 Product Design Overview

*Function Name:*

3DSCM service

Function and PES Owner ID

|  |  |  |
| --- | --- | --- |
| Function ID | PES Owner Name | PES Owner Role |
| Overview | IDC | IDC |

Document Maturity Status

Each time you have to change the PES maturity status, you may need to add a new MMn/ddn/yyyn row and copy/paste in the new row the pre-formatted columns from the previous row in the table below.

|  |  |  |
| --- | --- | --- |
| Date | Maturity | Notes |
| 7/19/2019 | *Drafted* | Initial version |

PES Table of Contents

[Section 1 Introduction 3](#_Toc20810955)

[Section 2 Service Design 4](#_Toc20810956)

[2.1 Service Overview 4](#_Toc20810957)

[2.2 Regions and Tenants 5](#_Toc20810958)

[2.3 Service installation, products, provisioning etc. 6](#_Toc20810959)

[2.4 Configuration 7](#_Toc20810960)

[2.5 User Access to Repositories 9](#_Toc20810961)

[2.6 Authentication and Access 10](#_Toc20810962)

[2.7 Persistent Storage 13](#_Toc20810963)

[2.8 Runtime monitoring and Supervision 13](#_Toc20810964)

# Introduction

This document provides an Overview of the Design of the 3DSCM service.

The intention is for this document to describe the overall service and then detail the design elements that are not covered in either the Service Referential or other specific Functional Specifications.

The main audience for this document are:

1. The Developers working on this service and the underlying product
2. The rest of the Connected Software team that are more familiar with our traditional way of documenting our design and implementation

Note that this is NOT a “PES” or “Functional Specification”, and therefore this document will not follow the standard for those items. It will simply be a series of sections discussing different aspects of the service.

This section provides a global understanding of the function scope: why must it be developed? What will it do and not do? How this new functionality does relate to what the product/component already does?

**Note:** Section 1 and Section 4 (acceptance tests protocol) constitute the “High Level PES” and must be clear and complete enough to enable reviewers to assess that this function will satisfy its associated requirement.

# Service Design

## Service Overview

The “3SDSCM Service” will be a “service” provided on-cloud ONLY. Initially this will be on Dassault Cloud. The question of whether this will be provided for Customer Cloud environments is undetermined at this point.

**Service:** A service is a set of related functionality (products) provided on one or more host machines (typically, virtual machines) and typically exposed to the end users through Web Apps and/or REST services exposed via one or more Apache TomEE servers. The 3DEXPERIENCE system is a whole set of services that inter-communicate. For example, the 3DSpace service provides relational database services and interacts with the 3DPassport service for user identification and authentication.

**Cloud Services:** Various 3rd-party companies provide facilities to host services. For example, Outscale or Amazon. This means that these companies provide the whole infrastructure (machines / disks / network) to run the services, and also provide management and monitoring of the services. When we talk about providing our services “on the Dassault Cloud”, we mean that they will be hosted on one of these 3rd-party companies, and our customers access the services from that 3rd-party company’s networks. It is possible for a (large!) company to provide their own Cloud infrastructure, and to deploy our “cloud” services on that infrastructure. This is known as “Customer Cloud”.]

**On Premise:** The alternative to using our products as part of a Cloud service, is for a customer to install them as individual products on their own machines. This is known as an on-premise installation. The existing “SCM Connector” products (allowing use of Git and DesignSync data with our Connected Software product) are “on premise”, since they can only be installed on a customer network.

The purpose of the 3DSCM service is to provide management of data (files) in an SCM system where the “repositories” are managed on-cloud. That is, the opposite of the existing “connectors” which only manage data on-premise.

The initial service will provide for management of “git” repositories, with the potential for other SCM data (such as DesignSync) to be managed in the future.

So, the 3DSCM service can be thought of as an on-cloud version of the “Git Connector”. The primary difference being that, instead of the git repositories being managed on the customer site, they are managed on our cloud servers.

Here is a picture of the previous system using the Git Connector:

On-premise

3DSpace (on-premise or on-cloud)

Software Items

SCM Repository

Connected Software WebApp

Git Connector

Adapter / Connecter Web App

User Browser

jgit

git repos (bare and clone)

User git client

Here we can see that the main “Software Items” are on 3DSpace, along with our “SCM Repository”. The “Git Connector” runs on-premise, and the actual git repositories are on-premise, and completely managed by the users / admins there.

The on-premise git repositories are created manually by the users, and it is up to them how they control access to those repositories, back them up, etc.

The equivalent picture for the 3DSCM service is this:

on-premise

3DSpace (on-cloud)

Software Items

SCM Repository

Connected Software WebApp

3DSCM (on-cloud)

User Browser

jgit

Local git repo clone

User git client

Git Bare Repo

**IMPORTANT**: This is a greatly simplified picture, just to help understand the remaining technical pieces of this document. For the full picture of the architecture, the “Service Referential” presentation should be read (See the Documents attached to [this object](https://dsxdev-online.dsone.3ds.com/enovia/common/emxNavigator.jsp?objectId=34152.36236.24325.56200) in RFL).

