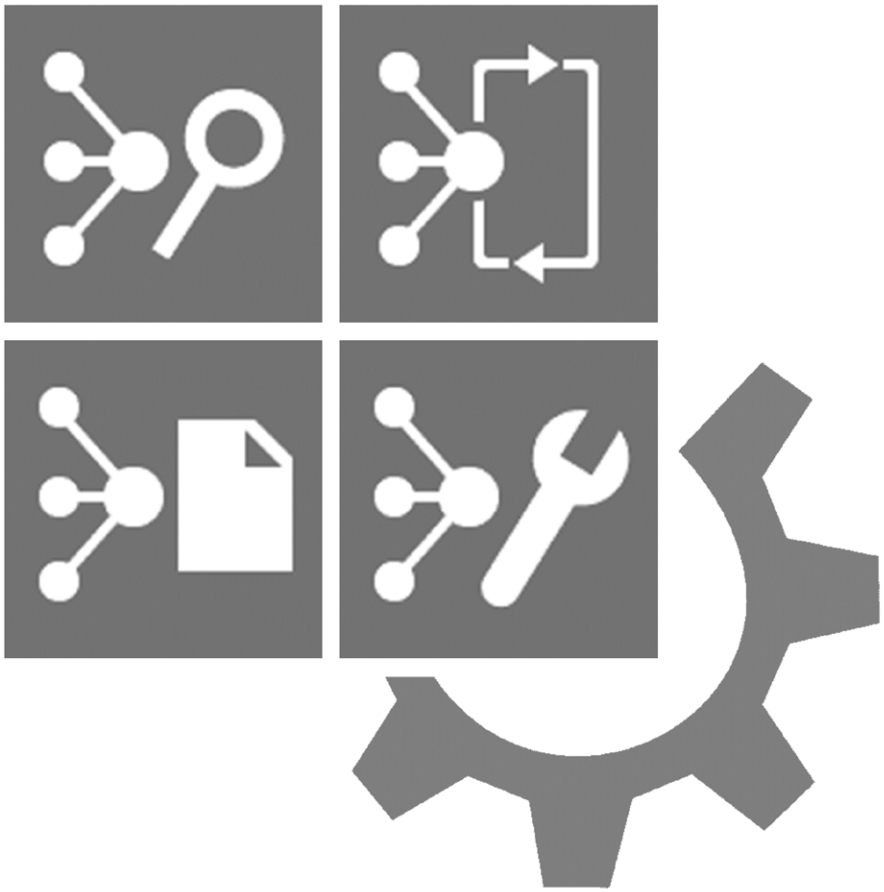
RENISHAW
apply innovation™

Manuel du développeur de DataHUB



www.renishaw.fr/additivemanufacturing

#renishaw

Cette page est laissée vierge intentionnellement.

RENISHAW
apply innovation™

Sommaire

1 Avant de commencer

1.1 Conditions générales et garantie

Sauf accord écrit séparé, signé entre vous-même et Renishaw, le matériel et/ou le(s) logiciel(s) est/sont vendu(s) conformément aux Conditions Générales de Renishaw (« Renishaw Standard Terms and Conditions ») fournies avec le(s)dit(s) matériel(s) et/ou logiciel(s), ou disponibles sur demande auprès de votre bureau Renishaw local.

Renishaw garantit son matériel et ses logiciels pendant une durée limitée (comme stipulé dans les Conditions Générales), à condition que ceux-ci soient installés et utilisés dans le strict respect de la documentation Renishaw qui leur est associée. Pour connaître tous les détails relatifs à votre garantie, vous devez consulter ces Conditions Générales.

Tout matériel et/ou logiciel acheté par vous-même auprès d'un fournisseur tiers est/sont soumis à des conditions distinctes fournies avec ledit matériel et/ou logiciel. Pour obtenir plus de détails, veuillez contacter votre fournisseur tiers.

1.2 Modifications de l'équipement

Renishaw se réserve le droit de changer les spécifications de l'équipement sans préavis.

1.3 Brevets

Les caractéristiques de la machine de fabrication additive de Renishaw et d'autres systèmes semblables sont protégés par l'un ou plusieurs des brevets suivants et/ou font l'objet de demandes de brevet :

1.3.1 Gamme RenAM 500 (modèles Q, S et Flex)

CA 2738618	EP 2331232	IN WO2014/125258	US 10335901
CA 2738619	EP 2875855	IN WO2014/125280	US 10493562
	EP 2956261	IN WO2014/199134	US 10500641
CN 102186554	EP 2956262		US 10639879
CN 105102160	EP 3007879	JP 6482476	US 10933620
CN 105228775	EP 3221073	JP 6571638	US 10974184
CN 105492188	EP 3221075		US 11033968
CN 107107193	EP 3299110		US 11040414
CN 107206494	EP 3323534		US 11104121
CN 107921659	EP 3325240		US 11267052
CN 108189390	EP 3357606		US 11305354
CN 108349005	EP 3377252		US 11478856
CN 108515182	EP 3377253		US 11565346
CN 109177153	EP 3566798		US 8753105
	EP 3689507		US 8794263
	EP 4023387		US 9114478
			US 9669583
			US 9849543
			US 2020-0023463
			US 2021-0354197
			US 2022-0203451
			US 2023-0122273

1.3.2 DataHUB

CN 109937101	EP 3482855	US 11167497	WO 2020/099852
CN 111315512	EP 3538295	US 2020-0276669	
CN 112996615	EP 3880391	US 2021-0394272	

1.3.3 InfiniAM Spectral

CN 105745060	EP 3049235	US 10850326	WO 2020/099852
CN 108349005	EP 3377252	US 11305354	WO 2020/174240
CN 109937101	EP 3482855	US 11040414	
CN 110026554	EP 3482909	US 2020-0276669	
CN 111315512	EP 3538295	US 2021-0039167	
CN 111491777	EP 3880391	US 2021-0394272	
CN 112996615	EP 3930999	US 2022-0168813	
CN 115943048	EP 2020-174240	US 2022-0203451	

2 Introduction

2.1 Contenu de la livraison

Ce document sert de guide pour créer, tester et déployer un composant logiciel plug-in pour DataHUB. Consultez le manuel d'utilisation de DataHUB (Réf. Renishaw H-5800-6851) pour une présentation et une explication des fonctions du logiciel DataHUB. Le cycle de développement du plug-in commence par l'installation de DataHUB sur le PC de développement, puis par la création (à l'aide de Visual Studio) d'un assembly .NET contenant le code du plug-in. Grâce à la configuration de débogage Visual Studio appropriée, le plug-in peut être testé et débogué pour vérifier son efficacité avant son déploiement final sur un PC de collecte de données (PCCD) connecté à un système AM de production.

2.2 Abréviations

Acronyme	Définition
AM	Additive Manufacturing (Fabrication additive)
AMPM	Additive Manufacturing Process Monitoring (Surveillance du procédé de fabrication additive)
API	Automate programmable industriel
DEEE	Déchets d'équipements électriques et électroniques
IHM	Interface Homme Machine (écran tactile)
OEM	Original Equipment Manufacturer (Fabricant d'équipement d'origine)
PC	Ordinateur personnel
PCCD	PC de collecte des données
REACH	Registration, Evaluation, Authorisation and Restriction of Chemicals (Règlement concernant l'enregistrement, l'évaluation, l'autorisation des substances chimiques, ainsi que les restrictions applicables à ces substances)

2.3 Consignes de sécurité contenues dans ce manuel d'utilisation

Dans le cadre de ce manuel d'utilisation, les informations complémentaires qu'il est important de lire et de comprendre seront mises en évidence par les mentions « Avertissement », « Attention » et « Remarque ». Vous trouverez ci-dessous leurs définitions accompagnées d'exemples.

2.3.1 Avertissement

Exemple d'avertissement :

AVERTISSEMENT : un avertissement a pour but d'indiquer à l'utilisateur qu'il risque de se blesser ou de blesser d'autres personnes à proximité si la procédure décrite n'est pas respectée.

2.3.2 Attention

Exemple d'appel à l'attention :

ATTENTION : un appel à l'attention a pour but d'indiquer à l'utilisateur qu'il risque de détériorer les équipements si la procédure décrite n'est pas respectée.

2.3.3 Remarque

Exemple de remarque :

REMARQUE : une remarque a pour but de signaler à l'utilisateur des informations importantes qui sont liées à la tâche ou à l'activité ou qui l'aideront à les effectuer.

