Blueprint for I3S

Table of Contents

- Table of Contents
  - Background
    - Why we need a different infrastructure platform
  - Providing services
    - Scaling capabilities for services
    - Containerization
    - Environment
    - Cloud
    - Considerations
  - Developing services
    - Why Open Source matters
    - Open Source makes your software better
    - Law & Order
    - Considerations
  - Overall security and logging
    - Zero Trust and Service Mesh
    - User administration, authentication and authorization
    - Logging and Monitoring
    - Automated Configuration Management
    - Strategies for sealing and managing secrets
    - Scanning code / Source code analysis
    - Utilizing GitHub and DependaBot
    - Scanning containers
    - Considerations
  - Concluding remarks

Background

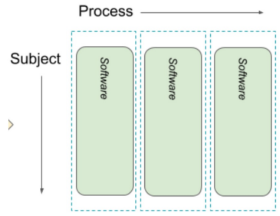
Based on the deliverable related to the definition of integration and architecture guidelines in WP2 and using modern application architecture patterns we want to create a blueprint. The blueprint is describing a reference runtime environment for modern, sharable services following CSPA standards/principles using containers. However, it will not give any guidance on how to develop shared services, as the focus will be on the runtime environment for the services. The deliverable will describe the basic infrastructure needed and implement a cloud instance for the needs of the ESSnet. The infrastructure will be documented as code, which will give to its users the opportunity to version it and fork it. Typical products implementing this pattern would be Ansible Playbook or Terraform. One advantage with the “infrastructure as a code” model approach is that it enables NSI’s to easily create their own modern infrastructure on their premises. As part of the deliverable, we will provide a simple container-based platform using a cloud infrastructure, which will allow us to validate the blueprint and to perform functional tests on the services developed as deliverables in WP1 - Develop new services. This also enables the service developers to validate their packaging and installation.

Retro-fitted, and modularized existing services will also be tested on the platform, either on premise or on the public cloud instance. This work package will also discuss components related security related to authentication and authorization, securing platform, services and code. The blueprint also look into how these types of security components can be added to existing services. Containerization and orchestration technologies, including Kubernetes and Docker, will be the basis of the platform, and all other infrastructure components will be built with it or around it.

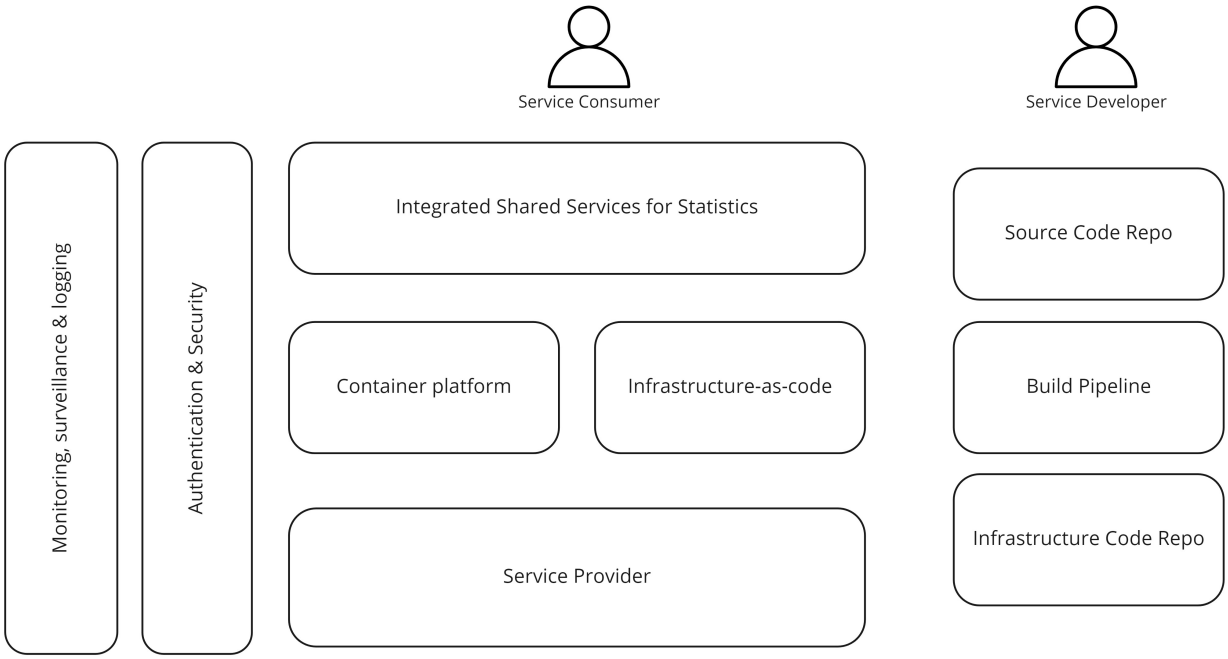
Why we need a different infrastructure platform