The primary differences are:

* The Main “Bare Repositories” are now managed as part of the on-cloud 3DSCM service
* There is no Adapter/Connector configuration web app.
* Users clone/pull/push their code from/to the 3DSCM-managed git bare repositories using whatever git client they prefer to use (command line, or plug-in to something like Eclipse.)

**Why do this?**For customers that do not want to manage their own infrastructure, this provides a way to have the main repositories managed on-cloud, and therefore all disk space and backup issues to be the responsibility of the cloud provider. Users just get their own clones (workspaces), which are always disposable.

Another way to think of this is that our 3DSCM system will be similar to a service like GitHub or GitLab!

A more general; reason for using Connected Software is so that customers can have 1 system that manages all of their data. 3DExperience manages their hardware design data, so having it manage their software data too keeps everything together. This also then brings the power of other 3DExperience apps to bear on the managed software source.

## Regions and Tenants

A “region” is an area of the world covered by a particular Cloud infrastructure supplier. For example, Europe is one region, North America is another. Regions do not concern us too much, except to note that the provider of the Cloud infrastructure can be different in each region. For example, Amazon might provide the infrastructure in the US, and Huawei in China. There can clearly be differences in the different providers! The aim is to have a single service that is the same across regions.

A “tenant” is basically a customer. Each customer gets access to the products that they have bought licenses for. But the actual services providing those products are not replicated for every tenant. For example, there is only one 3DPassport service that provides authentication for all users across all customers (tenants.) Similarly, one 3DSpace service (providing a whole series of products) might be used by multiple tenants. The 3DSCM service is similarly “multi-tenant”, meaning that one *instance* of the service is used by multiple customers accessing the 3DS cloud deployment in a region.

The important thing with multi-tenant services is that they MUST separate their data in such a way that it is never going to be possible for one customer to see the data belonging to another. For 3DSpace, this is achieved by storing the data in “vaults” in the database that are per-tenant.

For 3DSCM, we will need to partition the repositories, and provide access, in such a way that the repositories for one customer can never be seen or affected by another customer.

One important thing to note is that when there are multiple tenants, they always access the various servers using different URLs. This is important and the “id” of the tenant is part of that URL and can be used as part of the scheme for data separation.

Note that a single customer (tenant) is only ever present in a single region: They do not have their data / services spread across regions.

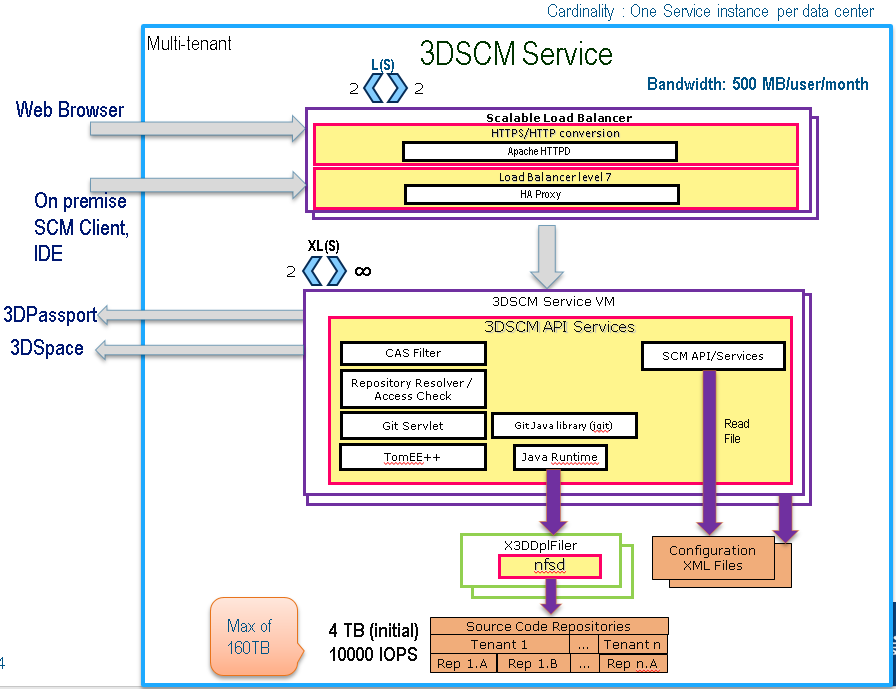
## Service installation, products, provisioning etc.

Installation and setup of a “Service” is not something that can be performed manually in the same way that individual products, or even the whole 3DSpace service, can be installed.

For internal development and QE, “PODs” and “Sandboxes” must be used.

What actually happens when a service is “deployed” (VERY roughly) is that a set of Virtual Machines (VMs) are created which have various products and tools installed on them. This generally includes one or more Tomcat servers that then provide the external visibility into that “Service”.

In the case of the 3DSCM Service, the following is the *initial* architecture of the service.



