Containerization:
syngo Authorization Server

Commented [S41]: General feedback: please try to better format the document to see the structure better

I. Introduction

Goal of the whitepaper

The goal of this whitepaper is to present the results of the prototype project of containerizing the **syngo AuthorizationServer (Foundations-AuthSrv option)** component (Feature 4599893: FEAT_FT17_CONTNR: Prototype AuthorizationServer containerized delivery): to show what modifications are needed to be made on the current AuthSrv implementation to enable containerization, and to outline how the AuthSrv container image can be tailored down to meet the stakeholders' needs.

Currently the only stakeholder is the Falcon project.

Containerization in a nutshell

Basics

A container is an isolated, lightweight silo for running an application on the host operating system. Containers build on top of the host operating system's kernel, only having limited access to it.

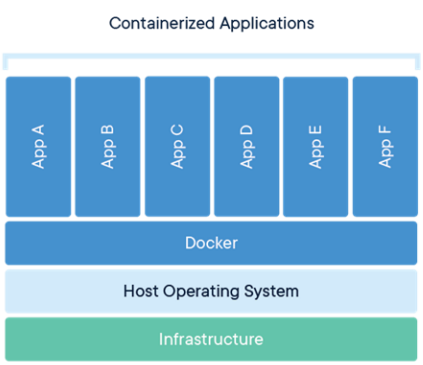
Commented [H42]: Van egy rövid új „bekezdés" ötletem, amit még érdemes lehet leírni, az pedig az, hogy hogyan kell összeállítani magadnak a fejlesztői gépeden a környezetet, hogy dolgozhass docker image-ekkel. Ilyenekre gondolok, hogy fel kell-e tenni valami programot (pl. azt a **docker.exe**-t? Az a GUI-s tool tudom, hogy nem kötelező, de van-e valami **előkövetelmény applikáció**?), vagy valami **Windows service**, amit esetleg be kell-e kapcsolni, hogy minden működjön, stb.
Ugyanebbe a „környezet összeállítása" bekezdésbe esetleg mehetnének ilyen praktikus tippek, hogy mittudomén **VPN**-ről néha szarakszik (ilyen „demo effect"-ekre emlékszem), vagy hogy a **proxy**-t mire kell beállítani, stb.

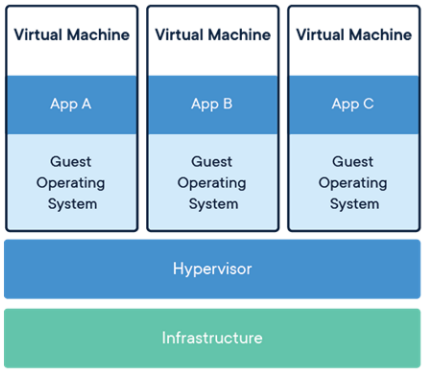
Commented [B43R2]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [B44R2]: Install Docker Desktop on Windows section.

Commented [H45R2]: Please update the "table of contents" section as well because it is a new section.

Commented [GC6]: Don't specialize it to Falcon. Rather write "Currently known requirements". At some other place (or in the same section, but different paragraph) mention Falcon as the first consumer candidate.





The main difference between virtual machines and containers is that virtual machines use a complete operating system with its kernel on top of the infrastructure, whereas Docker containers build upon the host OS' kernel. Containers are more lightweight while having the benefits of providing similar levels of isolation to virtual machines.

Since container images build upon the host's kernel, they are platform dependent. Docker containers can be either Linux- or Windows based.

Pros and cons compared to native applications

- Pros
  - Isolation, Encapsulation
    - Improved security
    - Easier deployment and installation
    - Portability
    - Easier debugging capabilities
    - Scalability
    - Popular technology
- Cons
  - Possible performance disadvantage

Commented [GC7]: No need this part as it is in the title of the section

Containers vs images:
"A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.
Container images become containers at runtime and in the case of Docker containers - images become containers when they run on Docker Engine." src

Base images:
"A container builds on top of the kernel, but the kernel doesn't provide all of the APIs and services an app needs to run–most of these are provided by system files (libraries) that run above the kernel in user mode. Because a container is isolated from the host's user mode environment, the container

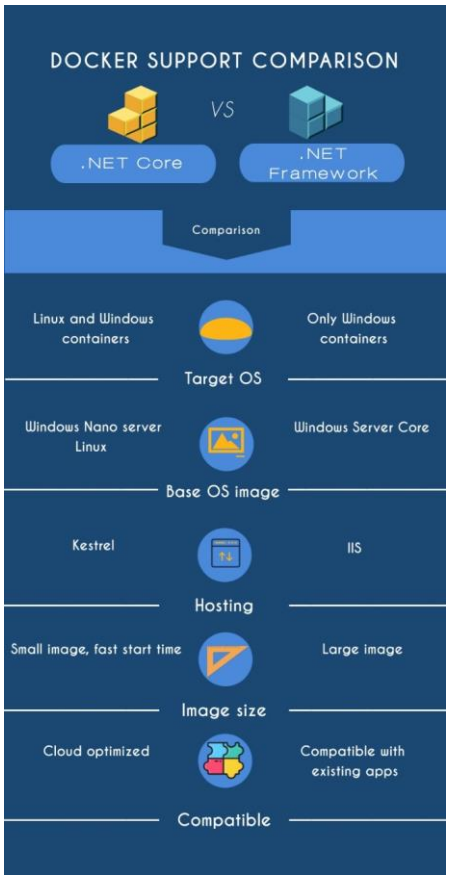
needs its own copy of these user mode system files, which are packaged into something known as a base image. The base image serves as the foundational layer upon which your container is built, providing it with operating system services not provided by the kernel." src

Because of the above, different applications require different base images depending on to what extent they try to access the host's kernel, what APIs are required for them to function.

Orchestration:
Orchestrator software is used to manage several containers over several hosts. They function as service controllers. The most widely used orchestrator is Kubernetes, which is also the chosen solution for the Falcon project. This whitepaper does not cover use cases where orchestrators are used.

Differences between .NET Core / .NET Framework in docker

.Net Framework and .Net Core applications may be containerized differently because of the distinctions between the two frameworks. .Net Core is cross-platform, it has native support in Linux as well as Windows operating systems, therefore either Windows-based or Linux-based container images may be built for it, whereas .Net Framework only supports Windows-based containers.



Container base images for ASP.Net Core: link

Install Docker Desktop on Windows

To prepare your developer workspace for working with Docker containers please see the installation instructions here. In order to build an image in the local machine, the mentioned Docker Desktop needs to be installed.

In case the image needs to access the syngo NugetFeed then, the build will fail.

The workaround: must provide a username/password that has access to the repository, in the NuGet.config. Also, the proxy settings could be necessary for the docker. (e.g. http://pxBUD.evosoft.com:8080). Please check the shelveset, which contains all required changes.

If the image uses pre-compiled assemblies, then no additional changes are needed.

Docker networking

Commented [S48]: Great to have this chapter

Overview link

Docker's networking subsystem is pluggable, using drivers. By default, it uses the bridge network driver, which provides a communication interface between containers within the same host computer.

Default: bridge network

"In terms of Docker, a bridge network uses a software bridge which allows containers connected to the same bridge network to communicate, while providing isolation from containers which are not connected to that bridge network. The Docker bridge driver automatically installs rules in the host machine so that containers on different bridge networks cannot communicate directly with each other."

If not specified otherwise, a new container created on a host belongs to the 'bridge' network. It is possible to create new networks and assign containers upon creation to them.
Example:
docker network create mynetwork
docker run --network mynetwork

Create networks over several hosts (Kubernetes vs overlay driver)

Docker also offers an 'overlay' driver to connect multiple Docker daemons on different hosts, using its Swarm orchestrator. However, Kubernetes is the chosen orchestration solution of the stakeholder, so Kubernetes networking is applicable.
The differences between Docker and Kubernetes networking are described here:

"The Docker networking model relies, by default, on a virtual bridge network called Docker0. It is a per-host private network where containers get attached (and thus can reach each other) and allocated a private IP address. This means containers running on different machines are not able to communicate with each other (as they are attached to different hosts' networks). In order to communicate across nodes with Docker, we have to map host ports to container ports and proxy the traffic. In this scenario, it's up to the Docker operator to avoid port clashes between containers.

The Kubernetes networking model, on the other hand, natively supports multi-host networking in which pods are able to communicate with each other by default, regardless of which host they live in. Kubernetes does not provide an implementation of this model by default, rather it relies on third-party tools that comply with the following requirements: all containers are able to communicate with each other without NAT; nodes are able to communicate with containers without NAT; and a container's IP address is the same from inside and outside the container.

Kubernetes follows an "IP-per-pod" model where each pod gets assigned an IP address and all containers in a single pod share the same network namespaces and IP address. Containers in the same pod can therefore reach each other's ports via localhost:<port>. However, it is not recommended to communicate directly with a pod via its IP address due to pod's volatility (a pod can be killed and replaced at any moment). Instead, use a Kubernetes service which represents a

group of pods acting as a single entity to the outside. Services get allocated their own IP address in the cluster and provide a reliable entry point."

Port publishing

Documentation: src

"By default, when you create or run a container using docker create or docker run, it does not publish any of its ports to the outside world. To make a port available to services outside of Docker, or to Docker containers which are not connected to the container's network, use the --publish or -p flag. This creates a firewall rule which maps a container port to a port on the Docker host to the outside world."