Traditional infrastructure is rigid, costly and not suited for supporting the rapid change in technology and agile processes. Even with the advent of virtualization, and the ability to run hyper convergent infrastructure on premise, we tend to hit struggle with high complexity of our infrastructure, and high management cost of infrastructure. With high complexity, managing adequate security is also an issue. Containers hide some of this complexity, especially when it comes to managing software compatibility between software projects. Implementing a cloud infrastructure help us manage underlying infrastructure complexity by using managed infrastructure that can scale depending on the need of the organization.

Goals
Agile business processes
Reuse and share services



From Subject perspective to process perspective (D2.1)



To enable agile business processes, and support the implementation of reusable and shared services The blueprint is describing an approach and patterns in order to support the implementation of reusable and shared services, and the capability of enabling agile processes for developing, implementing and govern business services.

For enabling shared services on a cloud platform, the following building blocks should be considered, and is a part of the implementation of the blueprint:

- Service provider - in the context of I3S the service provider delivers Infrastructure as a Service (IaaS) as the base platform for implementing business services and applications.
- Infrastructure-as-code - the capability to manage and provisioning the platform as machine-readable definitions and should be managed through a version control system.
- Container platform - is the logical package mechanism for running services and applications, decoupled from the environment or platform it is running on.

In addition to the above mentioned building blocks, one need to establish the capability to enable security mechanisms, these are related to securing the platform, containers as well as services and applications. The monitoring, surveillance and logging building block is related to managing and govern the platform and services running on the platform. Considerations should be related to cost, scaling and general governance of the components.

Service developers build services and applications to be implemented on a platform. These services and applications are deployed through build pipelines, enabling "Integrated Share Services for Statistics" to be exposed and consumed by service consumers. Source repos and infrastructure as code repos are used to keep track of changes and share code across projects and teams.

These building blocks and components are elaborated, described and rationalized further in the next chapters, and proven and validated in the deliverable "Implementing the blueprint".

Sharing services can be more than just running another NSI's service, it can be sharing and re-use in the context of sharing and reusing code. It is sometimes hard to make a service that will satisfy all needs. It is important to think of the scale and scope of the service, and the amount of business logic it provides. Sharing through code libaries is also something that should be encouraged. This will provide you with the possibility of sharing functionality as a function or statistical method in a code environment like Jupyter Notebooks.

Scaling capabilities for services

The design of a service greatly impacts the ability for a service to scale. One of the aspects of a microservice architecture is the ability to scale. When talking about scalability we often talk about horizontal and vertical scaling. Vertical scaling is the traditional way of scaling a service or application; Add more physical/virtual hardware dedicated to the service. F.ex. memory or CPU.

Horizontal scaling means the ability to duplicate the service to increase f.ex. response time, throughput, and processing. This requires easy duplication of services. This is where containers are important. The duplication of services requires a way of treating those services in a disposable way. Destroying one instance of a service as part of hundreds wont impact your system. One of the caveats for scaling horizontally is related to your downstream dependencies, like storage, session handling etc. The scalability of your service is hampered if you have downstream dependency on a single instance relational database. It is extremely important when designing services that have downstream dependencies to also take into account your scaling pattern. Public cloud vendors provide you with scalable infrastructure services that alleviate this. Sessions and state should also not be handled in a way that hampers your scalability. Make sure sessions are not tied to ephemeral storage and have the ability to be instantly synchronized across services.

Your infrastructure should be able to handle the load-balancing, and orchestration of services (spinning containers up and down depending on your needs).