What this pictures describes is:

* The 3DSCM Service (outer blue box) made up of…
* The “Scalable Load Balancer VM”: This is a standard component that provides a server with SSL access and reverse-proxy capabilities (just like on our 3DExp4All servers). All service calls go through this. The purple boxes around this indicate that it is “Active/Active”, and the “2 < > 2” means that there are exactly two instances of this VM. The idea is that calls can go to either of these VMs. From the server in this VM, the calls get sent to…
* The “3DSCM Service VM”: This has a single “product” installed on it (at this time), which is a modified version of the “Git Connector” that we use for our “External SCM” support, with the WAR file from that installed on a Tomcat server. Also on that same Tomcat server is the “git servlet”, which is the component that handles the requests that come directly from a git client. Again, the purple boxes around this VM indicate that it is “active/active”, which means that there are initially two copies of this VM, each with the connector installed, and more VMs can be added later if the server becomes heavily loaded. The Load Balancer handles passing of service requests calls to the different 3DSCM Service VMs. For disk storage, the Services infrastructure provides “Volumes” (disks) that can be attached only to a single VM. Since we need access to all the repositories from each of the 3DSCM Service VMs, we cannot directly access these “Volumes” from this service. Therefore, we also have...
* The “X3DplFiler VM”: This provides the “Volumes” (disks), and includes a NFS server. A NFS client is installed on each of the 3DSCM Service VMs, which then mounts the disks exposed from the NFS server on the X3DplFiler VM, and thus makes the disk space available to the 3DSCM Service programs.  
  Note that there are limits to the size of a “Volume” and to the number of Volumes that can be attached to the X3DplFiler VM, and therefore there is an overall limit to the total amount of disk space this architecture can provide. => If we start reaching this limit, then we will need to add additional X3DplFiler VMs.  
  The green boxes around this VM indicate that it is “active/passive”. This means that there are two VMs initially set up BUT only one of them is “active” at a time, and that active one has the Volumes attached. If the “active” VM is found to be “down” for any reason, then the “passive” one can be quickly brought on line, the disks attached to that, and the service recovered. The “passive” VM becomes the active one, and a new “passive” VM can then be deployed. Remember that the Volumes can only be attached to one VM at a time, which is why this VM is active/passive and not active/active.

Thus, both the calls we make from the Connected Software web app (to see the File Tree and fetch file contents) and the calls made from a git client (to clone/pull/push the repository) go through the Load Balancer / Reverse Proxy, to the 3DSCM Service VM, and ultimately access the repositories on disks provided by the X3DplFiler VM.

**Modified Git Connector and Packaging**

As noted above, to handle the requests coming from the Connected Software web app we use the same “Git Connector” as used for our “External Git Connector” product. This makes sense, since they both have the same job to do! However, there are a number of differences in the two products. For example:

* For 3DSCM, the built-in TomEE and JRE are never used, so do not need to be included
* For 3DSCM, the “configuration” web app is not needed [Configuration is “automatic”, as described later.]
* For 3DSCM, the way in which the “Repository Path” is used is different
* For 3DSCM, the value returned when a request is made for the “direct URL” of a repository is different.
* For 3DSCM, there is only the Git SCM option at this time, not one for DesignSync

It is possible (even likely) that separate packaging may be required for the modified product (instead of “SCM\_Connectors”: the current media kit that includes the Git Connector and DesignSync connector.) This will not impact end users, or even be visible to anyone, but we will record in here the details of the packaging changes once known.

**Licensing**

Licensing is yet to be defined. For the original SCM Connectors, we did not implement DSLC licensing, but we will need to consider this for the new service.

Ideally, what should happen is:

* All REST services (called from the Connected Software app, for example), which perform a modification of data should check for a license
* All calls from a git client that perform a modification of data (i.e. all “git push”) should check for a license

Note that normal practice is to check for licenses only on modification operations.

The exact license to be checked is also to be defined.

NOTE: As part of our Access Model, modification operations on a repository in 3DSCM result in a modification of a Software Item (see later), and the service call to 3DSpace that performs this modification will itself perform a License check, requiring the XSF license (or some other license that provides the Option we check for.)

**Scalability**

As the number of users of this service increases, the amount of data stored in the repositories will increase and the amount of “traffic” to the services will also increase.

The system must be designed to be “scalable”, meaning that the disk space increases according to need, and the servers can handle the amount of “traffic”.

As noted above, the 3DSCM Service VMs are “active/active”, with the load balancer responsible for sending services requests to those VMs. New 3DSCM Service VMs can be added (and removed!) as load demands.

For disk space, it is more complex. As previously noted, there is a limit to the size of the individual “Volumes”. This is 8TB, or possibly less depending on the Cloud service provider. If the total data across all the repositories we are “hosting” on the service (meaning all repositories across all tenants in a “region”) starts to approach this size, then we can add additional Volumes to the X3DplFiler VM. But, again, there is a limit to the number of Volumes that can be added. This limit is about 20 (again depending on Cloud service provider.) Thus we have an absolute top limit of disk space of about 160TB. It may also be that a single NFS server on the X3DplFiler VM cannot handle the traffic involved in providing access to these volumes.

If we find that either there is a bottleneck in the NFS access, or we start reaching the limit on Volumes, then the current solution will be to re-deploy the entire service with an additional X3DplFiler VM (or, rather, pair of active/passive VMs.)

Alternatively, in the future, we may have a difference form of storage available to us that provide completely scalable storage available across multiple VMs. If that is the case, then the X3DplFiler VMs would be replaced with that storage mechanism.

## Configuration

### Configuration Web App

The “Adapter” and “Git Connector” provided for “External SCM” connections includes a Web App that provides the means to configure the application.