2.4 Programme de formation

Renishaw fournit un niveau de formation fondamental pour utiliser DataHUB en toute sécurité. Renishaw offre également des stages de formation avancée pour les opérateurs et les ingénieurs de procédés. Veuillez consulter le manuel d'utilisation de DataHUB (Réf. Renishaw H-5800-6851) ainsi que le guide de formation fourni dans le cadre de la formation des utilisateurs que tous doivent suivre avant d'utiliser DataHUB.

2.5 Documents de référence

En plus de ce manuel d'utilisation, veuillez également vous reporter aux documents suivants pour obtenir davantage d'informations concernant d'autres aspects du système InfiniAM® et du système AM Renishaw.

- Guide d'installation du système de fabrication additive RenAM 500Q/S (Réf. Renishaw H-5800-4369)
- Manuel d'utilisation du système de fabrication additive RenAM 500Q/S (Réf. Renishaw H-5800-4370)
- Guide d'installation des logiciels InfiniAM® et DataHUB (Réf. Renishaw H-5800-6843)
- Manuel d'utilisation de DataHUB (Réf. Renishaw H-5800-6851)
- Manuel d'utilisation d'InfiniAM® Spectral (Réf. Renishaw H-5800-6839)
- Manuel d'utilisation d'InfiniAM® Camera (Réf. Renishaw H-5800-6847)
- Manuel d'utilisation de Renishaw Licence Manager v2.x (à télécharger via le lien inclus dans l'e-mail d'octroi de licence InfiniAM)
- Plug-in **Export Packets** (Export des paquets) (Réf. Renishaw 5800-6788)

3 Pièces de rechange

DataHUB ne comporte aucune pièce réparable par l'utilisateur. En cas de dysfonctionnement de DataHUB, la réparation se fait par réinstallation et configuration du logiciel.

Reportez-vous à la section 4, « Coordonnées », pour obtenir les coordonnées de votre revendeur local Renishaw et pour organiser une visite d'entretien.

Les logiciels InfiniAM et DataHUB feront l'objet de mises à jour régulières. Tous les utilisateurs détenteurs de l'abonnement pourront télécharger la dernière version du logiciel à partir de leur compte MyRenishaw.

Cette page est laissée vierge intentionnellement.

4 Coordonnées

N° de téléphone	+33 1 64 61 84 84	
Heures d'ouverture	Du lundi au jeudi	De 08h00 à 17h00
	Le vendredi	De 08h00 à 16h00
E-mail	am.support@renishaw.com	
Adresse du service	Renishaw S.A.S. 15 rue Albert Einstein Champs sur Marne, 77447 Marne-la-Vallée, Cedex 2 France	
Type de système AM		
N° de série du système AM		
N° de version logicielle	Révision IHM	
	Révision API	
	Révision PC	
N° de série du matériel InfiniAM Spectral (module MeltVIEW)		
N° de version du logiciel InfiniAM		
N° de version du logiciel DataHUB		

Veuillez indiquer les renseignements ci-dessus. Des informations sur le système AM Renishaw sont disponibles sur la plaque signalétique à l'arrière du système AM. Les détails relatifs au module MeltVIEW figurent sur la plaque signalétique visible lorsque InfiniAM est installé sur le système AM Renishaw. Des informations sur le module matériel CameraVIEW sont disponibles sur la plaque signalétique à l'arrière de la caméra. Le module matériel CameraVIEW se trouve derrière un couvercle au-dessus de la chambre.

Pour obtenir une assistance supplémentaire, veuillez contacter votre revendeur local Renishaw. Consultez le site :

www.renishaw.fr/contacter

Cette page est laissée vierge intentionnellement.

5 Sécurité

5.1 Introduction

AVERTISSEMENT : toutes les informations relatives à la sécurité sont conformes au manuel d'utilisation et au guide d'installation du système AM Renishaw, sauf indication contraire dans le présent document.

AVERTISSEMENT : veillez à ce qu'un couvercle ou le module matériel MeltVIEW soit installé au niveau de l'ouverture du laser avant d'allumer le laser du système AM.

AVERTISSEMENT : le module matériel MeltVIEW n'est pas verrouillé au circuit de sécurité du laser. En cas de retrait du module matériel MeltVIEW, le laser peut se déclencher et une lumière laser nocive sera émise par l'ouverture laser sur le système AM.

5.2 Étiquettes d'avertissement sur le laser propres à DataHUB Generator

Aucune étiquette de sécurité ou d'avertissement supplémentaire n'est apposée sur le système AM à la suite de l'installation de DataHUB Generator.

Cette page est laissée vierge intentionnellement.

6 Fonctionnement

6.1 Introduction

Grâce à son architecture plug-in, la suite logicielle InfiniAM prend en charge l'analyse en temps réel des données de surveillance de procédé collectées par les plates-formes matérielles LaserVIEW et MeltVIEW. Un plug-in est une unité logicielle autonome installée sur un PC de collecte de données (PCCD) équipé de DataHUB. Lorsque DataHUB traite les données collectées d'une fabrication additive, il transmet les valeurs de surveillance de procédé aux plug-ins pour qu'ils effectuent des analyses sur mesure. DataHUB prend en charge plusieurs plug-ins fonctionnant simultanément, ce qui permet d'appliquer divers algorithmes et techniques d'analyse aux données de surveillance de procédé. Il est intégré au système Renishaw Central et permet aux plug-ins de présenter les résultats de l'analyse en même temps que d'autres relevés de capteurs de la machine effectués pendant la fabrication.

DataHUB prend en charge deux versions d'API de plug-in : la version 1.0 (V1.0) prend en charge les plug-ins qui traitent les données de fabrication résumées sous forme de paquets (voir la section « Échantillons en paquets » à la page 6-8) ; la version 2 (V2.0) prend en charge les plug-ins qui traitent les données de fabrication sous forme de flux de jetons (voir la section « Échantillons en flux de jetons » à la page 6-8). Il est important de comprendre que l'API V2.0 ne remplace pas l'API V1.0, mais qu'elle présente les mêmes données de surveillance de procédé aux plug-ins, simplement dans des formats différents. Le choix de l'API à utiliser doit être soigneusement étudié en fonction de l'analyse à effectuer : les paquets fournis par la version 1.0 sont beaucoup moins nombreux que les échantillons fournis par la version 2.0, mais cela peut être suffisant pour certaines tâches d'analyse qui ne nécessitent pas une telle finesse dans les détails des données d'échantillons.

Ce document détaille l'architecture du système de plug-ins, le cycle de vie d'un plug-in et d'autres aspects du système qui s'appliquent aux deux versions de l'API.

Le plug-in **Export Packets** (Export des paquets) permet d'écrire un fichier délimité par des tabulations contenant un résumé des données de surveillance de procédé recueillies par les modules matériels LaserVIEW et MeltVIEW au cours d'une fabrication. Ce plug-in est fourni avec DataHUB et ne nécessite pas de licence supplémentaire.

6.2 Architecture

Les relations de haut niveau entre les éléments matériels et logiciels du système sont illustrées dans la Figure 1 :