Docker Compose

Source: link

"Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration."

With Compose, it is possible to define very complex configurations as well as simple ones, using yml config files, featuring image building, defining or starting containers, host resource balancing, networking, monitoring, defining restart policies, updating its managed services, running healthchecks, etc.

For a full list of features, see the compose file reference.

Compatibility with Kubernetes

Docker Compose on Kubernetes

Compose configurations cannot be used directly in a Kubernetes environment. Configurations can be converted with tools, such as Kompose etc.

Docker security in Windows

Users

Containers on a Windows host start in the name of the user who has started the Docker Desktop application.

By default, within a Docker container, the current user is based on what base image is chosen for our image. As the Windows documentation states, "The identity of the process running in the container ("ContainerAdministrator" on Windows Server Core and "ContainerUser" on Nano Server containers, by default) will be used to access to the files and directories in the mounted volume instead of LocalSystem, and will need to be granted access to use the data." The identity is ContainerUser the

same for the dotnet/aspnet base image as well, the base image that is required for AuthSrv to run. ContainerUser is not privileged.

This identity can be overridden by the dockerfile when the image is created. It is recommended to set as little privileges as possible.

For more on Docker security in Windows – which is a huge topic in itself and is not in the scope of the prototype to uncover –, start here.

AuthSrv background

- AuthSrv is an ASP.Net Core WebAPI project, an OpenID provider.
- AuthSrv requires a certificate because it only accepts HTTPS connections.
- the default authentication provider used by the platform is a Windows-based solution, using the operating system's user management.
- Customizable elements:
  - JSON-based local configuration (clients, resources, etc.)
  - Certificate for using HTTPS
  - Support for using provider extensions, e.g., custom authorization or auditing providers.
- CI/CD pipeline based on non-containerized deliveries is in place and cannot be discarded.
- Information persisted by AuthSrv shall be available outside the container.
  - logs
  - operational data (e.g., refresh tokens)
  - configuration data

External requirements, constraints

- Kubernetes already is the chosen container orchestration solution.
  - This aspect is not affecting the prototype currently. The implementation and how it is containerized is orchestration-type agnostic.
- there is an interest in using Linux-based containers.
  - While the architecture is based on ASP.Net Core, which may run on Linux as well, thus that alone could allow the usage of Linux-based containerization, the default authentication provider is relying on Windows-based authentication of the incoming requests. To switch to Linux-based containerization, this provider has to be replaced.
- some/all providers of AuthSrv will be overridden, new extensions may be added.
  - The prototype needs to offer alternatives to dynamically inject dependencies to the AuthSrv implementation, so that e.g., a platform provided base AuthSrv container image could be extended by the stakeholders.

Commented [GC9]: s.via, or other DS/Product also could be a Consumer. Don't express Falcon, rather "Currently known reqs.,constraints"

Commented [GC10]: Describe what these would mean to our Container image as requirement

II. Containerizing the AuthSrv component

Necessary changes to the AuthSrv implementation

- **Accessing Certificate**: for its HTTPS communication, AuthSrv reads the certificate from the host operating system's certificate store. This store is inaccessible by containers. Thus, AuthSrv needs to be modified to be able to read the certificate from a filesystem path. The certificate needs to be copied into the container. Suggested method:
    - mount into the container as a volume when creating the container with <**docker run**>. Example: docker run -it -v <cert_location_on_host>:<cert_dest_in_container>,ro
        - Pros: flexible, the same image can be used with different certificates easily
        - Cons: yet another option to be set with the docker run command

- **Adapting the default Windows-based authentication of REST requests**: the default authentication provider is using Windows-based authentication, where it tries to access host OS' user repository. However, a container image has no access to this repository. Suggestions:
    - create the users inside the container manually.
        - pros: the default authentication provider may work this way without changes.
        - cons: requires additional configuration.
    - use a custom authentication / replace the default provider within the container.
    - use an external service for authentication e.g., in another container.
    - use domain service accounts: Microsoft documentation

      The user creation option is cumbersome to maintain and is not recommended. Falcon project is not interested in domain service accounts, so the domain service account is not viable either. This whitepaper is not decided whether a new authentication provider or an external authentication service is the better option for REST authentication. This point is marked as a follow-up step for this prototype project.

- **Allowing dynamic dependency injection:** providers e.g., Authentication and Auditing, shall be replaceable. The implementation needs to be able to recognize and load newly added providers during the AuthSrv server startup. Please see the next section below for more details.

Commented [S411]: There are other binary extensions then the authentication one... e.g. the additional data to the response, persistency, etc. thesestatements are generally applicable for those ones also?

Commented [C412]: Maybe it's a dummy question, but under what account on the Host OS the docker containers are running? What is our recommendation for AuthSrv?

Commented [B413R12]: @Kungli, Marton (ADV D EU HU OPS 4 4) User management

Commented [K414R12]: im trying to answer this in a new chapter, "Docker security in Windows"

Commented [GC15]: By the way, this might not be a problem, as Falcon is using share authentication provider

Commented [S416]: Which is suggested?

Commented [B417R16]: @Kungli, Marton (ADV D EU HU OPS 4 4) Write summary.

Commented [GC18]: Suggested solution? Please put a reference here.

Commented [S419R18]: +1

Dependency Injection

If AuthSrv will be shipped as a docker image, then we need to provide a way to extend our Host container services to consume provider implementations (e.g. INamePasswordAuthenticationProvider).

Current State

Currently, the only way to extend our AuthSrv library to make an own host environment and inject service implementation at startup. In a way, the consumer needs to get our library as a NuGet package and wrap it around with an own hosting solution. Eventually, we have a hosting solution, but not extendable. It is using our default implementation of services.

Hosting startup assemblies in ASP.NET Core

The ASP.NET core platform has a built-in solution for runtime dependency injection.

An IHostingStartup (hosting startup) implementation adds enhancements to an app at startup from an external assembly. For example, an external library can use a hosting startup implementation to provide additional configuration providers or services to an app.

If the customer wants to overwrite our default services implementation such as INamePasswordAuthenticationProvider, then they can take advantage of the IHostingStartup feature, which is provided by .NET Core.

What steps are need for supporting this approach in AuthSrv?

Considerations for writing custom services:

1. Firstly, the injected services need to be placed into separate dll(s). Add the [assembly: HostingStartup(typeof(HostingExtension.HostExtension))] tag, and implement the IHostingStartup interface. So, you can easily inject services and configuration in runtime. If you are injecting a service implementation, that will substitute our default implementation of the provided service.
2. Then build your own extension and copy it to the container, the location must be the same where the host executable binaries are located. .
3. Set the container environment variable "ASPNETCORE_HOSTINGSTARTUPASSEMBLIES" to the corresponding assembly name that is copied to the container(e.g: HostingExtension)

If multiple assemblies are copied, then separate all assembly names by ";". The calling order will be the same as it present in the environment variable. The consumers shall be informed that an assembly can override implementation from the previous assembly! The prototyping of this functionality is part of the next steps.

Please see the code example below.

Commented [GC20]: This is covered somewhere how to do that with our AuthSrv. Please add cross references

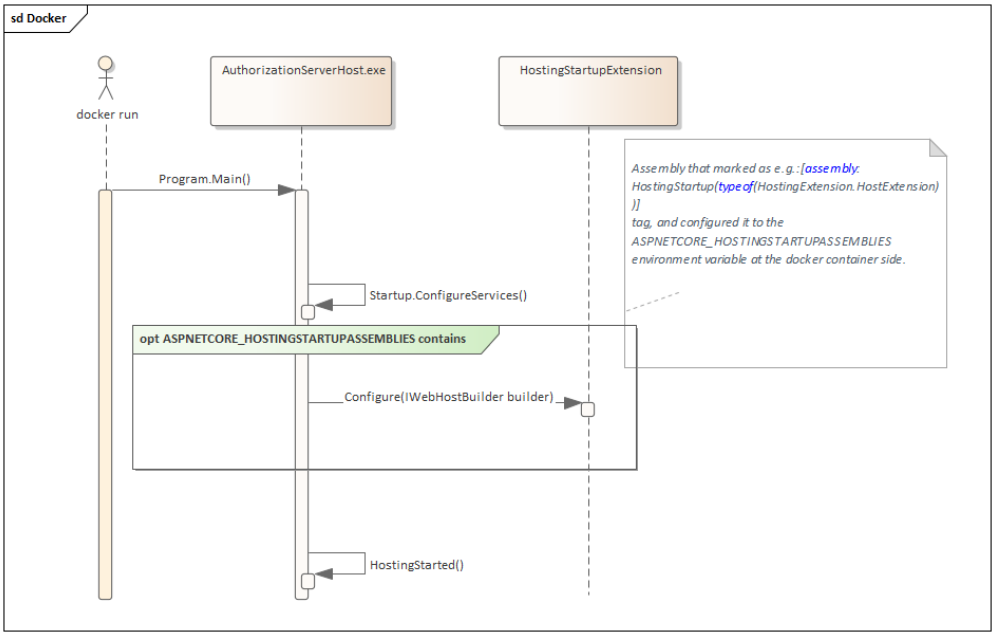
Commented [B421R20]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [GC22]: What is the order of calling extensions? (I guess it is listing order)
The consumers shall know/be informed that an assembly can override implementation from previous assembly!