This web app will NOT be provided with the 3DSCM service, for the following reasons:

* We really do not want to require Cloud users to perform this kind of configuration
* It is not currently licensed or access controlled
* The design does not meet 3DS UI design requirements
* The interface is not “tenant aware”

As the web app is not provided, it is reasonable to ask how each option is handled with this service, and the following tables describes this.

**SCM Adapter Configuration Panel**

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | **Replacement** |
| Enable Logging | Used to turn on logging of, primarily, the “notification” handling. | This form of “notification” will no longer be supported, so this logging is not needed. |
| Connected Software Service URL | The URL of the 3DSpace server, used for notifications. | This value is no longer needed, so will not be set. |
| SCM Adapter URL | The URL of the 3DSCM server itself. Used for CAS purposes. | This value will be automatically set on installation. |
| 3DPassport Service URL | The URL of the 3DPassport server. Used for CAS purposes. | This value will be automatically set on installation. |
| 3DPassport Username / Password | A generic, valid, un/pw. Used to authenticate when notifications are processed and passed through to 3DSpace. | No longer needed as notifications will no longer be sent to 3DSpace using the previous system. A new system for notifications, using a generic “messaging” system may be implemented in the future. |

**Git Connector Configuration Panel**

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | **Replacement** |
| Temporary Workspaces Path | Used to temporarily clone workspaces when required for certain operations. | This will be automatically set to an appropriate place for the server. |
| Git Repository Path | Each git repository (path) has to be “registered” before it can be accessed. | This process will no longer be required. |
| Git Tools Path | The path to the git tools. Per-repository, in case a different version is required. | The git tools are no longer used, as they are replaced by jgit (the Java-based implementation of git) which is included in the product distribution. |

### Repository hooks

The on-premise solution installs a pair of “triggers” into each Git repository as it is “registered”, in the form of pre-receive and post-receive hooks.

With no registration now taking place, these triggers cannot be installed.

In addition, the “git servlet” that handles the requests from a git client would not fire these triggers anyway.

Therefore, a replacement system is needed:

Pre-receive hook: This was used as part of the branch locking system, to check for a lock before allowing a “push” operation to take effect. This will be replaced with a customized java piece of the git servlet, and the same customized piece in the “upload manifest” service that we have.

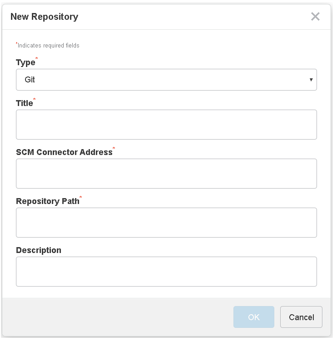
Post-receive hook: This was used to send the “notification”. As noted above, these notifications are no longer used. However, this was also used to clear the internal “cache” of the file tree manifest. A new system will be implemented that doesn’t rely on the trigger, but instead checks on each call whether a newer commit exists, and re-fetches the file tree if there is.

### SCMRepository Objects

The “SCMRepository” objects, which are created and maintained through the “Repositories” panel of the Connected Software WebApp, still exist, and there will be specific SCMRepository objects for the 3DSCM service.

These objects are still needed, as they indicate that a Software Item is managed by 3DSCM rather than by an external connector. Also, the connection to the SCMRepository object still holds the addressing information that identifies the specific repository and version that the Software Item is associated with.

The SCMRepository objects will still be created by the Project Administrator. These objects still need to be visible to the end users, and so need to be created by administrators so that they have appropriate access.



However, there are some differences with the SCMRepository objects used with 3DSCM (see the picture above for the current Create form):

* The “Repository Type”, which previously had choices of “Git” or “DesignSync” will now have an additional type that shows in the WebApp as “3DSCM” (though the actual attribute range value is “3DSCM\_Git”). Initially, there will only be the one additional type, for the Git repositories stored in 3DSCM, but in the future there may be more to differentiate between underlying SCM systems (e.g. one for Git and another for a future DesignSync-based solution.) [All the Repository Type strings are subject to change, and this will be outlined in the relevant PES.]
* The “SCM Connector Address”, which normally contains the URL of the “connector”, will now need to point to the 3DSCM service URL. This URL will **not** be exposed to the end users. When creating a SCMRepository of the “3DSCM\_Git” type, the admin will not enter this value (the field will not be available, and the value will be set automatically.) And when viewing such SCMRepository objects’ property pages the value will not be shown.  
  This SCM Connector Address is not completely “hidden”: There may be other interfaces, such as generic property pages, which show the value, and that is OK.  
  Technically, the SCMRepository object will still store this address, but it will be identified automatically, either by the WebApp itself, or by the back-end services code (depending on some technical constraints.)
* The “Repository Path”, which normally limits the set of repositories that a SCMRepository can access, is not applicable for 3DSCM, as the 3DSCM system itself will decide where the repositories are stored. When creating a SCMRepository of the “3DSCM\_Git” type, the admin will not enter the repository path either (the field will not be available.) And, similarly, the value will not be shown on the Properties panel. Again, the value may appear on other panels in the wider system, but it will be “”.  
  Technically, what this means is that the “Item Address”, recorded against the individual Software Item, will contain the full address of the repository, or, rather, as much as is needed by the 3DSCM system to find the repository.