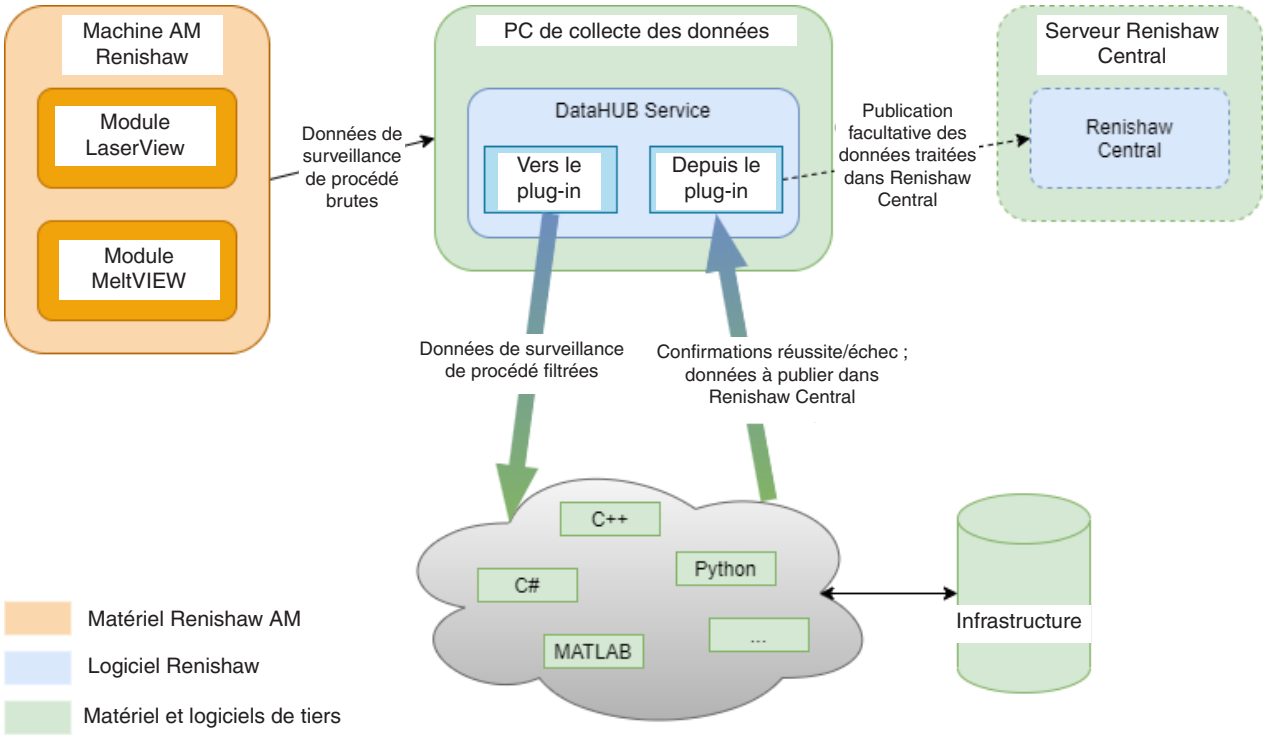


Figure 1 Architecture de plug-ins de DataHUB

6.2.1 Flux de données

Lorsqu'une machine AM Renishaw équipée de LaserVIEW et de MeltVIEW réalise une fabrication, les données de surveillance de procédé sont transférées à un PCCD. Le service DataHUB exécuté sur ce PCCD transmet les données à tous les plug-ins installés. DataHUB exécute une instance de chaque plug-in installé, par fabrication. Chaque instance se voit attribuer un dossier de sortie portant un nom unique pour toute sortie maintenue, et chaque instance est exécutée indépendamment, ce qui garantit qu'une défaillance d'une instance n'engendrera pas par une défaillance générale du système.

La Figure 2 illustre le flux de données type lors du traitement d'une fabrication :

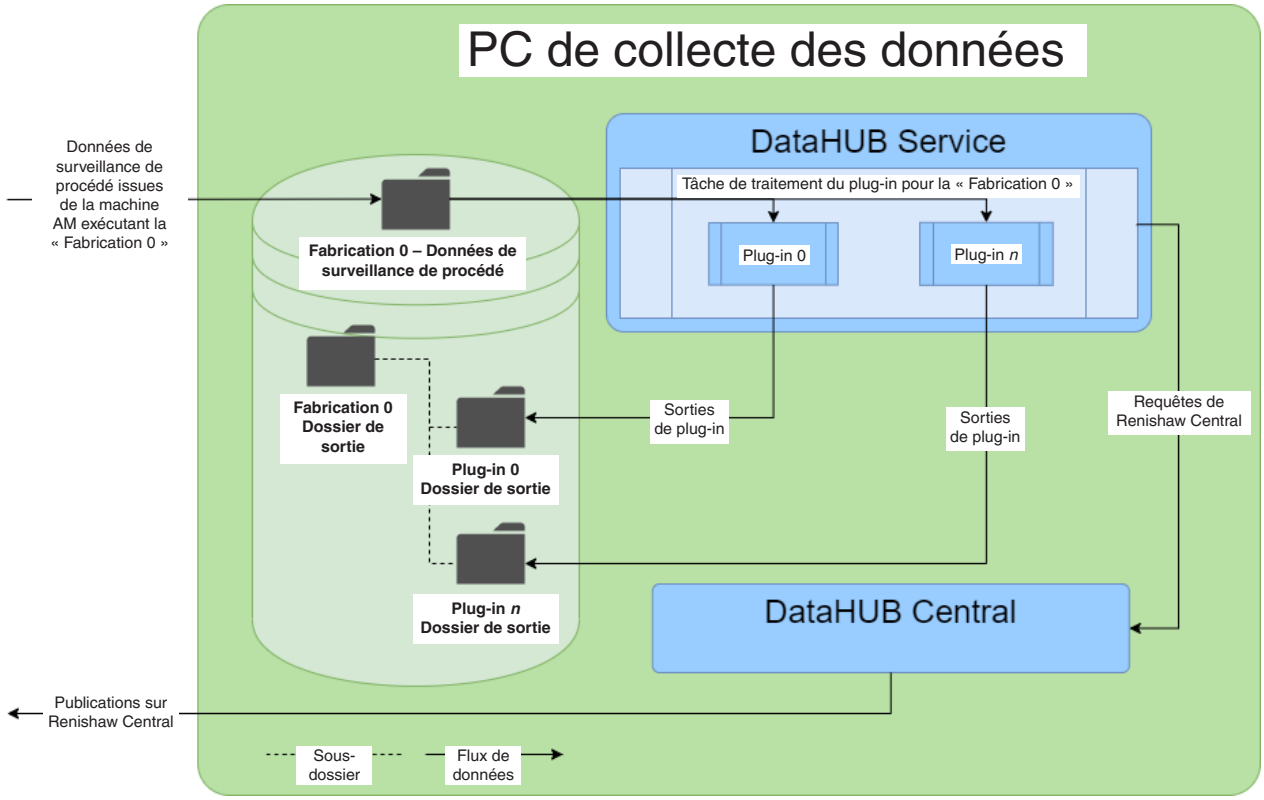


Figure 2 Flux de données de fabrication

Les données de surveillance de procédé recueillies par une machine AM exécutant une fabrication sont envoyées vers un dossier sur le PCCD. Le service DataHUB surveille ce dossier et, lorsque de nouvelles données arrivent, les transmet aux instances de plug-in associées à la tâche de fabrication. Les plug-ins peuvent notifier au service DataHUB les résultats (données de séries temporelles, alertes et résultats d'analyse) à publier sur Renishaw Central. Le service DataHUB Central gère ces demandes de téléchargement.

6.2.2 Technologie d'analyse des plug-ins

DataHUB est largement indépendant de la technologie utilisée pour mettre en œuvre les plug-ins et de toute infrastructure de support nécessaire. Ses deux exigences sont les suivantes :

1. Un plug-in est un assembly .NET mettant en œuvre l'ensemble de méthodes défini comme étant l'API de plug-in.
2. Toutes les dépendances d'exécution sont installées sur le PCCD.

DataHUB appelle les méthodes d'API de plug-in avec des valeurs renseignées avec les données de surveillance de procédé, et la mise en œuvre du plug-in est alors libre de transmettre les données à la technologie qui convient le mieux à la tâche d'analyse en cours. Une explication détaillée de l'intégration de .NET avec d'autres technologies et langages de programmation dépasse le cadre de ce document, mais vous trouverez de nombreuses informations relevant du domaine public à ce sujet.

6.2.3 Le flux de données « Push »

Le service DataHUB « pousse » les données vers les plug-ins : les API de plug-ins ne sont pas conçues pour interroger les ensembles de données et en extraire les valeurs. Le cas d'utilisation type où l'analyse est effectuée en temps réel au fur et à mesure de l'avancement de la fabrication est mieux pris en charge par ce flux de données. Pour les données archivés, vous pouvez utiliser DataHUB Generator pour demander au service DataHUB de retraiter les données afin que des analyses supplémentaires puissent être effectuées par les plug-ins.