Commented [B423R22]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [S424]: Okay, so this activity is listed as topmost Development activity at the end of the whitepaper, please refer to that... as this very important

Commented [B425R24]: @Borka, David (ADV D EU HU OPS 4 4)



Example:

```
[assembly: HostingStartup(typeof(HostingExtension.HostExtension))]
namespace HostingExtension
{
    public class HostExtension : IHostingStartup
    {
        public void Configure(IWebHostBuilder builder)
        {
            builder.ConfigureServices((context, collection) => collection.AddTransient<INamePasswordAuthenticationProvider, DummyNamePasswordAuthentication>());
        }
    }

    internal class DummyNamePasswordAuthentication : INamePasswordAuthenticationProvider
    {
        public string Name { get; }
        public Task AuthenticateAsync(NamePasswordValidationContext context)
        {
            throw new NotImplementedException();
        }
    }
}
```

Changing the app configuration:

- IWebHostBuilder.ConfigureAppConfiguration method is for loading the configuration after the app's ConfigureAppConfiguration delegates execute. Hosting startup configuration takes priority over the app's configuration using this approach.

- IWebHostBuilder.UseConfiguration to load the configuration before the app's ConfigureAppConfiguration delegates execute. The app's configuration values take priority over those provided by the hosting startup using this approach.

This Hosting startup assemblies' approach is fulfilling our needs. Eventually, the consumer can extend our hosting app with (services, configurations, middleware's). Because of this approach even if the consumer only uses our docker (base) image we need to ship our libraries (e.g. NuGet package) too to enable consumers to (compile) depend on the interfaces we allow to implement!

If we would get rid of the library delivery, then the whole AuthSrv module needs to be reworked to implement an indeed micro-service architecture. The consumer doesn't have to inject DLL's just need to provide an API endpoint to implement the needed functionalities, the AuthSrv has to communicate to the provided endpoints.

So what changes are needed at the AuthSrv level?

The first approach is that we should provide a default implementation of all provided services to be used if the consumer did not inject their own implementations. Consequently, we need to figure out how to support all provided capabilities in the docker environment (e.g., WindowsAuthenticationProvider).

The second approach is to provide an empty hosting environment that does not contain any default implementations, only using the AuthSrv, so the consumer must provide the additional implementations, such as INamePasswordAuthenticationProvider.

The first approach is better if we would give a fully functional base image that anyone can use without changes, the consumer only must provide the configuration files (Certificates, Api Resource Configuration, Client Configuration), and it can be a long-term solution, but with the second approach we can move forward easily we just need to provide an empty hoster that takes advantage on the hosting startup assemblies functionality as a first step.

Kerberos authentication use case

AuthSrv supports authentication via a Kerberos service. This use case is possible from within a Docker container, but it has not been tried as part of this prototype. Theoretically no adaptation is required from AuthSrv's point of view.

III. Building the AuthSrv container image

How shall the docker image be delivered to the stakeholders? The docker documentation proposes two ways to build ASP.NET Core applications: link:

1. Build the image with the Dockerfile from source

Create the Dockerfile in the project folder and build it from the source.

Pros:

- paves the way to container debug support in Visual Studio

Commented [GC26]: As a consequence of this approach even if the consumer only uses our docker (base) image we need to ship our lib too to enable consumer to (compile) depend on the interfaces we allow to implement!

Commented [S427R26]: How to overcome this? mean ship the public interfaces only as library? Or something else to avid shiping intwo different form?

Commented [C428R26]: Why do we wanna avoid this? Why it is a problem having NuGet delivery and a Docker?

Commented [C429R26]: Is there a way to extract/export file from docker image? Namely exporting the lib and its dependencies would result a bunch of assemblies that consumer developer can use...or is there a more convenient way to work with docker images in Visual Studio? (if this aspect was not covered during the QR then please put this as open question)

Commented [B430R26]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [BD(DEHO4431R26]: The clear way is if we shipping our interfaces as separate DLL's so the consumer can rely on that and they can inject it to the hoster. The consumer can copy our DLLs from the container into the host but a little nightmare, the Nuget delivery is a more gentle way to provide interfaces to them.

Commented [B432R26]: Endpoint configuration.

Commented [B433R26]: Mention alternatives e.g MicroService Architecture.

Commented [GC34]: It's very good, that you separated the functionality from the actual container image build solution. Would be nice having either some conclusion here from the two alternatives or at least have a pro/con analysis

Commented [S435R34]: +1

Commented [B436R34]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [GC37]:
1.Build the image with the Dockerfile from source
Pro (next to the existing ones)
•Overriding default implementation is more easier
•Not necessarily need to provide default implementations on existing extension points
2.Build app based on the deliverables of the existing pipeline
Con (next to the existing ones)

Commented [B438R37]: @Kungli, Marton (ADV D EU HU OPS 4 4)

Commented [K439R37]: i think these points are not valid after all. Extensibility of the implementation does not depend on how the image is built. Dávid, opinion?

Commented [K440R37]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [B441R37]: Yes, you are right. This is not relevant to this topic.

- possibility to break down the image to multiple stages, e.g., build-env and release. This enables better upgrade/update capabilities and flexibility for a container-based CI/CD pipeline

Cons:

- The product and/or the build environment needs additional enhancements which are not yet prototyped or known to be used within the project.
- Setting up a pipeline parallel to the already existing one requires a constant maintenance.
- Requires its own testing strategy.
- NuGet package restoring requires authentication in syngo private feed. Need to create a new account for docker container authentication.

2. Build app based on the deliverables of the existing pipeline.

Provide a Dockerfile to build the image from the binaries delivered with the already existing pipeline.

Pros:

- No need to maintain parallel pipelines and test strategy.
- Requires less modifications on the AuthSrv infrastructure.

Cons:

- No integrated container debug support within Visual Studio is possible

This whitepaper is not decided on which solution is preferred. This is marked as a follow-up topic at the end of the paper.

Commented [S442]: Conclusion, desired approach of the two fundamentals?

Commented [B443R42]: @Kungli, Marton (ADV D EU HU OPS 4 4)

IV. Customizing the AuthSrv container image

As mentioned earlier, the AuthSrv implementation and thus any base image that is to be delivered to the stakeholders must be customizable in the following aspects:

- The certificate needs to be configurable and replaceable.
- JSON-based configuration needs to be replaceable.
- Providers such as Auditing and Authentication shall be replaceable, dynamic dependency injection shall be possible within the implementation.

Using a dockerfile to create a custom image

Stakeholders may create their own custom container images based on AuthSrv. Regardless of using a platform-provided base image or merely building upon AuthSrv's binaries as they are delivered today, with a dockerfile it is possible to build a new image that contains new items or uses a

different base image, e.g., a Windows Server base. The base image could contain the Environment variable, that will be used in the container, so we can set the “MED_BIN” variable for pointing out our image location. This environment variable could be overridden in the consumer dockerfile.

The below example demonstrates one of the several options on how the dockerfile could be used to build upon an AuthSrv base image.

```
FROM AuthSrvContainerImage:1.0

ENV MED_BIN=C:\\app

WORKDIR /app

COPY /customItems/*.* ./

COPY /customItems/testInputData/AuthorizationServer/ServiceMocks.json
/app/testInputData/AuthorizationServer/

ENTRYPOINT ["dotnet", "syngo.Security.AuthorizationServer.DummyHost.dll",
"AssemblyLoadingProtoPreset"]
```

In the above example

- the AuthSrvContainerImage v1.0 image is used as a base image
- The environment variable MED_BIN is set
- New items are copied into the new image
- A new entrypoint is specified (optional)Using volumes and the dockerfile for extending the AuthSrv container image

For later examples this image will be named as customImage:0.1, as if the command ‘docker build -t customImage:0.1’ were called to create it.

Any files may be copied into the image, including binaries, certificates, or the initial configuration. The files can be modified later when or after a container is created from the image.

Remark:

The dockerfile can’t be standardized, because it is not supporting it only the syntax is standardized, the dockerfile fate is on the consumer side.

Customizing docker containers

As discussed in the containerization overview section, a container is not equal to the image, it is similar to an implementation of a type, where the implementation is the container, and the type is the image. When a container is initiated, it can be further customized.
While it is possible to copy configuration data as well as binaries inside an image when it is created, our recommendation is to keep the image, and with it, the dockerfile, for binaries, whereas configuration files and the certificate should be placed within the container by other means. Due to AuthSrv’s architecture, many configurations can be set with files that can be copied into the containers during or after container creation.

Commented [S444]: Would be a need for standardization here? It can be varry between the different containerized services?

Commented [B445R44]: Make a remark, and add to the open points @Borka, David (ADV D EU HU OPS 4 4)

Check this overview on how copying files between the host and the container is possible.

Commented [C446]: How does this come here?

Commented [K447R46]: it can be removed, it was for my reference and I found it to be an excellent overview of the topic

Commented [K448R46]: it could be placed in a new chapter that describes this very aspect

Commented [C449R46]: There was another part of the doc which was detailing this copy. I think it is enough if you just put this reference there.

Commented [B450R46]: @Kungli, Marton (ADV D EU HU OPS 4 4)

Using bind mounts

docker run -it -v d:\certs\:c:\app\certs -v "D:\configs\ConfigAuthSrv:c:\app\ConfigAuthSrv:ro" -p 47101:47101 –name customContainer customImage:0.1

In this example

- certificate is mounted as a bind mount into the container
- Configurations are read only – ':ro' – are mounted as a bind mount
- The host OS' 47101 is assigned to the container

Pros of using bind mounts:

- The same directory may be mounted for several containers, thus adding new files might be easy.
- Can add configurations etc. during container creation, which might not have been available or valid during the image build.