## User Access to Repositories

In the “External Connectors” system, it is up to the users to create the repositories (in git or DesignSync), wherever they should be on disk. The user also sets up the addressing on the Software Items to those repositories. And when the user wants to work on the data they “know” where their repositories are and specify them to the SCM client commands (git clone or DesignSync populate.)

In the 3DSCM system, the users cannot manually create the repositories.

Instead, new capabilities will be provided through the Connected Software WebApp to create a repository and associate it with a Software Item (version.) The details of this will be given in the UI PES, but essentially when creating a Software Item, or from the “Repository” tab, the user will be able to request a 3DSCM-type SCMRepository, and the git repository will then be created automatically. The “address” of the new repository is then also automatically associated with the Software Item.

As with the addresses on the SCMRepository items, the Item Address will not be shown to the user, as it is not something they need to know.

The existing capability to “Edit” The Item Address is being removed as part of this functionality: To change an Item Address, a user must disconnect the Repository (another new capability) and create a new connection. This is to better support several scenarios. See the relevant PES for more details.

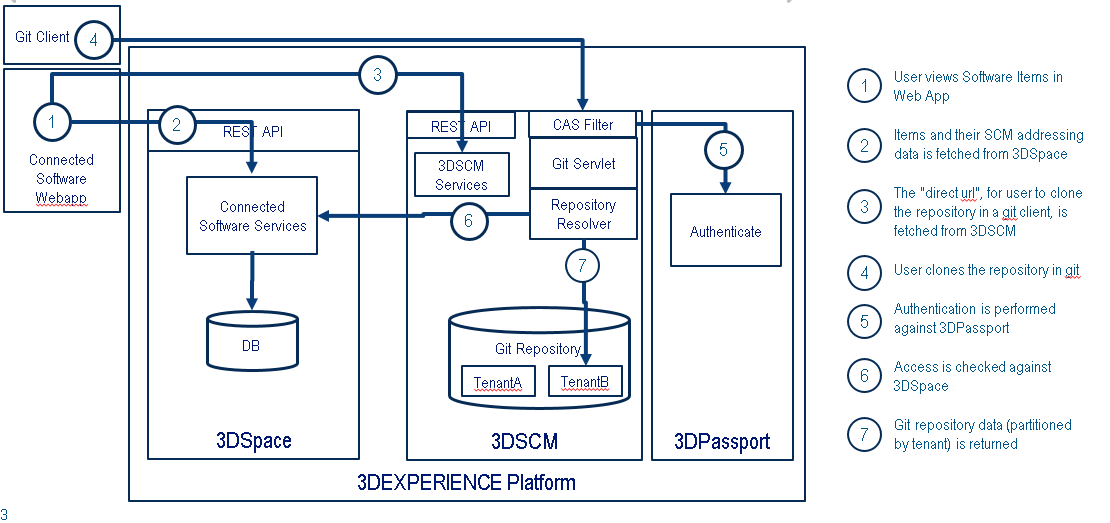
So, the Item Address is not shown, and neither is the “SCM Connector Address”. How does the user then find the repository in order to clone it from a git client? The answer is that users must always use the “Repository Path” command from the Connected Software WebApp to get the URL that is then used with the initial “git clone”. Remember that fetching this URL is a one-time operation: Once the repository is cloned, the URL is “remembered” by git for subsequent push/pull operations.

**Important:** The URL that is given can only be used to access the repository by users that have access to the Software Item for which it was generated. See the access and authentication section for more details.

**TECHNICAL NOTE:** Programmatically, the addressing values from the SCMRepository object and the Software Item are still available via the REST service to GET a resource, using an appropriate mask. And the URL for “git clone” is available using a service provided by the 3DSCM service (and also provided by the External Connector services.)

## Authentication and Access

User authentication and repository access is granted following a process shown in the following picture, and further described in the items below that reference this picture.



### User Authentication

All access to the 3DSCM system passes through the standard 3DPassport (CAS) authentication (5).

When the Connected Software WebApp (1) is used, and that calls services from 3DSCM, such as that to get the repository URL (3), the user is authenticated as part of “single sign-on”. That is, the authentication performed to access the WebApp also provides the authentication to the 3DSCM system. This all happens “behind the scenes”, without the user being aware of it (not shown in the picture above.) [Technically: This uses the WafData system to send the requests, and that takes care of handling the special response that causes it to generate a CAS proxy-ticket and use that for authentication.]

When a git client is used to perform an operation such as cloning the repository (4), the following (logical) process happens:

* The 3DSCM server provides a response to the git client indicating that “basic authentication” is in operation, which means a username and password are required
* The git client obtains these credentials from the user. Either by directly asking for them, or by fetching them from a credentials cache, according to the git client and its configuration. The exact form in which this is requested from the user will depend on the client application that is in use. For example, on windows the Git windows credentials manager could be in use, which would pop-up a form showing the URL of the repository and asking for the username and password.
* The git client sends the username and password (SSL encrypted) to the 3DSCM server
* The server authenticates those with the 3DPassport system (5)

NOTE: It is possible to enable “2-phase authentication” on the 3DPassport server. This is the typical system whereby an authentication system sends a user a code by a separate means, such as an SMS (text) and the user has to enter that. It is also possible for other advanced authentication systems to be used. However, the git client only supports “basic” username/password authentication. So, how will this work? The 3DPassport system is to be enhanced with a system for storing an “application specific” password, which acts as a simple password and can be used for this authentication. (Timeframe for this enhancement is TBD.)