6.2.4 Cycle de vie d'un plug-in

Pour qu'un plug-in soit utilisé par DataHUB, il est d'abord validé par rapport aux exigences de mise en œuvre de chaque API, et ce n'est que s'il est jugé conforme que des instances seront créées et utilisées pour analyser les données de fabrication. Une fois validé, une instance du plug-in est créée pour chaque nouvelle tâche de traitement des données de fabrication. Les étapes du cycle de vie d'un plug-in sont illustrées dans la Figure 3 :

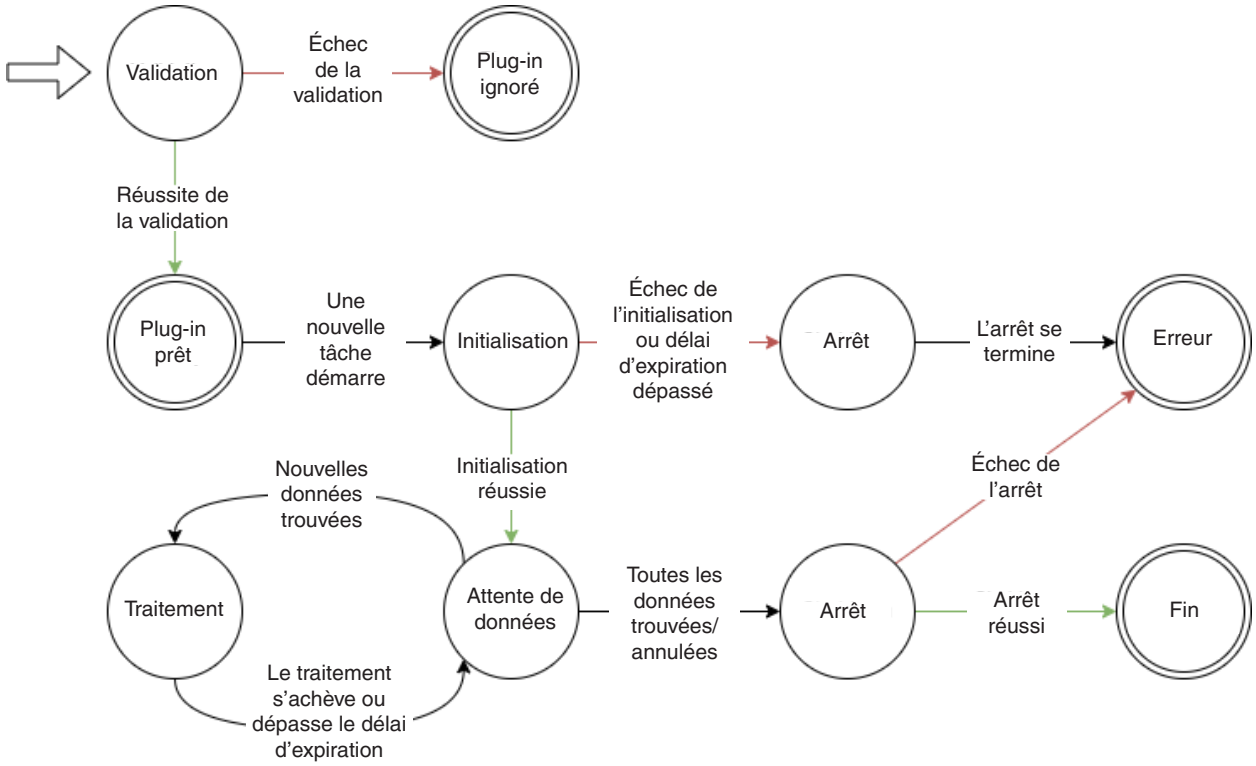


Figure 3 Différents états par lesquels un plug-in peut passer au cours de son cycle de vie

6.2.4.1 Validation

Lorsque DataHUB démarre, il recherche les ensembles .NET qui mettent en œuvre une ou plusieurs classes publiques conformes aux API de plug-in DataHUB. Tous ceux qui sont conformes sont enregistrés en tant que plug-ins valides et seront instanciés chaque fois qu'une tâche « Traiter les données LaserVIEW et MeltVIEW via des plug-ins » est lancée.

Un ensemble peut contenir des mises en œuvre de plusieurs plug-ins, si cela est souhaité, y compris en utilisant différentes versions de l'API.

6.2.4.2 Initialisation

Lorsque DataHUB démarre une tâche de plug-in, une nouvelle instance de plug-in est créée et sa méthode *Initialise* (Initialiser) est appelée. Les valeurs transmises à cette méthode sont des « constantes de fabrication », par exemple le nombre de lasers à partir duquel le plug-in peut calculer les besoins en ressources, etc. La méthode renvoie une indication de réussite ou d'échec de l'initialisation et, en cas d'échec, la méthode *Shutdown* (Arrêter) de l'instance est appelée immédiatement. Le plug-in peut éventuellement renvoyer des définitions pour les axes de séries temporelles qui seront remplis par les points de données générés au cours d'un traitement ultérieur (voir la section « Publication des données dans Renishaw Central » à la page 6-6).

6.2.4.3 Attente de données/Traitement

La méthode de traitement des données est appelée autant de fois qu'il est nécessaire pour transmettre toutes les données de surveillance de procédé à l'instance de plug-in. À chaque appel, le plug-in indique s'il a traité les données avec succès ou s'il a échoué. En cas d'échec, le plug-in ne passe pas en état d'erreur, il continue à recevoir les données (l'hypothèse étant que les données des autres couches restent intéressantes pour le plug-in). Le plug-in peut éventuellement renvoyer des points de données de séries temporelles, des alertes et d'autres résultats d'analyse à transmettre à Renishaw Central (voir la section « Publication des données dans Renishaw Central »).

6.2.4.4 Arrêt

La méthode *Shutdown* est appelée lorsqu'il ne reste plus de données à traiter (c'est-à-dire lorsque la fabrication a été terminée ou qu'elle a été annulée). Dans cette méthode, le plug-in doit effectuer le traitement final des données de fabrication, libérer les ressources acquises, etc.

6.2.5 Temps

Le temps est communiqué dans l'API en microsecondes par rapport au démarrage d'une couche. Un temps de 0 coïncide avec le début d'une couche, un temps de 1 000 correspond à 1 000 microsecondes après le démarrage de la couche, et ainsi de suite. Si nécessaire (par exemple, lors de l'envoi à Renishaw Central), DataHUB convertit tous les temps relatifs renvoyés par un plug-in en un format de temps absolu tel que l'UTC.

6.2.6 Publication des données dans Renishaw Central

Si DataHUB a été connecté à Renishaw Central, un plug-in peut publier les résultats d'analyse via DataHUB. Si la chaîne renvoyée par les méthodes *Initialise* (Itnitialiser) et *Process* (Traiter) est conforme au schéma JSON enregistré dans les définitions de l'API (voir la section 6.4, « API de plug-in V2 »), DataHUB publiera les données contenues sur le point de terminaison Renishaw Central.

DataHUB reconnait trois types de téléchargement vers Renishaw Central :

- Alertes
- Résultats d'analyse
- Séries temporelles

6.2.6.1 Alertes

Une alerte indique qu'un événement important s'est produit à un moment donné. Une alerte peut être générée par un plug-in lorsqu'il détecte une anomalie dans les données de fabrication, par exemple lorsque les données impliquent qu'une erreur s'est produite ou va se produire de manière imminente dans la fabrication. Il existe trois niveaux d'alerte : *Info*, *Warning* (Avertissement) et *Error* (Erreur). Ces niveaux d'alerte permettent d'informer l'application de visualisation (telle que le site Web de Renishaw Central) à propos du niveau de priorité qu'il convient d'accorder à l'alerte.

Les alertes ne peuvent être déclenchées qu'au cours de l'étape de cycle de vie *Process*.

6.2.6.2 Résultats d'analyse

Les résultats d'analyse, comme les alertes, peuvent être renvoyés en fonction de l'analyse effectuée sur les données de fabrication. Ils sont définis de manière flexible, ce qui permet de préciser leur contenu en fonction du type de résultat. Cela signifie que si (par exemple) le site Web de Renishaw Central ne sait pas comment représenter toutes les facettes d'un résultat d'analyse, il est possible d'écrire une application personnalisée en aval qui le fait.

Les résultats d'analyse ne peuvent être communiqués qu'au cours de l'étape de cycle de vie *Process*.

6.2.6.3 Séries temporelles

Une série temporelle représente une série de valeurs dans le temps. Une série temporelle peut contenir n'importe quel type de données, à condition qu'elles puissent être représentées graphiquement dans le temps. Au cours de l'étape *Process*, un plug-in peut ajouter deux types de données à une série temporelle : des valeurs et des limites. Les valeurs sont des paires temps/valeur qui ajoutent un point de données à la série temporelle. Les limites définissent les plages attendues pour faciliter l'affichage et indiquer les valeurs extrêmes dans les données.

Les séries temporelles diffèrent des alertes et des résultats d'analyse en ce sens qu'elles doivent d'abord être définies avant que des points de données puissent être ajoutés. Un plug-in peut renvoyer des définitions de séries temporelles à partir de sa méthode *Initialise*. Si le plug-in tente ultérieurement d'ajouter des données à une série temporelle qui ne figurait pas dans cette liste de définitions initiales (ou si les définitions elles-mêmes n'étaient pas valides), les données seront rejetées et perdues.

6.2.7 Journaux d'audit

DataHUB conserve un ensemble de journaux qui enregistrent les actions effectuées. Certains d'entre eux peuvent être utilisés pour diagnostiquer des problèmes avec un plug-in. Les journaux se trouvent à l'emplacement suivant :

<$PROGRAMDATA$\Renishaw\DataHUB\Audit Logs>

Les journaux sont nommés en fonction du jour où ils ont été enregistrés au format « aaaaMMjj ».