Cons:

- This approach uses a direct path in the OS' filesystem. As such, this approach is not OS agnostic.
- Security concern: the container is given (although quite limited) access to the host's file system.
- File ownership issues: files created inside the bind mount by the container will have root ownership by default.

Using the docker cp to copy files from/to the container

With docker cp it is possible to copy files between the host and the container. However, the command's scope is only one container, which is not preferable.

Commented [S451]: Is it recommended in community do do such? I think not, as it copies to the running container instance, so not be applicable for further instances. Simply looses the benefit of the approach in my eye.

Commented [B452R51]: @Kungli, Marton (ADV D EU HU OPS 4 4)

Using a named volume to share configuration between containers

The same named volume may be assigned to several containers and then data may be copied into the name volume.

Pros:

- The shared configuration directory can be stored within the docker runtime of the host.
    - Therefore, a shared volume can be used without many of the drawbacks of bind mounts.
    - This volume is easier to be handled by docker compose or container orchestrators.

Commented [S453]: Example command line, config for that?
So this is a kind of a preferrred approach?
Btw. is there anything like secure volume, e.g. for the certificate?

Commented [B454R53]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [K455R53]: Let us discuss this Dávid, I believe this might not be necessary to provide examples here. I have not tried it, here only the options are listed.

In an orchestrator environment it is recommended to use volumes instead of mounts for improved security and flexibility. The docker cp is not recommended to be used for setting up a container.

Authenticating Windows users by the default hoster

Warning! This is not the proposed solution to handle REST connection authentication. This is a workaround presented to make the default authentication provider work currently, for demo and testing purposes.

Currently the default hoster authenticates REST requests by windows user accounts, that is, the request shall include a username and password currently, which is authenticated against users defined in the host OS' user repository. Since a container has no access to the host's user repository, the users shall be available within the container to be able to support the default authentication provider.

Users may be defined in dockerfiles or later within a created container with 'docker exec'. Example:

```
docker exec -u Administrator -it authsrv.platform net user testuser TestPassword_123 /add
```

V. Persisting logs created by the container

General log handling of Docker

"By default, docker logs or docker service logs shows the command's output just as it would appear if you ran the command interactively in a terminal. UNIX and Linux commands typically open three I/O streams when they run, called STDIN, STDOUT, and STDERR. STDIN is the command's input stream, which may include input from the keyboard or input from another command. STDOUT is usually a command's normal output, and STDERR is typically used to output error messages. By default, docker logs shows the command's STDOUT and STDERR. To read more about I/O and Linux, see the Linux Documentation Project article on I/O redirection." src

The default behavior mentioned above uses the json-file logging driver of docker. Under the hood, this driver stores its logs in the file system in the Docker Root Dir in a json file. Consider limiting the size of this file with the daemon configuration file when using the default logging driver.

Thus, the recommended solution is to write logs to STDOUT and STDERR, since this is the industry standard, and Docker and external tools e.g., data collection software, orchestrators, are well prepared to handle these streams.
The default AuthSrv hoster is logging to STDOUT already. STDERR is not used by the current implementation of AuthSrv.

Nevertheless, it is possible to configure logging drivers to send logs to a file, an external host, a database, or another logging back-end, for example to configure log forwarding to a syslog server, or use logging plugins to further customize logging behavior.

Furthermore, technically it would be possible to save logs to a docker volume or a bind mount and write an external service to forward them from there, but this is not the industry standard approach.

Commented [S456]: This section suppose to cover: the 1st acceptance criteria?: "Enable AuthorizationServer hoster executable to load product specific implementations", then make it clear.
Also how they can load additional services also tot he provided hoster? (they have such usecase also)

Commented [K457R56]: @Borka, David (ADV D EU HU OPS 4 4) please take this comment into account when writing the dependency injection chapter

Commented [S458]: This is definately not the solution for this issue (actually, it is rather a dependency category). MArk it as limitation.
Please state that it need to be resolved, and what is the desired architecture of the solution.
Is this aspect meant by "Windows AD authentication prototype Container service account" development activity? Then we have to talk about the priority of that

Commented [B459R58]: @Kungli, Marton (ADV D EU HU OPS 4 4)

Commented [GC60]: Is this the only solution?
Ok, currently our executable using our WindowsAuthentication provider implementation, but why do you stick to that so hard? That's just an implementation to an existing extension.

Commented [K461R60]: it had been briefly mentioned earleir that the current implementation, if we were to keep it, needs to be adatped. This is the part where the actual workaround for using windows authentication is explained.

Commented [C462R60]: Ok, my strategy would be here is rather accept a limitation than doing workarounds. Only do the workaround or resolve limitation when needed. I'm not against what you wrote here, just reading through the doc I have the impression that we want to change the current implementation....btw. if really that's the case to authenticate HostOS local users (e.g. Scanners) then I'd also consider hosting AuthSrv by either IIS or in LCMService or in ServiceController instead of struggling with docker containers...fortunately we don't have to decide here...ye ...

Commented [K463R60]: well this is not the only solution, it is the way to make it work with the current ...

Commented [B464R60]: @Kungli, Marton (ADV D EU HU OPS 4 4)

Commented [S465]: Please add that fluentd is able to collect logs and feed to Elasticsearch

Commented [K466R65]: fluentd is just an example for a tool that handles STDOUT. It is not decided to use this too ...

Commented [S467]: STDERR is not used?

Commented [K468R67]: no

Commented [K469R67]: not yet at least

Commented [S470]: what does it mean? to use fluentd->Elasticsearch one have to configure something? OR it is by default? ...

Commented [K471R70]: In my interpretation it is the scope of the prototype to add examples for all aspects tha ...

Handling multiple log streams

The default AuthSrv logging implementation writes to STDOUT currently. Stakeholders when they extend this component with custom providers, e.g., a new auditing provider, shall consider that while it is advised to write logs to STDOUT, mixing several log streams to STDOUT might not be a good idea within the same container, because a data collection software would require further configuration to be able to separate mixed logs from the same output. Moreover, it is not possible to setup more than one log drivers for a container. When a container handles several log streams, e.g., auditing, standard logging, tracing, the industry standard is to create sidecar containers for each stream.

A "sidecar" is any docker container that you use alongside another docker container to work with it in some way. In this case for each of your three logs you would have a separate container that scans or tails the logs and outputs to STDOUT.

More info on the Kubernetes documentation: link

Commented [GC72]: These are relying on three different interfaces (i.e. extension point), I don't get why do we need to complicate it.

Commented [K473R72]: experts do not recommend writing different log types in the same STDOUT. This is the standard way to separate them.

Commented [C474R72]: Let's discuss this. I think we are not on the same page here.

Commented [B475R72]: @Kungli, Marton (ADV D EU HU OPS 4 4) Create a new open point for this.

Logging

AuthSrv uses the ASP.Net Core ILogger type to handle logging. Custom loggers shall implement this interface.

Auditing

The default hoster implements the n/s platform's IAuditWriter interface. IAuditWriter writes its logs to the configured output. When replacing the audit writer provider and choosing the output, refer to the 'General log handling of Docker' chapter of this whitepaper.

Tracing

As a design decision, no tracing is implemented in the current AuthSrv implementation, yet. When using tracing in custom providers, consider the points mentioned in the 'General log handling of Docker' chapter of this whitepaper.

Commented [S476]: Write a sentence why it is not used. e.g. because the logging of the OTS is enough, etc.

Commented [B477R76]: @Kungli, Marton (ADV D EU HU OPS 4 4)

VI. Integration aspects of containerization

The pipelines shall satisfy the following points:

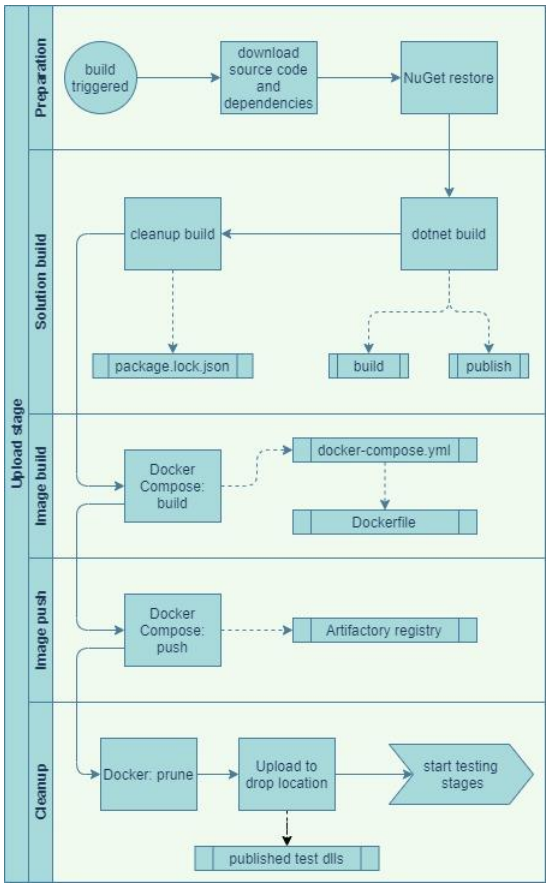
- Building Docker image from module codebase.
- Pushing created Docker image to registry hosted in Forchheim (Artifactory) or Budapest infrastructure.
- Pulling Docker images (syngo product or other baseimage) from the registry.
- Starting container with the pulled Docker image.
- Performing tests on the running Docker container.
- Running other workflows which supports quality in containerized delivery (gated check-in, regression, backward compatibility).