### Repository Access

Just because a user can authenticate with 3DSCM does not mean that user should also have access to the repositories.

There is therefore an additional layer of “access control” applied to the repositories.

* When the user requests the URL for cloning a repository (3), the request sent to the 3DSCM service includes the user’s current “Security Context” and the “resource URL” of the Software Item for which the request is being made.   
  This “resource URL” is a URL on which we can later perform certain REST calls. For 3DSpace objects, it is the URL that can be used to perform a GET request or PATCH request on a software item, as defined in the Web Services PES, so it is something like:  
  <https://host:port/3DSpace/resources/v1/modeler/dssoft/dssoft:SoftwareLogicalItem/00124AB?$mask=dsmvsoft:ItemVersionMask.access>
* The URL that is returned includes within it not only the main URL of the 3DSCM service, but also this Security Context and “Resource URL”.
* When the user performs a git operation, such as “clone”, a request is made by git using this URL (4)
* The “Repository Resolver” system takes this URL, extracts the Security Context and Resource URL and makes a call to that Resource URL to check the user has access to the object id specified (6)
* If the user does have access, then the “Item Address” of the Software Item is returned (by the mask specified)
* Only if the access passes will the user then have access to the repository and the data be cloned (7)

This process ensures that a user can only access the repositories they “should have access to”.

Effectively, this system “defers” the repository access to the 3DSpace objects: You must have access to the 3DSpace object to access the repository.

In the future, this system can be extended to other “services” that want to use 3DSCM for managing their own repositories, and which have their own equivalent to the 3DSpace “business objects” that will reference the repositories: All those services have to provide is their own “Resource URLs” that can be called to check access and return the actual “Item Address” of the repository.

The actual access required on the 3DSpace object depends on the operation that is being performed:

* A “clone” or “pull” or other read-type operation requires “read” access to the Software Item
* A “push” or other write-type operation requires “modify” access to the Software Item

It is important to note that the READ access does not check the *version* that the Software Item is associated with. What this means is that if a user has read access to the Software Item, then that user can read the whole repository, since “git clone” will always clone the entire repository. However, a MODIFY operation (i.e. a git push) WILL check that the data being pushed is to the branch identified by the “Version Address’ of the Software Item, and will disallow the operation otherwise. What that means is that from a workspace that has cloned the repository using a Software Item associated with Branch A, it is not possible to push data for any other branches. The read access approach is a limitation of the git system, since the entire repository gets cloned. This access *may* be implemented differently, and made more “strict” in the future when a DesignSync-based system is implemented.

### Tenant-based Access

As previously noted, individual customers (companies) that purchase our solutions are known as “tenants” and the same “service” can support multiple “tenants”. For example, on cloud there is (as we understand it) only ever one 3DPassport service in a particular data center which supports all customers.

When a service supports multiple tenants it is absolutely vital that all data is completely partitioned so that it is never possible for one tenant to access the data from another tenant.

The 3DSCM service is “multi-tenant”. This means that the repositories must be carefully managed to ensure that no tenant can access the repositories from another tenant.

How is this achieved?

There are two levels to it:

1. All repositories are physically stored in a directory path that includes the “tenant id”. For example, if the tenant id is “ten123” and the root of repositories is “/data” then all repositories for that tenant are under /data/TEN123. (There is more to this path: see the section below.)  
   All requests include the tenant id. (Note that the tenant id is always converted to upper-case, to avoid issues with mixed case.)  
   The “item address” stored on the Objects in 3DSpace, and returned when the access check is applied, is only the part of the path under the tenant area: it doesn’t include the tenant id itself.  
   The repository that is requested can thus only be accessed if it is within the path identified by that tenant id
2. The object id that is checked in 3DSpace is also per-tenant. So, access to that object in 3DSpace cannot be obtained by a user from another tenant

If a customer (tenant user) somehow got hold of the URL for a repository from another customer (tenant), and they were able to authenticate with 3DPassport, they would still not be able to clone the repository because a) that repository is not in the directory areas under their tenant id and b) they would not pass the access check for the object in 3DSpace (which is not available in their tenant).

### Physical Storage Paths and Item Addresses

The actual physical storage path for a repository needs to take into account a number of factors. These paths are completely under the management of the 3DSCM service, and will never be exposed to the end users.

The format of the paths is best described by giving an example path:

**/opt/data/sda/TEN0001/dssoft/AA/myitem**

The pieces of this path are as follows:

**/opt/data**: This is the root of all stored data in the service. The actual path will be determined during implementation.

**/sda**: This is the “mount point”. We may have multiple “Volumes”, which are mounted disks, or NFS shared disks, each with their own path.

**/TEN0001**: This is the tenant identifier. So, all data for a specific tenant (on a particular Volume) is under a single directory. The tenant ID is converted to upper case to avoid issues with URLs sometimes having lower-case tenant IDs.