Les principales entrées qui intéresseront un développeur de plug-in sont les suivantes :

- Au démarrage, DataHUB enregistre les éléments suivants :
  - les numéros de version logicielle
  - les détails des licences
  - la validation des ensembles de plug-ins
- Au démarrage d'une fabrication, DataHUB enregistre les éléments suivants :
  - la création de l'instance de plug-in
  - les valeurs utilisées pour initialiser chaque instance de plug-in
  - le résultat de la méthode *Initialise*
- Au cours d'une fabrication, DataHUB enregistre les éléments suivants :
  - les résultats significatifs des appels *Processing* de plug-in (pour éviter d'encombrer les journaux avec des informations inutiles, un plug-in peut renvoyer un résultat trivial qui n'est pas enregistré).
- À la fin d'une fabrication, DataHUB enregistre l'élément suivant :
  - le résultat de chaque appel de méthode *Shutdown* de plug-in

6.2.8 Pré-traitement des données de surveillance de procédé

Les modules LaserVIEW et MeltVIEW recueillent des données sous la forme d'un flux d'échantillons, chacun associé à un laser spécifique, à une position sur le lit de poudre et à diverses mesures de capteurs, à un moment donné. DataHUB effectue un pré-traitement de ces données « brutes » de surveillance de procédé. Pour les plug-ins V1.0, DataHUB synthétise les échantillons en paquets, et pour la version V2.0, les données des échantillons sont rassemblées en un flux de jetons.

6.2.8.1 Échantillons en paquets

Le service DataHUB regroupe dans un paquet les échantillons consécutifs qui partagent la même position lorsque le laser est activé. L'heure de début d'un paquet est déterminée à partir de son premier échantillon, tandis que l'heure du dernier échantillon participant est utilisée pour calculer la durée du paquet. La moyenne des valeurs de mesure des capteurs LaserVIEW et MeltVIEW pour les échantillons en question est calculée.

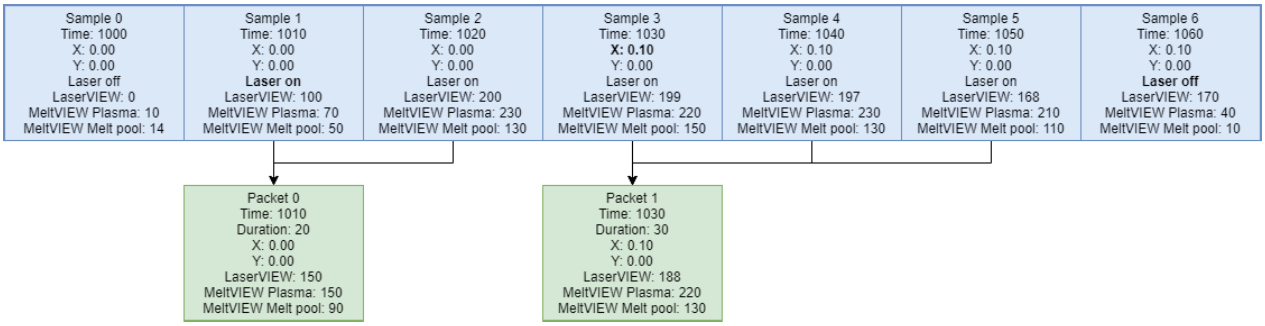


Figure 4 Échantillons en paquets

À la Figure 4, sept échantillons sont regroupés en deux paquets. Le premier échantillon est ignoré, car le laser ne se déclenche pas. Comme ils partagent la même position et que le laser se déclenche, les échantillons 1 et 2 sont regroupés en un paquet. La position de l'échantillon 3 change par rapport à son prédécesseur, un nouveau paquet commence donc, rassemblant les échantillons jusqu'à ce que le laser cesse de fonctionner. L'échantillon final 6 n'est pas ajouté car le laser est éteint, bien qu'il partage la même position que l'échantillon 5.

6.2.8.2 Échantillons en flux de jetons

Lors de la conversion des échantillons de surveillance de procédé en un flux de jetons, DataHUB crée un jeu de jetons *Sample* (Échantillon) entouré d'autres jetons d'information, lesquels décrivent la structure et le contenu de la fabrication. Contrairement à la version 1.0, les échantillons prélevés alors que le laser n'était pas activé sont pris en compte et ajoutés au flux de jetons.

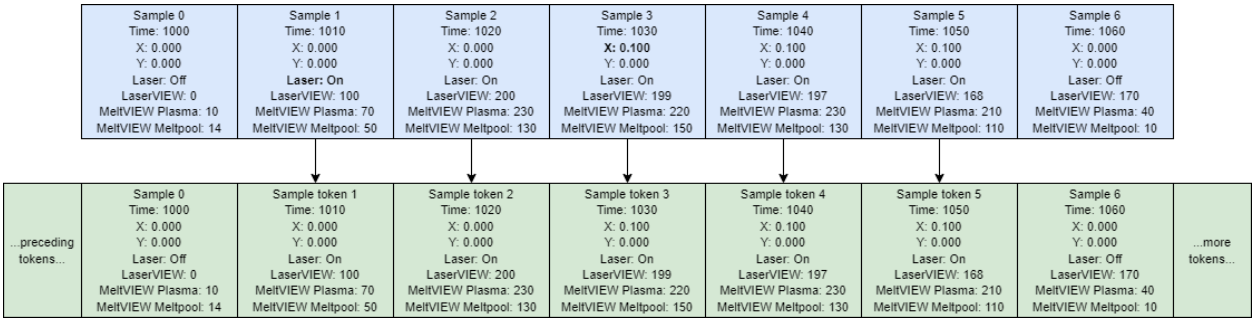


Figure 5 Échantillons en flux de jetons

La Figure 5 montre une partie d'un flux de jetons avec les mêmes échantillons que ceux utilisés dans la section *Échantillons en paquets*, ci-dessus.

6.3 API de plug-in V1

Ce document sert de guide pour créer, tester et déployer un composant logiciel de plug-in V1.0 pour DataHUB. Le cycle de développement du plug-in commence par l'installation de DataHUB sur le PC de développement, puis par la création (à l'aide de Visual Studio) d'un assembly .NET contenant le code du plug-in. Grâce à la configuration de débogage Visual Studio appropriée, le plug-in peut être testé et débogué pour vérifier son efficacité avant son déploiement final sur un PC de collecte de données (PCCD) sur un système AM de production.

6.3.1 Activités préalables

Les documents suivants doivent être lus parallèlement au présent document :

- Guide d'installation des logiciels InfiniAM® et DataHUB (Réf. Renishaw H-5800-6843)
- Manuel d'utilisation de DataHUB (Réf. Renishaw H-5800-6851)
- Manuel de l'utilisateur de Renishaw Licence Manager v2.x

Avant de poursuivre, les actions suivantes doivent être effectuées sur le PC qui sera utilisé pour le développement du plug-in :

- Assurez-vous que le PC est équipé d'un GPU correspondant aux exigences requises par DataHUB (voir la spécification matérielle du PC de collecte de données dans le guide d'installation des logiciels InfiniAM® et DataHUB).
- Vérifiez que les dernières versions des pilotes pour le GPU sont installées.
- Assurez-vous que le serveur de licences dispose de suffisamment de licences appropriées pour DataHUB et l'API Spectral.
- Vérifiez l'accès au réseau du serveur de licences.
- Assurez-vous qu'une version de Microsoft Visual Studio prenant en charge .NET Framework 4.7.2 est installée.