Analyzis of syngoONE pipeline

All syngoONE projects use a standardized multistage pipeline structure to handle Docker image building, pushing, and testing. The followings describe the details of such pipelines. The documentation does not describe every build task necessary for a working pipeline, only the ones that are important in the aspect of containerization. The not mentioned tasks are maintained by the infrastructure teams (e.g., SyngoTools install).

Pipeline structure for upload

The first stage of the pipeline is responsible for the source code build, image build and push. It is performed by a build agent in SY-FOR-Coruscant-Upload pool. The build agent has .NET Framework and Docker installed on it. SysInt utilizes Docker Desktop which also installs Docker Compose. Scripts and processes launched by the pipeline tasks can access the OAuth token through the `System.AccessToken` variable is used for authentication to Syngo-extern.



Summary of the upload stage

Commented [S478]: My impression that we need a chapter here, as the upcomming is the analyzis of the solution used by syngoONE

Commented [B479R78]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [S480]: Is there a diagram/image available? I mean I guess below the details of these steps are described, would be way more easy to understand/follow

Commented [CA(DEHO4481R80]: I don't have yet, but I plan to make one.

Commented [CA(DEHO4482R80]: done

Commented [S483]: May somewhere at the document conclude if docker compose files or dockerfiles with customized docker run command are the desired approach? My gut feeling based on your presentation is that the first one.

Commented [CA(DEHO4484R83]: In the previous paragraph it was stated that the followings describe a standardized pipeline structure used by syngoONE projects. Later, in the 'Alternative solutions for Fnd-AuthSrv' section, I stated that I favor this approach compared to the simplified one. Assuming that the Dockerfile and the Docker Compose yaml files are part of the pipeline, then I can also state that this Docker/Docker Compose approach is desirable.

Step 1: Preparation

The `Syngo TFVC Workspace Syncer` task downloads the source code from the given branch. `Use NuGet x.y.z` and `NuGet restore` tasks install NuGet tool and download every NuGet package dependency. All necessary source code and dependency should be on the build agent by the end of the preparation. It's important to state here that the NuGet process has to be used even if we decide on the side of the containerized delivery in the future, using Docker images will not replace NuGet packages.

Commented [S485]: This implicity makes a quite important statement that NuGet repo is used for the binaries (no propriatery solution at all)

Commented [CA(DEHO4486R85]: The description has been updated.

Further information about NuGet handling can be found here:

https://healthineers.sharepoint.com/sites/999-00002005/SingaporeAssets/Core/Core%20SDK%20Wiki/Delivering%20projects%20as%20NuGet%20packages.aspx

https://healthineers.sharepoint.com/sites/999-00002005/CoP/CoPIntegration/Wiki/Home.aspx (NuGet section)

Step 2: Building the downloaded codebase

The build is executed by a `.NET Core` task which calls a `dotnet build` command on the module's solution. It receives a `--no-restore` flag which tells the task not to restore dependencies here, because it was done by the preparation.

This way of build is only possible because all the module's projects (product and code) are organized under one solution named `{module's name}_All.sln`. The SCPs are still organized into solutions which are placed under the module's solution. With this approach the build process does not create any „monster" solution (which is created if the module has the one SCP - one solution model), so we can easily target the root solution with our tasks.

After the build is finished, the `Siemens.TFS.BuildTool` collects the `package.lock.json` files and copies them to the artifact staging directory. These files contain the complete NuGet package dependency tree, it can be used for exact reproducibility. It is necessary because most of the NuGet references require a minimal version from a given version range, thus it can be problematic to restore an exact NuGet package version vector later.

Additional settings in project structure

The `.csproj` files are modified so an automatic publish command is executed after every build in the pipeline:

Commented [S487]: I guess "build in the pipeline", e.g. not after build at the local machine.

```
<Target Name="AutoPublishAfterBuild" Condition="'$(PublishDir)' != ''"
AfterTargets="Build">
    <PropertyGroup>
      <NoBuild>True</NoBuild>
    </PropertyGroup>
    <CallTarget Targets="Publish" />
</Target>
```

The publish directory for product assemblies is placed at the module's root:

```
<PublishDir>..\..\..\publish\{module's name}</PublishDir>
```

There are additional build settings organized in `.props` files which can be found under `_Globals` folder. These settings can be imported in the `.csproj` file like so:

```
<Import Project="..\..\..\_Globals\{settings file name}.props" />
```

The test assemblies are also published at the root (in different folders than the product publish folders). This is set up in a `.props` file:

```
<PublishDir>..\..\..\publishTest\$(AssemblyName)</PublishDir>
```

The test projects reference the product project, so the product assemblies will be also in the test publish folder.

Commented [S488]: also? I mean above mentiones "publish directory for product assemblies"

Commented [CA(DEHO4489R88]: clarification added

Step 3: Building and pushing Docker image

First, we create an environment variable which stores the Windows version of the build agent:

```
echo ##vso[task.setvariable variable=WinVersion]1809
```

This variable will be used by the Docker processes later for image tagging/versioning.

Most Docker related actions are done by Docker Compose tasks. The module has a `docker-compose.yml` file at the root which contains information about all the Docker images which must be built. Such file looks like this:

```
services:
    imagename1:
        build:
            context: ./publish/{module's name}
            dockerfile: ../../{SCP name}/Source/{project name}/Dockerfile
            args:
                WinVersion: ${WinVersion}
        image: "{registry path}/{module}:${BuildVersion}-
{additional tagging info}-${WinVersion}"
    imagename2: …
```

`services` is a keyword, the Docker images' details will be listed after that. After that comes the image name. This image name is only used inside of this file for structuring. Under `build` keyword comes the build details for the Docker file:

- `context`: The context of the Docker file, it is good practice to set it to the publish folder.
- `dockerfile`: The path to the Dockerfile which is used for image building. Notice that the path is relative to the context.
- `args`: We can give environment variables to the Dockerfile. Here we are passing the Windows version which was defined earlier. Environment variables are also used at the image name.

The `image` keyword stores what the new image's name will be in the registry. The image name structure and the registry path is maintained by SysInt team. The syngoONE projects use this path: `shs/syngo/carbon/space`. The image tagging contains the build number and the Windows version. The additional tagging info is the base image used in the referenced Dockerfile. For example, KDIS uses *dotnet/core/aspnet:3.1.3-nanoserver* as base image in its Dockerfile, thus the full image name of KDIS is: `shs/syngo/carbon/space/kdis:${BuildVersion}-nanoserver-${WinVersion}`

Commented [S490]: This is proposed to be a standardized content? I mean is there any xinclude like possiblity in docker file? So that a base with the common part can be maintained centrally? Or each have to be manually maintained?

Commented [B491R90]: No, the docker file can not be standardized in that manner which you describe, it has its own syntax that needs to follow, thats all. It can be varied by the consumer.

Commented [B492R90]: add it as a remark

Commented [CA(DEHO4493R90]: clarification added

A referenced Dockerfile looks like this:

```
ARG WinVersion

FROM dotnet/core/aspnet:3.1.3-nanoserver-${WinVersion}
```

```
WORKDIR /Program Files/Siemens/syngo/bin
COPY . .

ENV ASPNETCORE_ENVIRONMENT Development

ENV MED_CONFIG "C:\Program Files\Siemens\syngo\bin\Config"
ENV MED_LOG "C:\store\log"

EXPOSE 80

USER ContainerAdministrator

ENTRYPOINT ["cmd", "/k", "entrypoint.bat"]

HEALTHCHECK --interval=10s CMD curl --
fail http://localhost/api/healthcheck/ping || exit 1
```

- ARG: We will use the Windows version environment variable.
- FROM: Base image is *aspnet:3.1.3-nanoserver*. Environment variable is used to fetch an exact image version.
- WORKDIR: Working directory is defined here. Every path inside the image will be relative to this folder. syngoONE projects use the following path: */Program Files/Siemens/syngo/bin*
- COPY: Copying everything from context (publish folder) to the working directory.
- ENV: Define environment variables for the image. ASPNETCORE_ENVIRONMENT is set to Development so the usage of https protocol is avoided.
- EXPOSE: Exposing port.
- USER: Define user for image.
- ENTRYPOINT: A batch file should be executed via command line after the container started. An entrypoint batch file could contain additional folder and file preparations with logging. After that, the service itself could be started with a `dotnet service.dll` command.
- HEALTHCHECK: An URI endpoint is pinged with *curl* every 10 seconds. This can show us if the running container is not working correctly (e.g. stuck in an infinite loop – in this case the container status would be running while it is stuck).

Currently there is no schema (like in the case of xml files) process implemented for the Docker/Docker Compose files. Each Docker/Docker Compose file has to maintained by the owner team (sometimes integration farm responsible help might be needed for that).

The Docker Compose file and Dockerfile is structured this way because no matter how many Docker images we want to build from given module in the future, we have to update only the Docker Compose file after the requested Dockerfile has been created. The pipeline's structure won't be impacted if we want to create more images/containers.