**/dssoft**: This is the “service identifier”. Repositories created using the Connected Software system will use dssoft, other services could use other names. See the section on “spoofing” of addresses below for why this is important.

**/AA**: A two-hex-character value that is used simply to spread out the repositories (like a sea-of-vaults structure)

**/myitem**: The actual repository directory.

The ‘Item Address’ that is returned by the ‘Create Repository’ service, and then stored in 3DSpace against the ‘Software Items’, and returned by the ‘access check’ is then a *portion* of this path, something like:

**/v1/sda/dssoft/AA/myitem**

The pieces are:

**/v1**: A version value identifying the addressing scheme version. If we need to change the way addresses are used in a later release, this will allow for backward compatibility.

**/sda/dssoft/AA/myitem**: The same pieces of the physical path as described above, but notice that the “root” and “tenant identifier” are not included. The root is under the control of 3DSCM, and a fixed value so does not need including. The tenant identifier is not included: it is extracted from each service request made, making it impossible to request a repository from a different tenant (even if the other elements of the path could be guessed!)

### Access Spoof Prevention

Access to the repositories is based on the access a person has to the ‘Software Items’. There is a danger that a “spoofer” (with appropriate access) could create their own Software Item, and “copy” the Item Address from another item, and thereby get access. *How do we stop this?*

First, the Web App GUI does not provide a way to simply set the Item Address of a software item connected to a 3DSCM repository, so that could not be used to copy the address.

*The spoofer could directly call our REST service that sets the Item Address, couldn’t they?* No. This REST service will ONLY allow the Item Address to be set for repositories that are being managed by the External Connectors, and not those managed by 3DSCM. In addition, this API will check that the address passed in is not being used by any other Software Item.

*So, how does the Web App manage to create a repository and set the Item Address then?* What happens is that the web app calls a service in 3DSpace to create the repository, and 3DSpace then itself calls 3DSCM to create the actual repository, gets the address back, and stores that (new repository) address on the Software Item. Because of this flow, there is no code that will set the Item Address except in this “new repository” flow, which can only set it to the new repository, not to one that already exists!

*Could the spoofer simply change the Item Address in MQL or some other process that can modify the relationship attribute where we store this?* No: In the Cloud world, users do not have MQL access, and in addition we have a process in place that prevents the Item Address being set in any way except through our own code!

There is one further case: In the future, we might have two different services that are managing data in 3DSCM, for example we might have the Software Items in 3DSpace and ‘Semiconductor Items’ in some other service. *Could a spoofer find the Item Address in one of these services, and apply it to an object in the other service?* No: The ‘service name’ piece in the Item Address, as documented earlier, will prevent this. If an attempt is made to set the ‘Item Address’ to a value that does not include the correct ‘service name’ for the service, that will fail. (This will be implemented in the 3DSpace service that sets the Item Address, and will be required to be checked on other services that use 3DSCM.)

### 3DSCM Web Services

As well as accessing the repository using a git client application, we also provide access to the repositories through the Connected Software Web App. This Web App works by making REST service calls to 3DSCM, which are the same calls made to our external connectors.

*How does authentication and access work for these REST services?*

For authentication, these services all use the standard 3DPassport system, so there is nothing new there for the 3DSCM service.

For access, the services will be changing. In previous releases, these services all took a ‘Repository Path’ and ‘Item Address’, which together identify the actual repository, and the services simply operate on that repository with no further access checking. This means that a user could potentially get access to any repository. It is not quite that simple, as the way the services authenticate makes it impossible to call the services through something like Postman or YARC. However, it would certainly be possible to ‘hack’ the requests and get access to other repositories.

*So, how do we close this access hole?*

The REST services will change so that they work in a similar way to the git client requests. That is, rather than taking the ‘Repository Path’ and ‘Item Address’ they will take a ‘Resource URL’, as defined previously, which is a URL that will be called to check access and which will then return the actual Repository Path / Item Address. This gives consistency of access control between the two methods of accessing the repositories.

Note that this modified system and addition access control will be implemented both for 3DSCM and the external adapter/connectors.

*Could I set up my own service and pass in a fake Resource URL?*

This question actually applies to both methods of accessing the repository. Preventing this kind of attack is not so simple. However, the following step will be taken to prevent it: The ‘Resource URL’ specified must be to the same *domain* as the 3DSCM/connector service itself or must match a ‘white list’ of domains specified in a configuration file. On-cloud, the 3DSCM service will always be in the same domain as the service providing the resource URLs (3DSpace), and it will not be possible for an attacker to install their own services on that same domain. For the connectors, the 3DSpace service *could* be in a different domain, in which case the white list can be modified to allow for that, and it will be up to the administrators to ensure that it is not possible for an attacker to install their own services.

### Caching Authentication and Access

When a git command, such as a pull or push, is performed, there are two separate service calls made to the server. To avoid the need to authenticate the user twice and to check the access against the Software Item twice, a level of caching will be implemented. In addition, when a successful call to 3DSpace is made, a “JSESSION” id is returned, which can be used for subsequent calls without having to go through full authentication and session set up, and this session id can also be cached.