References:

- Dependencies and Data Sharing

Containerization

There are several container initiatives, but the one that has been there longest, and have the largest adoption is Docker. The container format is being standardized as part of the OCI (Open Container Initiative). Containerization is basically a way of creating a small virtual computer, that contains only the virtualized hardware required for the application you want to run, so it works as a way of transporting services without your application having to know anything about the environment around it. This is also its greatest challenge. In WP2 there will be described a lot of architectural guidelines for how you should design your application to make it scalable, and secure, so this document will only reference that work. A container is basically a virtual machine, but with as little or as much as you need to be able to run your application.

Environment

You can run Docker either on a Windows machine, or a Linux/Mac. Even .Net applications in containers are moving towards running on Linux host-systems (from .Net Core), so for minimal pain, you should set up your docker environment on a Linux machine, or in a virtual machine running Linux. Using Windows host-system in Docker is not recommended, as it's inefficient and not very standardized.

We provide pre-designed virtual environments which have Docker pre-installed which can be used to set up your environment on premises.

For test purposes it should be sufficient to use a standard Docker installation for getting things up and running. For production quality runtime environments for containers and for container orchestration we recommend looking into products like Apache Mesos or RedHat OpenShift.

Cloud

- Google Cloud Kubernetes
- Azure Kubernetes Service
- Amazon EC2, Amason EKS.

There is a plethora of other services that will ease the use of these public cloud vendors, like Pivotal's Cloud Foundry, which will give you "serverless" functionality that can run on any of the large public cloud vendors.

Considerations

For starting to build services that you want to containerize, you will need a machine that can run Docker as a minimum, or a virtual machine that runs docker. There are also several online options for running containerized services. They greatly vary in price and functionality but can be used as a test for running simple services in the cloud (or on premise).

In general, it’s hard to establish and maintain an on-premises, container platform from scratch, depending on your organization’s maturity. But there are several good on premise platform-products that will help you

with things like security and hardware provisioning, like Apache Mesos, and RedHat OpenShift.

Developing services

Why Open Source matters

The use of open source provides greater freedom to choose both products and technology that are right for your organization. It makes it easier to choose as little or as much as one need at any given time. It also gives your organization the freedom to choose vendors and expertise in the market.

Sharing code nationally and internationally improve quality and encourage reuse of code across organizations and borders.

- Culture, and specifically sharing culture contributes to speed
- Open Source contributes to innovation
- Open Source is more than reusing code GitHub
- Focus on internal and external sharing
- Open rather than closed
- Transparency promotes quality
- Transparency promotes security

Open Source makes your software better

- TCO - Total Cost of Ownership
- Change cost
- Reusability
- Competence and recruitment (the future)
- Showing your work (transparencies, governmental services), including public trust
- Security

We strongly recommend NSO's to adopt an Open Source strategy when developing services.

Law & Order

Choosing the right license for your project is important. It will have implications, both to how your code can be shared with individuals, organizations and corporations, but also what kind of re-use you can give permission to. Roughly speaking there are two main types of Open Source licenses that can be applied to your code. There are variations

1. Viral Open Source license
2. Permissive Open Source license

Viral Open Source license

A license that requires the use of the code to retain the same license as the code with which the new code is derived from. This ensures that the code you license under such terms, cannot be used in a derivative work that is closed sourced. This seems like a good thing (and in many cases it can be), but it limits the re-use of your code for organizations that have strict rules as to what type of Open Source projects they are allowed to re-use.

Examples:

- GPL/LGPL
- EUPL

Permissive Open Source license

As the name implies, this license will have some, or no requirements for re-use, or derivative work. Most of these licenses just requires attribution but does not restrict your use of the code. This is often favorable, as it requires very little effort to assess the legality of creating derivative work but can in the extreme result in the use of your organizations code in a closed sourced/proprietary code base.

Examples:

- Apache License 2.0
- BSD License
- MIT License

Considerations

- Use a tool like JLA(Joinup Licencing Assistant) allowing everyone to compare and select open licences based on their content.
- The right competence to choose the right products for the right purposes, in addition to basic knowledge of open source licensing.
- In order to share code internationally, English should be used as the development language.
- Make sure that code is decomposed so that specific business logic is separated from the code that may be of interest to the general public.
- At all times, make sure that you have the right to the code being developed in-house and/or that the source code is shared under an open license.
- There may be reasons why one chooses products/solutions that are not open source, but then quality, scope and degree to which the product provides for strong links to the architecture must be assessed and documented.