6.3.2 Installation de DataHUB pour les développement de plug-ins

Le service DataHUB doit être installé sur le PC de développement. Lors de l'installation, vous avez la possibilité de choisir les licences à récupérer. Assurez-vous que l'option **LaserVIEW/MeltVIEW Pluggability** (Ajout de plug-ins LaserVIEW/MeltVIEW) est cochée :

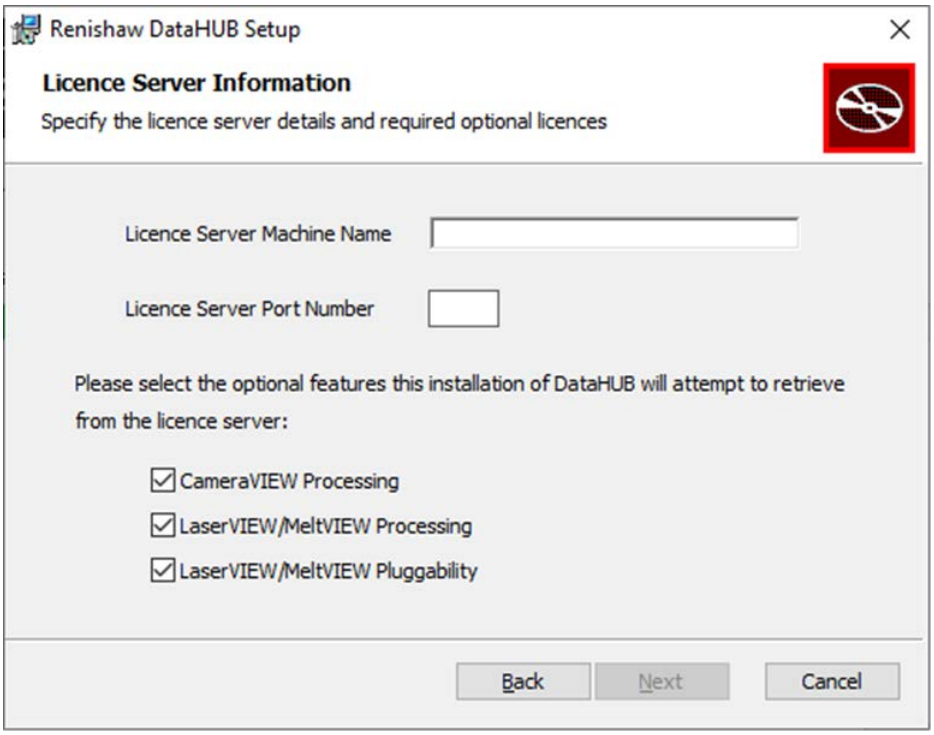


Figure 6 Octroi de licences DataHUB

6.3.3 Mise en œuvre du plug-in

Les sections suivantes de ce document présentent le flux de travail de bout en bout de la création, de la compilation, du débogage et du déploiement d'un plug-in à l'aide de Visual Studio.

Les plug-ins sont écrits en langage C# et définissent une classe publique mettant en œuvre le jeu spécifique de méthodes publiques que DataHUB utilise pour identifier, valider et utiliser les instances de plug-in lors du traitement des données de surveillance de procédé d'une fabrication.

REMARQUE : les captures d'écran présentées ci-dessous proviennent de Microsoft Visual Studio 2019, mais le flux de travail est globalement similaire dans les autres versions.

6.3.3.1 Création de la solution Visual Studio

Lancez Visual Studio et choisissez *Create a new project* (Créer un nouveau projet). Parmi les options disponibles, choisissez *Class Library (.NET Framework)* (Bibliothèques de classes (.NET Framework))

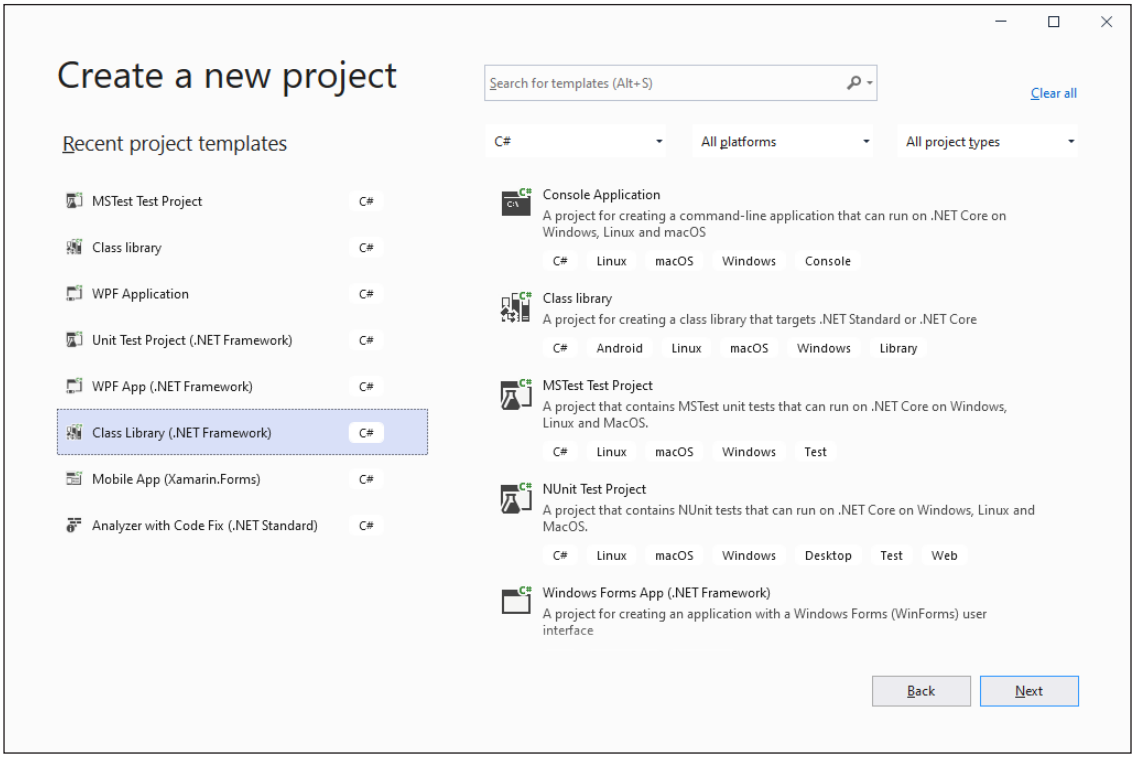


Figure 7 Création de la solution

Ensuite, configurez le projet :

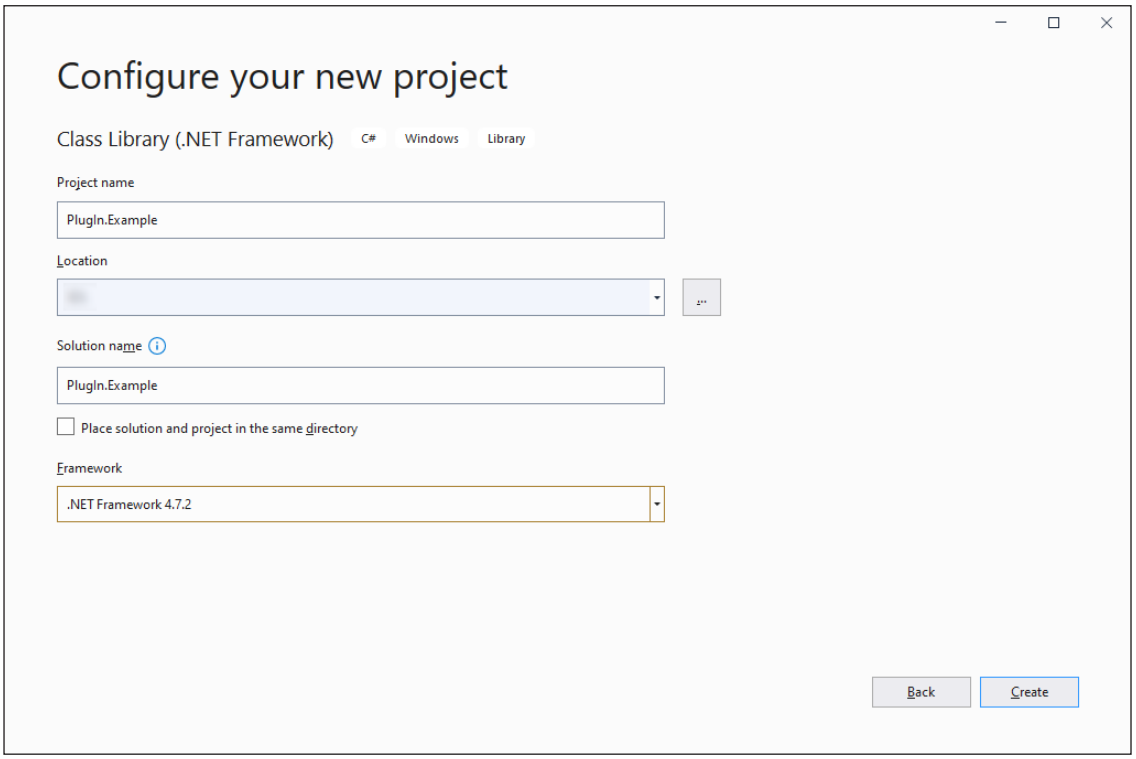


Figure 8 Configuration du projet

DataHUB exige qu'un assembly de plug-in commence par **PlugIn**. (Le mot « PlugIn » suivi d'un point). Sélectionnez un dossier de votre choix dans la zone *Location* (Emplacement) et créez la solution.