Let's see the pipeline. First we have build the new Docker image with a Docker Compose task. The setup:

- *Container Registry Type: Container Registry* – We want to use a private registry which is the Artifactory.
- *Docker Registry Service Connection: for.artifactory* – Artifactory's service connection.

Commented [S494]: Is it desired to use such, or a simpler would be desired?
I mean in the above exaplne f AuthSrv it mentiones c:\app

Commented [B495R94]: The mentioned example is a syngoONE use case.

Commented [S496]: This is very important mandatory aspect of the image. Refer to the Development next step

Commented [B497R96]: @Borka, David (ADV D EU HU OPS 4 4)

- *Docker Compose File: $(build.sourcesDirectory)/docker-compose.yml* – Path to the Docker Compose file.
- *Environment Variables: WinVersion=$(WinVersion), BuildVersion=$(SHS.BuildVersion)* – We pass the Windows and build version to the Docker Compose file.
- *Action: Build service images* – Docker Compose will build the image.
- *Additional Image Tags: $(WinVersion)-latest, $(WinVersion)-$(Build.SourceVersion)* – Additional image tags

After it's done, we have the Docker image locally and we have to push it into the Artifactory. We use the same task and only one settings will differ from the previous one:

- *Action: Push service images* – Pushing image into registry.

Step 4: Cleanup

After the image is pushed, we want to clean the agent from the images. For that a Docker command is executed which results in clean slate: `image prune -a --force`

We archive the test publish folder and move it to *$(Build.BinariesDirectory)/testbin/testbin.zip*. We upload the zipped test binaries to the drop location using `Publish build artifacts` task. We also upload the contents of artifact staging directory (e.g. the `package.lock.json` files can be found there) the same way.

Image signing

A built-in docker service exists for signing docker images, called Notary service. The service automatically installed within the Docker Desktop in the local machine. From the integration point of view, the SysInt must install and configure it. We can control who can sign a docker image and, we can specify what signing key will be used for that.

The client, that is the consumer of the docker image should turn on the feature by setting to the "DOCKER_CONTENT_TRUST" environment variable to '1'. This prevents users from working with images unless they contain a signature.

During the release pipeline, we should consider extending the pipeline steps with the signing procedure, eventually, we should use the following CLI:

"`docker trust signer add --key cert.pem example registry.example.com/admin/demo`"

The key file should originate from a certificate authority, and on top of that should be configured in the Notary service side.

More information about the Docker content trust and the usage with a self-signed certificate can be found here.

Pipeline structure for testing

After the image build and push is complete, the next pipeline stage starts which will perform the testing. The test agents' processes also can access the OAuth token through the System.AccessToken. The following existing agent pools will be used for testing:

- unit tests: **SY-FOR-SysInt-uTest**
- integration tests: **SY-FOR-SysInt-iTest**

Commented [S498]: This is not part of the above sequence. Where is this chapter in the structure?

Commented [S499]: Who/where shall install this service? SysInt?

Commented [B4100R99]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [BD(DEHO44101R99]: I refer this section from next points

Commented [S4102]: What is the "client" here?

Commented [B4103R102]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [S4104]: Please add conclusion on how testing shall be done (high level approach)

Commented [CA(DEHO44105]: done

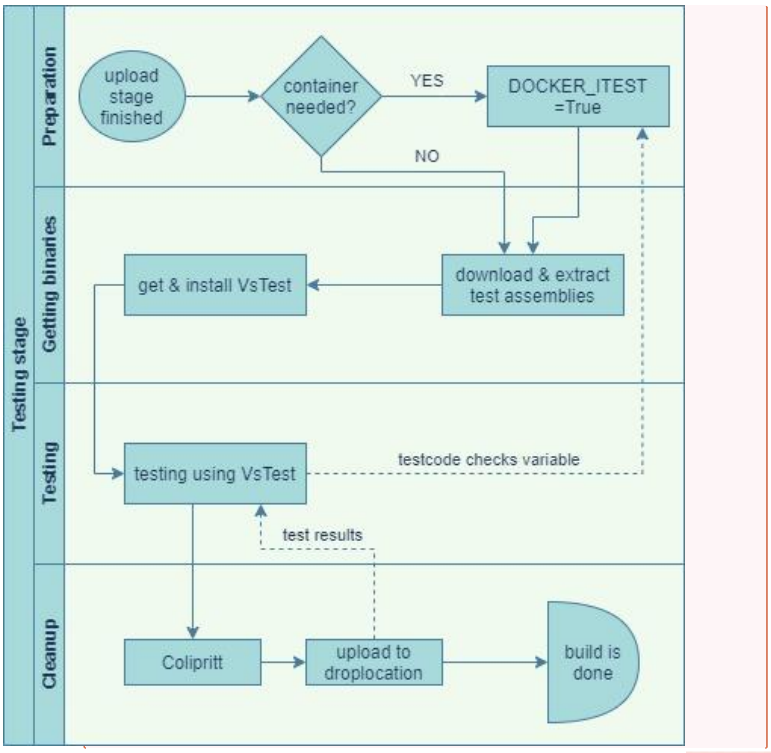
Commented [S4106R105]: Then please mark the comment as "Resolved", see [...] menu

Commented [B4107]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [S4108]: Is it a decision or fact? Or it is the current available infrastructure setup?

Commented [B4109R108]: @Borka, David (ADV D EU HU OPS 4 4)

- integration tests using Docker container: **SY-FOR-SysInt-Docker**



Summary of the testing stage

Testing without container

SysInt created tasks groups which handle test execution in a compact way. The task groups execute the following tasks:

1. First, the test artifacts will be downloaded from the drop location. Remember that the test binaries were uploaded in the Cleanup step.
2. Extract the downloaded artifact zipped file.
3. Installing Visual Studio Test Platform (VsTest) on the test agent. The setup is done via NuGet package using the syngo-Extern feed. `System.AccessToken` variable is used for authentication for accessing the syngo-Extern Nuget feed.
4. Test execution with VsTest task. The task searches for test assemblies in the downloaded artifact folder. The process will skip tests with Explicit category. Different task groups will look for different test assemblies:
    - The unit- and component test task group searches for `**\*_utest.dll, **\*_ctest.dll` patterns in the assembly's name.
    - The integration test task group searches for `**\*itest.dll` patterns in the assembly's name.
5. Calling `Colipritt` tool which logs the hardware and software elements of the testagent into an `.env` file.

Commented [S4110]: Where is the "container needed"=NO path?

Commented [CA(DEHO44111R110]: fixed

Commented [S4112]: When is desired to use this way? (e.g. for unit tests?)

Commented [B4113R112]: The fourth point is describing all related uTests and iTests will be executed without the container.

Commented [S4114]: for what? For accessing the syngo-Extern feed?

Commented [B4115R114]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [S4116]: The other test distrubution capabilities of the propriatery infra is available? E.g. parallel execution, etc?

Commented [CA(DEHO44117R116]: It has been registered as a next steps action point with medium priority.

6. Upload the `.env` file and the VsTest results to the drop location.

Testing with container

The pipeline creates the DOCKER_ITEST environment variable which will tell the test logic to use Docker container:

```
echo ##vso[task.setvariable variable=DOCKER_ITEST]True
```

After that the integration test task group will be called. The container handling will be done via the test logic inside the text fixture, whose structure will be discussed in the next chapter.

Project structure for container testing

Functions with `[OneTimeSetUp]` and `[OneTimeTearDown]` attribute within a class with `[SetUpFixture]` attribute handles the containers. These functions will be called before the first test execution and after every test was executed, thus they are responsible for starting and stopping the containers.

First, they check if there is an environment variable called DOCKER_ITEST with a value `True`. If there isn't then it will not start handling the containers. This variable was defined by the pipeline before the test assembly was called.

With the help of `System.Diagnostics.Process`, they call docker-compose and pass it the Docker Compose file (which is called mock; in this case mock means that it spins up a container with the product in it, but it's only used for testing), so it could create or stop a container for testing:

```
docker-compose.exe -f {path}\mock.yml up -d
```

for starting container and

```
docker-compose.exe -f {path}\mock.yml down
```

for stopping it.

They enable the process to raise events, so they can wait for termination (30s timeout) and check if the process terminated with no error. Besides that, they create additional environment variables for the mock Docker Compose file (e.g. URL of the Artifactory registry). The paths used here is relative to the test assembly currently executed.

They save the container logs before shutting down the container. Using the same approach as at the container control, they execute a command to print the logs:

```
docker.exe logs dockercompose_{container name given in mock file}_1
```

The process output is redirected to the standard output, so it can save it line by line using `StreamWriter`.

A mock Docker Compose file looks like this:

```
services:

  mock_container_1:
    image: {Artifactory registry}/{registry path}/{module name}:{Windows versi
on}-{tagging}
```

Commented [S4118]: Can you please ellaborate? What does it mean? The container start-stop etc. need to be done by the test code? Will there be a pipeline enabler for this? Or need to be (proposal)?

Commented [CA(DEHO44119R118]: Up until this point only the pipeline was discussed, after that the test logic is elaborated. Clarification has been added.

Commented [S4120]: Is it a subcahpter of the previous one, or what is the hierarchy level of this chapter?

Commented [S4121]: "mock" means an empty file here, or what is the definition of that? Who creates this file?