Overall security and logging

The description provides a brief documentation with an overview of relevant concepts to support a security model for establishing services in cloud environments. The documentation is based on the description of the security model in Statistics Norway (SSB Developer Guide) and documentation of deliverables in WP1, WP2 and WP3 in I3S. The deliverable "Implementing the blueprint" describes the security components used in the implementation of the I3S platform. A fundamental principle is security (CSPA064): *"Maintain community trust and information security. Conduct all levels of business in a manner which build the community's trust. This includes the community's trust and confidence in the statistical organization's decision making and practices and their ability to preserve the integrity, quality, security and confidentiality of the information provided."*

Exposing and sharing services between organizations and users need to be built on trust. Security issues are amongst other related service security, confidentiality and user administration in order to establish capabilities for policy management and service governance. The document "Guidelines for describing statistical service definitions" discusses different levels of service exposure. For the deliverables in I3S one

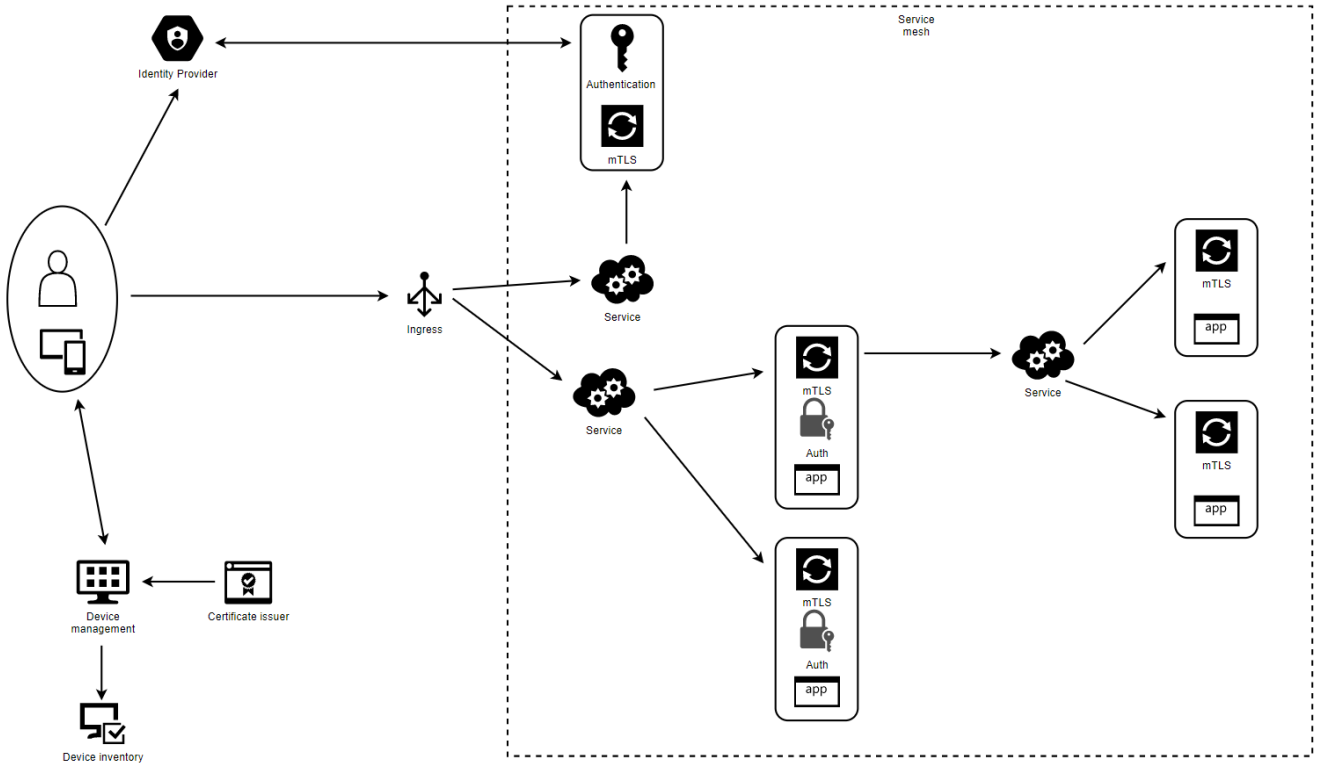
should consider service security at an Inter-institutional and Public level, in addition to security considerations one need to consider service lifecycle management, governance of cloud platform etc.