So:

* When the first service call is made, the user is authenticated with 3DPassport, which gives us a “ticket” to get into 3DSpace
* When we then perform the access check against 3Space, we get back the session id
* At this point, we can cache both the fact that we have the access (read or modify) and the session id
  + These are separate in-memory caches
  + The cache of the access is *very short term* (a small number of seconds, to be determined, maybe 10). If another request comes in to the server within that short time, then we will use this cache, and if there is a match that provides the right access we do not have to make another call to 3DSpace to re-check the access. This short time is enough to handle the 2 requests that git makes for a command.
  + The cache for the session id is long term. The session id can be re-used until such time as the 3DSpace system refuses the session id, which is generally a much longer time. When the user performs a subsequent operation, we can re-use the session id, try to call 3DSpace, and if that fails because the session has timed out, then we have to re-authenticate with 3DSpace and try again.

These caches will be used both for the calls from git clients and calls to the 3DSCM REST services (used by the web app for operations such as viewing the File Tree.) For the REST Services, the short term access cache will probably not be of use, as the calls are unlikely to come that fast, but the session cache will certainly be useful and will provide a good performance gain as the time to set up the session in 3DSpace is significant.

### CSRF

Cross-site request forgery is a security risk whereby the following process happens:

* A user is authenticated, within a browser process, against server A
* The user is sent an email or similar with a link in it
* The user clicks the link, and it is actually a call to server A, which performs some malicious operations. The call manages to go through because the user is already authenticated with the server.

CSRF is only an issue for operations that can modify data. If the call only reads data, then there is no security risk: the user may just see that data, it won’t go back to the original caller. (This is the standard CSRF view.)

For 3DSCM, CSRF is not an issue for the git client commands that modify data (git push), as each call requires the username/password to be passed, and a spoof URL would not have those.

However, for the REST calls made from the Connected Software Web App, CSRF must be taken into account.

The currently planned solution (to be confirmed) is as follows:

* The Web App already has a CSRF token for the 3DSpace system, which is used to protect the calls to 3DSpace
* This same CSRF token will be passed to the 3DSCM calls that perform modifications.
* Those calls will then always call to 3DSpace to perform the authentication check, and will pass the CSRF token into that call
* That authentication check call will itself check the CSRF token, and thus fail if no token is given or the token is invalid.

## Persistent Storage

A Service is implemented as a number of “Virtual Machines” (VMs) which have running on them, the various programs and servers, and which may also have the physical storage.

For our 3DSCM service, we clearly need to have physical storage for the git repositories, and this storage needs to be available to the server that is handling the requests both from our web app and the git clients.

We also need a service that provides high availability and scalability. This essentially means that we need to be able to support the growing disk space needs as more customers manage more data, and we need to be able to support the growing number of requests that will result from that.

For the disk space, the topics we needed to resolve are the following:

* How the disks are managed, how they are backed up etc. => This is part of the ‘service definition’ (poddef) for the service. This includes backup/restore capabilities, plus systems for monitoring and reporting issues.
* How additional disk space is provisioned and increased over time => Additional Volumes can be added, but only a certain number, as describe earlier. The additional volumes are accounted for in the addressing scheme identified previously. The fact that there is a physical limit to the total disk space available is something that will continue to be discussed at the service definition level: Resolution for this will require a future extension to the service capabilities. Either by the addition of more “Filer” VMs, or by implementing some other completely scalable storage mechanism.
* How the repositories are partitioned amongst the disks, allowing for the need to partition data from different tenants => This is covered in the section on physical storage paths described above

### Repository Information File

As each repository is created, a file will be placed in the repository at:  
*<repositoryPath>/*info/3DSCM.xml

This file will be used to store any per-repository information that is needed by our processing.

This file will contain a version number, and initially will contain information from the Security Context that is specified when the repository is created. A field containing the whole of the Security Context value will be added. In addition, if the Security Context value matches the standard 3DS form of “ctx::<role>.<company>.<collab space>”, then additional fields will be added with those elements. So, if the Security Context is “ctx::VPLMProjectLeader.MyCompany.Common Space” then the file contents will be something like:

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<tenantMetadata>

<version>1</version>

<securityContext>

ctx::VPLMProjectLeaader.MyCompany.Common Space

</securityContext>

<role>VPLMProjectLeaader</role>

<company>MyCompany</company>

<project>Common Space</project>

<originalPath>

/opt/data/shared/tenants/WALVCL51MON/2a/repoName

</originalPath>

</tenantMetadata>

An example of the use of this data is that we may have a usage probe that reports disk usage per “CollabSpace”, per tenant.

The file also contains the original storage path of the repository. This may be useful if a repository has to be moved at a later date.

## Runtime monitoring and Supervision

We need to provide support to the people that manage the cloud installations (the Cloud Run Operators). This is in the form of “probes” that monitor the “health” of the service (whether the service is running, how it is performing, how much space is being taken up etc.) and scripts that aid the runtime management.

Details of this will be developed as we proceed.

Another area we are likely to need to provide is the means to monitor the usage (CPU / IO and disk) for individual tenants, since this impacts the charges made to the customers.

This section complements the Capability Summary provided above by detailing all aspects of the functionality that will be exposed to the user.