Commented [CA(DEHO44122R121]: clarification added

```
    volumes:
      - ..\:C:\Program Files\Siemens\syngo\bin\testbin
      - ..\SyngoShare\Certificates\:C:\Program Files\Siemens\syngo\bin\Config\
Certificates
    ports:
      - target: 80
        published: 10101
        protocol: tcp
        mode: host
    environment:
      - VAR1=${agent's environment variable}
      - VAR2=Test

  mock_container_2: ...
```

- `image`: The Artifactory registry URL is given via environment variable. The tagging is set to *latest* because we want to test the image we just built.
- `volumes`: We can attach volumes to the container (e.g. certificates). The paths are relative to the Docker Compose file.
- `ports`: Port settings.
  - `target`: The port inside the container. Remember that the Dockerfile had an `EXPOSE 80` command.
  - `published`: The publicly exposed port.
  - `protocol`: Using of TCP protocol.
  - `mode: host` - We want a single machine setup (alternative would be `ingress` for swarm mode).
- `environment`: Defining environment variables.

After the container is up and running, the testcase can begin:

1. Checks if the test should be performed in Docker environment (checks the `DOCKER_ITEST` variable mentioned earlier). If it shouldn't then `Assert.Ignore(…)` will be called and the current testcase will be skipped.
2. Resolve the container service's base URI. The resolver method checks for environment variables describing e.g. reverse proxy host, secure communication. If none of the searched environment variables are present, then the resolver method will return with a default URI which has the same port as the one given to the `published` variable in the mock file.
3. Create an URI request.
4. Send an asynchronous GET request to the specified URI. Execution blocked until response is received.
5. Checking response of GET request using NUnit.

Alternative solutions for AuthSrv

The pipeline and project structure described earlier has one main goal: Applying an easily scalable solution from both DevOps and test logic/development perspective. The integration responsible doesn't have to update the pipeline if there are some changes in the Docker image setup (e.g. an additional Dockerfile has been defined). Considering this and the fact that this is the structure the Forchheim System Integration team can support with maximum efficiency, my suggestion is to go forward with this setup.

Commented [S4123]: The HEALTHCHECK is not verified here?

Commented [BD(DEHO44124R123]: The HEALTHCHECK is the extend the status of the container instance if you are querying by the „docker ps –aq" command. By default the command output giving details about the running container, but the actually hosted service status is not visible. By the HEALTHCHECK endpoint is giving that information whether the needed endpoint is reachable or not in the container.

Commented [CA(DEHO44125R123]: The Healthcheck next step has been updated to check the usage of Healthcheck in tests.

Commented [B4126]: Docker environment category in TargetSpec would be nice.

Commented [CA(DEHO44127R126]: High prio next step action item has been created for this.

Commented [S4128]: what "code"? Is it the development time?

Commented [CA(DEHO44129R128]: clarification added

Besides that, it's worth noting that after examining the current situation of AuthSrv module, which is:

- we plan to deliver one Docker image, and
- we plan to test using only one running Docker container,

we can state that there are possibilities to simplify the syngoONE structure.

Skipping the usage of Docker Compose files has the biggest potential for simplifying things because we need only a single container environment, thus Dockerfile should be able to control it. To do so, we have to outsource every information which has been handled by Docker Compose files, to the Dockerfile, pipeline and source code.

Such information on the image build side would be: We will make a conclusion via next steps activities.

- Docker Compose file passes the Windows version as an argument to the Dockerfile (which is accessible only during the image build process). The pipeline could pass this to a Docker `build` task using `--build-arg` flag.
- It also sets up the image name (under what name the new image will be stored), which could be done in the pipeline with a Docker task using `--tag` flag.
- Pushing to register can be done with a Docker `push` task.

And on the testing side:

- Docker Compose controls which image from which registry should be run as container. A Docker `run` task in the pipeline could handle this (image path/name is a mandatory argument).
- It attaches bind mount volumes, a Docker `run` task in the pipeline could handle bind mounting.
- It publishes a target port with TCP protocol, Docker `run` task in the pipeline can support it with `--publish` flag.
- It defines environment variables, Docker `run` task in the pipeline can set them with `--env` flag (or an entire file listing the variables using `--env-file` flag).
- Obviously, the test fixture setup also must be updated to execute Docker commands, not Docker Compose commands.

The conclusion is that we can cover every Docker Compose functionality with a Docker feature, so a simplified approach could be applicable.

Alternative approaches for testing

Currently the syngoONE projects test the running containers with a black box approach. An image is built, it is pushed to the Artifactory registry, then the test spins up a container and sends request to it while testing the responses. The advantage of this approach that the testing is done fully by consumers' perspective.

In the long run, we should plan for implementing white box testing, too, because running tests inside our application's container allows us a greater sense of assurance that everything works as expected. Some example:

- White box testing inside the official product container:
    1. We modify the Dockerfile so it copies not just the product, but the test assemblies, too. Before `ENTRYPOINT` command we can execute `dotnet test` command while

Commented [S4130]: For this very simple component yes. But if we think in general and standardized, does it make sence to deviate?
I mean not only AuthServer n/s platform want to containerize, but other components also. May be ones depend on AuthSrv... then still worth to deviate?

Commented [CA(DEHO44131R130]: Currently it's not woth it, details in a later comment.

Commented [S4132]: But is it worth it? OR rather accept this complication for this component, in order to support standardization...

Commented [B4133R132]: We will make a conclusion via next steps activities.

Commented [CA(DEHO44134R132]: Our agreement is that we will go forward with the syngoONE structure, so the current answer is that it's not worth it. But we don't want to drop completely the possibility of simplifying things yet, so we pushed its action item lower in the priority list. Later, when we know more, we will be more sure about this question.

Commented [S4135]: Please refer to followup at point 4 at integration next steps...

Commented [B4136R135]: @Borka, David (ADV D EU HU OPS 4 4)

targeting the necessary test assemblies with it. The testing part (copy and execution) should be placed in a separate build stage.

2. Advantages:
    - If a testcase throws an error, then the Docker task fails, thus the pipeline would also fail, and the image wouldn't be pushed into the registry.
    - With a multi-stage Dockerfile we could target only the testing stage, so we could execute tests without starting the service itself.
3. Disadvantages:
    - More complicated Dockerfile.
    - The container should be started right after the image build so the tests could run.
    - The tests would be executed every time the container starts.
    - The image would be bigger because it would contain the test assemblies, too.
    - *The last two points could be solved using Dockerfile conditionals and build arguments. We could turn off the stage which copies the test assemblies and starts the testing.*

- White box testing inside a test container:
    1. We create a separate Dockerfile just for testing. It would contain the same commands as the previous suggestion, except the `ENTRYPOINT` and `HEALTHCHECK` commands are not needed in this case.
    2. Advantages:
        - Simpler solution compared to the previous one.
        - It could be built and run before building the product image. It will break the pipeline if a test fails.
    3. Disadvantages:
        - Another Dockerfile must be maintained.
        - It's a less efficient solution because we would build Docker image twice every time.
        - *This one could be mitigated using temporary images, but it would also put an extra load on the pipeline.*
        - The container should be started before pushing the image to the registry so the tests could run.
- Black box testing "inside" a test container:
    1. We create a Docker image with the black box testing assemblies in it, so the test container would ping the product container.
    2. Advantages:
        - The tests run in the same containerized environment as the product.
    3. Disadvantages:
        - Another Dockerfile must be maintained.
        - Networking challenges could come up.

Other supporting pipelines

Besides the already discussed upload build, we need additional supporting pipelines to ensure the quality of containerized delivery:

- Check-in:
    1. Current status in syngoONE:

        - It works with continuous integration behavior, so the changeset will be checked in no matter which build task will fail.
        - It executes every unit- and integration tests that doesn't need a running container.
    2. Suggestions:
        - Use gated check-in behavior instead of continuous integration for more secure development.
        - Insert containerized unit- and integration white box testing into the pipeline which would also tests the shelveset.
- Regression nightly build:
    1. Current status in syngoONE:
        - No regression builds are implemented.
    2. Suggestions:
        - Implement a nightly build for regression testing.
        - Reuse the structure implemented in the upload build (excluding the image push).
        - Using containerized unit- and integration white box testing, plus containerized black box testing.
- Backward compatibility:
    1. Current status in syngoONE:
        - No backward compatibility builds are implemented.
    2. Suggestions:
        - Implement a nightly build for backward compatibility testing.
        - Reuse the structure implemented in the upload build (excluding the image push and solution build - all the binaries/images should be downloaded from repository/registry).
        - Using containerized unit- and integration white box testing, plus containerized black box testing.
        - As it was described earlier, the test assemblies are copied to the drop location by the upload build. The task which downloads the test assemblies, needs to be updated so it wouldn't download the latest test binaries, but the ones selected by the given baseline.
- Upload build:
    1. Current status in syngoONE:
        - The details of it were discussed earlier.
        - It works with a continuous integration behavior (meaning that the image will be pushed even if a test fails).
    2. Suggestions:
        - Using containerized unit- and integration white box testing, plus containerized black box testing.
        - Use gated check-in behavior instead of continuous integration for more secure development.

The gated check-in pipeline feature can be only used if the pipeline has a single agent job (see: https://developercommunity.visualstudio.com/t/vs403427-error-when-gated-check-in-build-pipeline/1247883). E.g. the current syngoONE upload has 4 agent job (build, unit test, integration test, integration test with container), each one works in their separate agent pool.

Commented [S4137]: What is our gut feeling. Our project will continue to use the gated check-in approach in this new environment, or it is time to change?