Zero Trust and Service Mesh

Zero Trust, Zero Trust Network, or Zero Trust Architecture refer to security concepts and threat models that does not assume that actors, systems or services operating from within the security perimeter should be automatically trusted. Instead, it requires that services must verify explicitly, use least privileged access and assume breaches before granting access to a service or application. The zero trust concept would be a preferred concept for securing shared services, and especially for services shared across institutions and borders. Considerations when implementing a zero-trust concept for your services should be related to identification of who the users are and to what their actions should be during the session. Other issues would be related to traffic visibility and context for monitoring and verification of traffic across users, device, location and applications.

Service mesh is often considered as an important infrastructure component for facilitating Zero Trust in a micro service architecture.

In software architecture, a service mesh is a dedicated infrastructure layer for facilitating service-to-service communications between micro services. Having such a dedicated communication layer can provide a number of benefits, such as providing observability into communications, providing secure connections, or automating retries and backoff for failed requests.



The figure shows an example on technical implementation of the zero-trust concept using service mesh to deliver functionality for authentication and authorization of network flow and access proxy for user and device authentication.

References:

- BeyondCorp - Google's implementation of the zero trust model
- What is Istio?

User administration, authentication and authorization

Authentication is the process of verifying a user's identity. Authentication should be based on open standards for instance OpenID connect (OIDC) and OAuth2 for authorization. When implementing an IdP-solution one should consider the OIDC compatibility, multi-factor authentication, self-serviced password reset, real-time logging, password-policy features.

Logging and Monitoring

Logging is important in any security model for auditing and forensics.

Log scraping should be done for alle Applications. It is especially important that access related information is logged by applications to assist with auditing.

References:

- OWASP Logging Guide
- Logging Cheat Sheet

Automated Configuration Management

The role of automated configuration management is to maintain systems in a desired state in order to reduce cost, complexity and errors. This is especially important in a Zero Trust architecture where configuration transparency, traceability and consistency in a system is essential for security.

Terraform, Ansible and declarative manifests for describing system and application state and GIT for change management and traceability is often used.

Strategies for sealing and managing secrets

Secrets like passwords, certificates and keys are probably the hardest assets to manage in any system, but also the most important asset. Leaked keys can lead to unauthorized access to sensitive data and have severe consequences for the organizations trust, reputation and reliability.

Examples of tools is Secret Manager, Berglas and Sealed secrets.

References:

- OWASP Key Management Cheat Sheet
- TechBeacon: Top resources for cloud native secrets management

Scanning code / Source code analysis

Automatically scanning your code is a good practice. There are many scanners to choose from and a lot of them are free for open source code. There are also different kinds of scanner and some of them covers several categories.

- Code quality
- Code review

- Dependency management
- Security

References:

- Depandabot (Github)
- Snyk
- SonarQube
- Code Climate
- Codecov

Utilizing GitHub and DependaBot

DependaBot is a service that was recently acquired by GitHub for scanning code repos in different languages for known vulnerabilities. Depending on your configuration DependaBot will also create Pull Requests that will fix security-issues related to vulnerable dependencies in your code. It's an easy feature to turn on, and it will start to create reports on your repository.

Scanning containers

Scanning containers is similar to scanning for vulnerabilities in dependencies in code. Several services will scan containers for underlying image dependencies that have security issues

Considerations

- Establish confidentiality actions in order to establish the right mechanisms for protecting information. Encryption at-rest and in-transit, securing personal and organizational information, disclosure controls.
- Consider security concepts for cloud services, like the zero-trust model.
- Consider authentication requirements for service discovery and service delivery. Be aware of the challenges related to trust-models for chained services in a statistical production line, as shared services could be based on different security models that need to be handled
- Risk assessment, mitigate the risk that a CSPA Service or the data it controls is misused.

Concluding remarks

This document describes the basic infrastructure need for implementing a cloud platform for the ESSnet, using modern application architecture patterns for building a platform for enabling "Integrated Shared Services for Statistics" following the CSPA standards and principles and the Integration and architecture guidelines delivered in WP2.

In the Implementation of blueprint, we demonstrate how the infrastructure is documented as code, enabling NSIs to easily create their own modern infrastructure. The implementation shows the implementation of a container-based platform using cloud infrastructure and is validated and proved as I3S has deployed the services developed in WP1. This is also validated and proved as we have "Retro-fitted" and modularized PXweb and deployed it on the platform.