their meanings. This is a lot of effort and requires training and time, which many people don't have or spend.

I have seen good documentation pages built on the Diàtaxis framework. Maybe it can be useful to learn from them and to structure the documentation in several ways, in so people find multiple entry points, depending on what they are looking for?

Diátaxis

The Diátaxis framework solves a problem of quality in technical documentation, describing an information architecture that makes it easier to create, maintain and use.

8. The Osuny CLI is decoupled from the website code and theme repositories and is available as an installable package from a package registry

Allow me to paste this paragraph from a message of mine in the email conversation which precluded this series of articles, because it's hilariously nerdy:

> Regarding your approach the the CLI, I am probably the wrong person to evaluate which languages to add. As a systems integrator, I like to use existing tools which provide simple, declarative syntax and which are considered « state of the art » in the industry. Using generic tools like Just is just (sic) one example. From a programmer standpoint, I know that it is always exciting to try and use new languages. As a FLOSS and Linux user, I generally prefer what is made available via **GitHub - asdf-vm/asdf-plugins: Convenience shortname repository for asdf community plugins**
>
> For the Osuny binary, I'm not sure if adding Go to the choice of languages, like Ruby and HTML, CSS (, JS) increases maintainability and consistency a lot. For the users it should make no difference, but it already brings up the question of packaging and distribution. Not everyone has Go installed and most people will install Hugo via its Binary, in so that the language of choice is transparent to them. I think it's good to choose a language one feels comfortable with.
>
> About the implementation of the Osuny binary (wrapper), I have few comments:
>
> 1. The build.go file uses exec.Command, while the serve.go, update.go and upgrade.go use the Shell wrapper from root.go, which seems a bit incongruent.
> 2. To me wrapping external shell commands in a non-generic way appears to be obfuscating what the program actually does. Reading a list of commands in a justfile or in a shell script is much more convenient than having to decode the nested source code. Also using Go with its language features, like its type system and the binary compilation, seems a bit like overkill to create a the wrapper. But that may be personal preference. As NPX is used in the scriptprogram, why not continue to build on JS, then? Publishing the modules on NPM would make it easier to use the individual components by third parties, since it doesn't add another language and compiler dependency. I've also seen instances, where people publish Go binaries on NPM, but that's another story. What I like about the wrapper is, that it leverages Cobra to bring helpful description texts to the user/developer/person using it.

> 3. When looking at update.go, I'm wondering about the choice of git commands. These two **osuny/cmd/update.go at db1467a6726f9b492bbb76139790c16e1f8722f8 · osunyorg/osuny · GitHub** could probably be replaced with ‹ git submodule update --recursive ›? Since there are git language bindings available for git, I would even consider to use those instead, which could allow for more advanced use cases and also be a justification to compile a static binary, which brings additional features over scripts or task runners. **GitHub - go-git/go-git: A highly extensible Git implementation in pure Go.** Right now, the application doesn't ingest or expose any data on its own, but relies on third party applications to mutate the state of an instance, which seconds my impression of using a statically typed language for this use case.

Let's take it from there.

> First **Ergonomy of development experience and common reproducibility questions around maintaining degrowth.net**.
> Previous **Osuny case studies: degrowth.net/ + explore.degrowth.net/ (3/5)**.
> Next **Observations and projections II (5/5)**.

- **Observations and projections II (5/5)**
- **Ergonomy of development experience and common reproducibility questions arou…**
- **Osuny case studies: degrowth.net/ + explore.degrowth.net/ (3/5)**