REMARQUE : DataHUB prend en charge les plug-ins développés pour les architectures x86/x64/tout processeur. Il est recommandé d'utiliser Framework 4.7.2 ou une version ultérieure.

6.3.3.2 Dénomination du type de plug-in

Le seul fichier source par défaut Class1.cs contient la définition de `Class1`, et nous l'utiliserons comme point de départ.

```
namespace PlugIn.Example
{
    public class Class1
    {
    }
}
```

Une fois que DataHUB a détecté un assembly de plug-in potentiel, il recherche un type public `PlugIn`, et il faut donc renommer `Class1` en `PlugIn`.

```
namespace PlugIn.Example
{
    public class PlugIn
    {
    }
}
```

6.3.3.3 Ajout de l'API

En présence d'un type de plug-in potentiel, DataHUB exige qu'il mette en œuvre les méthodes publiques suivantes :

`public PlugIn()` (generated automatically)
`public string APIVersion()`
`public string DisplayName()`
`public string Initialise(…)`
`public string ProcessPackets(…)`
`public string Shutdown()`

Ajoutez les mises en œuvre de méthodes suivantes à la classe :

```
public class PlugIn
{
   public string APIVersion() => "";
   public string DisplayName() => "";
   public string Initialise(
      string plugInFolderPath,
      string outputFolderPath,
      uint numberOfDatFilesPerLayer) => "";
```

```
    public string ProcessPackets(
        uint layerNumber,
        uint laserId,
        int count,
        uint[] startTime,
        uint[] duration,
        double[] demandPositionX,
        double[] demandPositionY,
        uint[] laserVIEW,
        uint[] meltVIEWPlasma,
        uint[] meltVIEWMeltpool) => "";
    public string Shutdown() => "";
}
```

Notez que toutes les méthodes de l'API renvoient une `chaîne` (qui, pour l'instant, a été laissée vide). Le contenu de la chaîne renvoyée est utilisé pour signaler à DataHUB le résultat de la méthode. Le rôle de cette valeur de retour sera expliqué plus loin, au fur et à mesure que chaque méthode sera détaillée.

6.3.3.4 La méthode APIVersion

Cette méthode indique à DataHUB la version de l'API utilisée par le plug-in. Pour utiliser l'API version 1.0, cette méthode doit renvoyer exactement « 1.0 » :

```
public string APIVersion() => "1.0";
```

6.3.3.5 La méthode DisplayName

Le contenu de la chaîne renvoyée par cette méthode est utilisé comme nom de plug-in affiché dans DataHUB Monitor et pour la création du dossier de sortie du plug-in. Bien qu'il ne soit pas obligatoire que ce nom soit unique, cela permet d'identifier le plug-in lorsque, par exemple, plusieurs versions sont installées sur un PCCD.

```
public string DisplayName() => "Example plug-in";
```

6.3.3.6 La méthode Initialise

Cette méthode est appelée par DataHUB au début d'une fabrication, avant que les données de la couche ne soient envoyées au plug-in. Elle transmet donc les données des *invariants de la fabrication* au plug-in. Dans le plug-in en exemple, la méthode *Initialise* attribue le chemin d'accès au dossier de sortie de l'instance au champ **_outputFolderPath** :

```
public string Initialise(
   string plugInFolderPath,
   string outputFolderPath,
   uint numberOfDatFilesPerLayer)
{
   _outputFolderPath = outputFolderPath;
   return "Success";
}
…
private string _outputFolderPath;
```

6.3.3.7 La méthode ProcessPackets

Cette méthode est appelée par DataHUB lorsque toutes les données d'une couche ont été reçues. Dans le plug-in en exemple, la méthode *ProcessPackets* construit d'abord une ligne d'en-têtes de colonnes séparées par un délimiteur, puis ajoute ligne par ligne les valeurs des données des paquets de surveillance de procédé :

```
public string ProcessPackets(
   uint layerNumber,
   uint laserId,
   int count,
   uint[] startTime,
   uint[] duration,
   double[] demandPositionX,
   double[] demandPositionY,
   uint[] laserVIEW,
   uint[] meltVIEWPlasma,
   uint[] meltVIEWMeltpool)
{
   var builder = new StringBuilder().
      Append($"Start time / us\t").
      Append($"Duration / us\t").
      Append($"x position / mm\t").
      Append($"y position / mm\t").
      Append($"LaserVIEW\t").
```

```
        Append($"MeltVIEW Plasma\t").
        Append($"MeltVIEW Melt Pool").
        Append(Environment.NewLine);
    for (var i = 0; i < count; ++i)
        builder.
            Append($"{startTime[i]}").
            Append($"\t").
            Append($"{duration[i]}").
            Append($" \t").
            Append($"{demandPositionX[i].ToString("F3", CultureInfo.CurrentCulture)}\t").
            Append($"{demandPositionY[i].ToString("F3", CultureInfo.CurrentCulture)}\t").
            Append($"{laserVIEW[i]}\t").
            Append($"{meltVIEWPlasma[i]}\t").
            Append($"{meltVIEWMeltpool[i]}").
            Append(Environment.NewLine);
    System.IO.File.WriteAllText(
        Path.Combine(_outputFolderPath,
                    $"Packet data for layer {layerNumber}, laser {laserId}.txt"),
        builder.ToString());
    return "Success";
}
```

6.3.3.8 La méthode Shutdown

Cette méthode est appelée par DataHUB lorsque toutes les données d'une fabrication ont été envoyées au plug-in, ou lorsque tout appel précédent au plug-in a échoué. Cette méthode ne dispose d'aucun paramètre.

Dans cette méthode, le plug-in doit effectuer le traitement final des données de fabrication, libérer les ressources acquises, etc. Dans ce plug-in en exemple, la méthode *Shutdown* est :

```
public string Shutdown() => "Success";
```

6.3.4 Déploiement du plug-in pour le débogage

Construisez la solution en mode *Debug*, ouvrez un navigateur de fichiers et localisez le dossier de sortie qui contiendra un fichier *.dll* nommé conformément au nom donné au projet, par exemple, *PlugIn.Example.dll*. (Des fichiers auxiliaires tels que des symboles de débogage, etc. peuvent également être présents.)

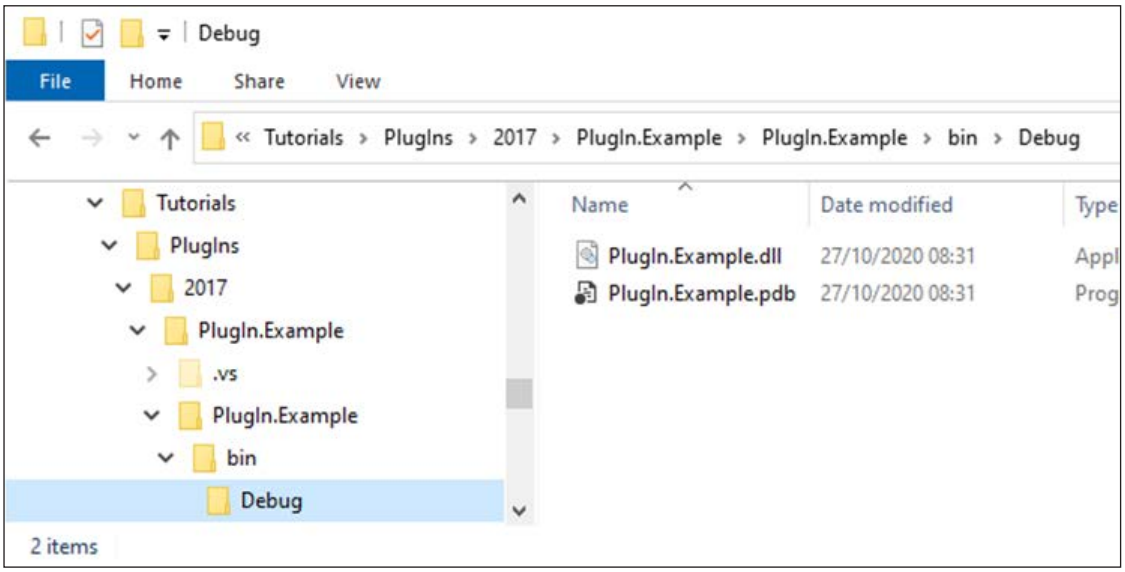


Figure 9 Fichiers binaires d'un plug-in

Copiez ce dossier et collez-le dans le sous-dossier *PlugIns* dans le dossier de données DataHUB :

<$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>.