Commented [B4138R137]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [BD(DEHO44139R137]: This question is part of the integration next step points. I added a link at end of this section.

Commented [BD(DEHO44140R137]:

If we want a gated check-in behavior then we can use only one agent pool in the pipeline which must be SY-FOR-Coruscant-Upload. The white box testing could be a solution for that because to execute white box tests we only need a Docker or Docker Compose task which can be called in the SY-FOR-Coruscant-Upload pool. In case of check-in build, a failing white box test could fail this task, so it would break the pipeline before reaching the shelveset check-in task. In case of upload, the same failure could break the pipeline before reaching the image push task.

We have to consider that if we use additional agent pools for other types of testing (e.g. black box testing), then continuous integration is the only possible way. See integration next steps.

Strategy for test result handling and branching

We suggest deploying a new branch alongside the main branch. This new branch should be used to build Docker image from. We can merge the code base we want to see in the image to this branch, so we can control the image's content more easily. This would be a long-term solution as it is in SyngoOne because we can easily detect the problem if these two are separated into two branches. For example, what if the docker infrastructure facing an infrastructure issue, then the whole delivery will be blocked.

Commented [S4141]: This is for this transition period, or as a long term solution?

Commented [B4142R141]: @Borka, David (ADV D EU HU OPS 4 4)

An initial test result handling strategy could be prioritizing the test results. Possible priority order (from higher to lower - the higher priority points have higher impact on the consumer):

- Based on which build throws the error:
    1. Upload
    2. Backward compatibility
    3. Regression
    4. Gated check-in
- Based on test type:
    1. Containerized black box test
    2. Not containerized black box test
    3. Unit/integration white box test
    4. Classic unit/integration test

Initially, when a test base is still small, every tests can be executed by every build without any significant drop of efficiency. But later on we suggest the following test filtering:

- Check-in: affected tests should be run
- Regression: all tests should be run
- Backward compatibility: only tests marked by architects should be run
- Upload: only a small proportion of tests which covers the module's most important feature should be run

Pipeline hosting in Budapest infrastructure

The Budapest infrastructure is up and ready to host Docker based pipelines. Although a basic Docker pipeline has already been implemented in the BUD infrastructure, currently there is no such pipeline running there like the ones the syngoONE projects use in the Coruscant pool. Artifactory registry can also be accessed, if needed. Currently the only difference between the BUD and FOR infrastructure regarding the hosting of Docker processes is that the BUD infrastructure doesn't have opened ports towards Artifactory, but our current knowledge is that we can solve this difference with low effort.

Commented [S4143]: Any conclusion regarding proposed next step in the BUD infra? Does it need to be improved to match the capability of the FOR one, or not, or?!

Commented [CA(DEHO44144R143]: clarification added

VII. Conclusion

Product/business

According to n/s platform business vision/roadmap the platform shall move away from a monolithic release and deployment. In order to do so functionalities/capabilities offered by the platform shall be containerized.

AuthSrv is definitely a functionality which shall be provided as a dedicated container delivery as fulfill end-2-end security use cases (with end product customization). Mid- and long-term AuthSrv is state of the art successor of the platform proprietary security subsystem component. Keeping it up to date and versatile is a differentiator factor.

AuthSrv is a new, .NET Core-based component was a solid candidate for this pilot activity, to get familiar with these new technologies and techniques in the team.
This know-how can be utilized when defining and implementing the containerization of further functionalities in the domain.

Integration

From integration perspective a containerized delivery would be favorable in the long run. It gives us more control over the test- and production environment. The environment can be easily updated from source control (e.g., Docker Compose file, Dockerfile, test code calling container services) or pipeline. The build- or test agent only needs Docker installed on it, after that it depends only on the pipeline and the container files referenced by it how our environment will be built up. Because of that the structure of the supporting pipelines are more comprehensive and compact than a pipeline serving a classic repository-based delivery method. Additionally, less support from SysInt team would be needed because they only need to provide the installed Docker on the agent, the environment is set up in the Dockerfile. Although there are several open questions waiting to be answered and opportunities for improvement to be explored, the overall effort to satisfy these topics would be considerably lower compared to the potential profit.

Development:

Pros:

- By providing a default hoster that is extendible, the hoster's core functionality will not be duplicated by stakeholders, which is beneficial regarding security updates.
- The dockerization providing a separated environment, it can be a more secure way to offer authorization functionalities.
- Improved scalability, we can fulfill future needs if we need to support multi-server scenarios.

Cons:

- Windows AD authentication is cumbersome, because of the access limitation of the host OS.
- If we are choosing this delivery method, might we need to create a new branch for this, then we must maintain two branches. Further details can be found here.
- The current implementation of the Foundations-AuthSrv is not properly fitting into the Docker concept. There is room for improvements in the Foundations-AuthSrv module to fully support Docker delivery. See the next steps.

Commented [C4145]: Would be nice having a summary for all the details above. What would be our preference? How we would provide AuthSrv for Docker environment?

Commented [K4146R145]: next steps are detailed, and all previous chapters include suggestions on how to actually implement, wherever applicable

Commented [S4147]: So this section is the "Product" conclusion?

Commented [S4148]: That is a conclusion? I mean it means that there is no one solution possible fits to both deplyment method for this intermediate period?

Commented [B4149R148]: @Borka, David (ADV D EU HU OPS 4 4)

Commented [BD(DEHO44150R148]: I added a reference to the bracnhing section.

Commented [S4151]: Could be or will be?
I mean it is a new component using modern technologies... if such verdict for this, then what will be the case for the legacy components.
I personally, do not see - even fromt his document - this huge effort. These is some effort (generalize the hoster, solving the AD authentication and the certificate configuration), but it does not sounds huge. Huge means a complete rewrite...

Commented [B4152R151]: @Borka, David (ADV D EU HU OPS 4 4)

Next steps

Integration (from highest to lowest priority):

1. Creating a working upload pipeline in the Coruscant pool, which works like the other syngoONE builds (continuous integration, uploading to Artifactory registry, simple unit- and integration tests, black box tests).
2. Image signing
3. Prototyping alternative testing methods (containerized white and black box testing)
4. TargetSpec/BundleTestSpec support for Docker environment – how can we avoid the usage of Assert.Ignore method?
5. Creating supporting pipelines (gated check-in, regression, backward compatibility)
6. Prototyping gated check-in behavior in the supporting pipelines
7. Prototyping solution building methods – should it be done by pipeline or Docker?
8. Assessing the possibilities of VsTest; e.g other pipeline settings, parallel execution
9. Prototyping proposed branching and testing strategy
10. Synch with Forchheim infrastructure about the possibilities of hosting image upload build in BUD infrastructure.
11. Check the usage of VCRedist in Docker images – where/why is it needed?
12. Check the Visual Studio's support for debugging local multi-container environment.
13. Check additional build cleanup tasks (writing into Docker candidate file, tagging successful builds, Colipritt).
14. Image handling in a bigger picture; image promotion process; connection to release processes.
15. Prototyping the proposed solutions to simplify the upload/image file structure for Foundations-AuthSrv.

Our PIR has roughly 20% capacity to continue this activity in the QR2107, he will do so with the above mentioned integration related next steps, in the above priority order.

Development:

- Prototype the Dependency injection
- Security characteristic of the Docker compared to the current delivery method
- Check product-side realization for Dockerfile HEALTHCHECK monitoring
    - Check the possibilities of using the Healthcheck in the testfixtures
- Test if the Kerberos use case is working from a containerized environment.
- Check docker security aspects.
- Windows AD authentication prototype Container service account. https://docs.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/manage-serviceaccounts
- Figure it out the best way to configuration injection.(Certificates,ClientConfig,ApiResourceConfig)
    - Entity Framework?
- Certificate management
    - Allow to get hosting and signing certificate from different source (e.g. Kubernetes secrets)
    - Introduce Signing Certificate rollover mechanism => allows changing the singing certificate without restart of the system

Commented [B4153]: Refer to all section

Commented [CA(DEHO44154R153]: Every action point is referred if there is a specific section for it

Commented [S4155]: As mentioned above, does it make sence? Or rather have a standard solution?

Commented [CA(DEHO44156R155]: The details can be found in an earlier comment.

Commented [S4157]: What exactly this means? Did I miss something?

Commented [B4158R157]: @Kungli, Marton (ADV D EU HU OPS 4 4)

Commented [K4159R157]: it means that the kerberos use case should be tested. The document is adjusted that this is clear now.

Commented [H4160]: Certificate related changes as a development next step (as mentioned in chapter 2)?

Commented [B4161R160]: @Borka, David (ADV D EU HU OPS 4 4)

    - Certificate validation service https://en.wikipedia.org/wiki/SCVP, https://datatracker.ietf.org/doc/html/rfc5055
- External Configuration and Operational storage
    - e.g. https://docs.identityserver.io/en/latest/quickstarts/5_entityframework.html

These activities shall be planned for the upcoming QRs.
Ideally this should be part of the FT the component belongs to (FT05 in this case).
But it is very unlikely that BL will have any interest in this deployment/architectural topic, so platform strategy is necessary, like was for the modularization (dedicated FT), this case it could be FT12?!

QMS related:

- In MRS we shall create a cross cutting concern, describing the containerized version as another supported environment. Derive from that in the test specs.
- Q process for Container as delivery
- Architecture modelling: to be defined. Clarify with Lutz Dominick.
- Formal test reporting
- OTS clearing, handling in development and pipeline environment