REMARQUE : en général, $PROGRAMDATA$ se trouve sous <C:\ProgramData>, mais il peut être masqué. Dans ce cas, il faudra cocher l'option en suivant le chemin **View, Show/hide, Hidden items** (Afficher, Afficher/masquer, Éléments masqués).

Des privilèges d'administrateur peuvent être requis pour modifier le contenu de ce dossier.

Renommez le dossier *Debug* afin qu'il corresponde à celui du plug-in (Dans l'image ci-dessous, le dossier à été nommé *PlugIn.Example*).

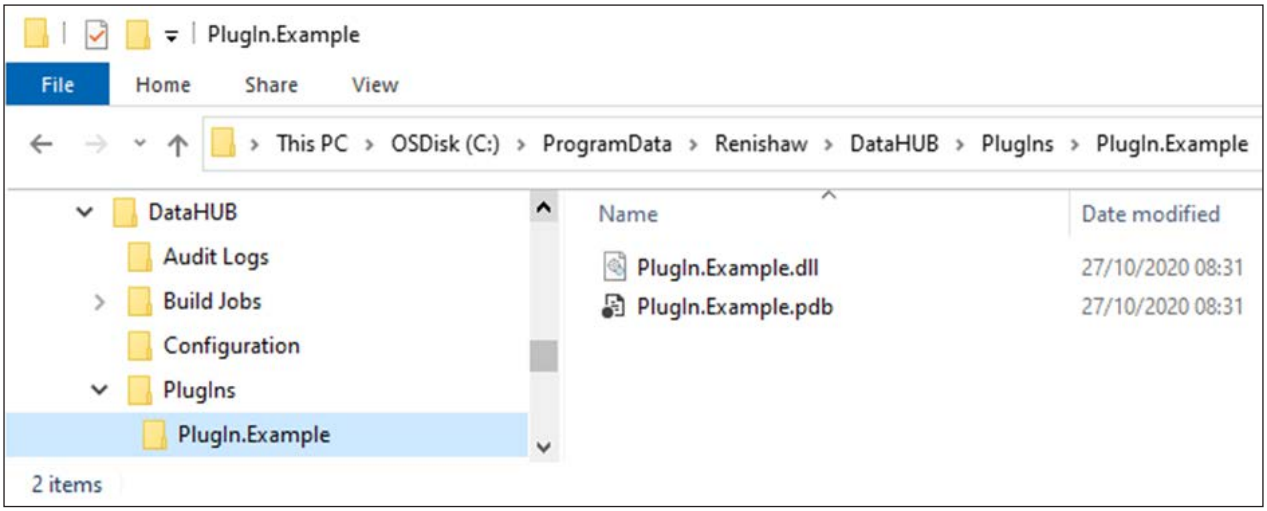


Figure 10 Plug-in déployé

6.3.4.1 Enregistrement du plug-in

DataHUB Service doit être redémarré pour enregistrer le nouveau plug-in. Pour ce faire, utilisez l'application *Services* en cliquant avec le bouton droit de la souris sur *DataHUB Service* et cliquez sur *Restart* (Redémarrer) :

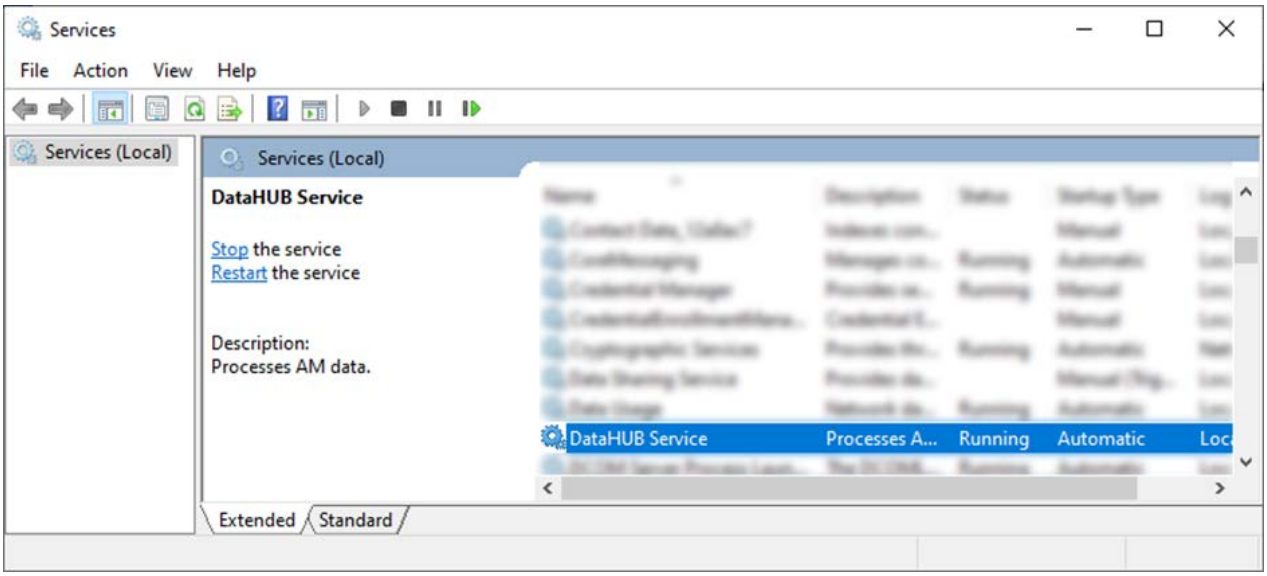


Figure 11 Application Services de Windows

Après quelques secondes, le service DataHUB s'affichera à l'état *Running* (En cours d'exécution).

6.3.4.2 Vérification de l'enregistrement

DataHUB génère un fichier journal au fur et à mesure de son exécution, détaillant les événements importants tels que les démarrages de fabrication, les erreurs de fabrication et, bien sûr, l'audit des plug-ins. Dans l'Explorateur de fichiers, localisez le dossier des journaux <$PROGRAMDATA$\Renishaw\DataHUB\Audit Logs>.

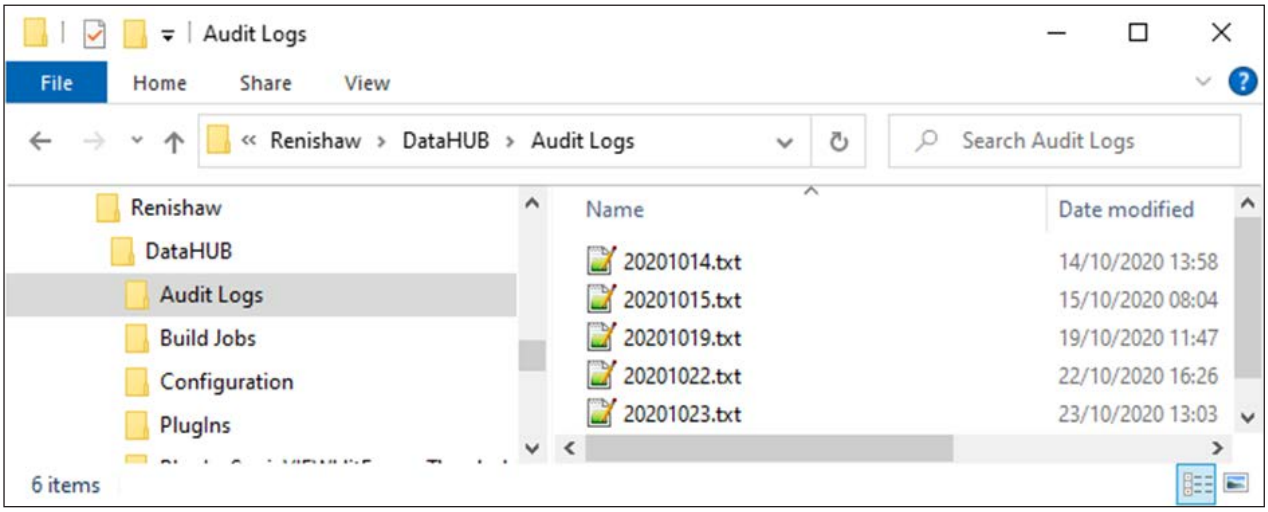


Figure 12 Dossier des journaux de DataHUB

Ouvrez le fichier journal le plus récent (il doit porter la date d'aujourd'hui) faites défiler jusqu'à la fin et recherchez les lignes détaillant la découverte de plug-ins valides :

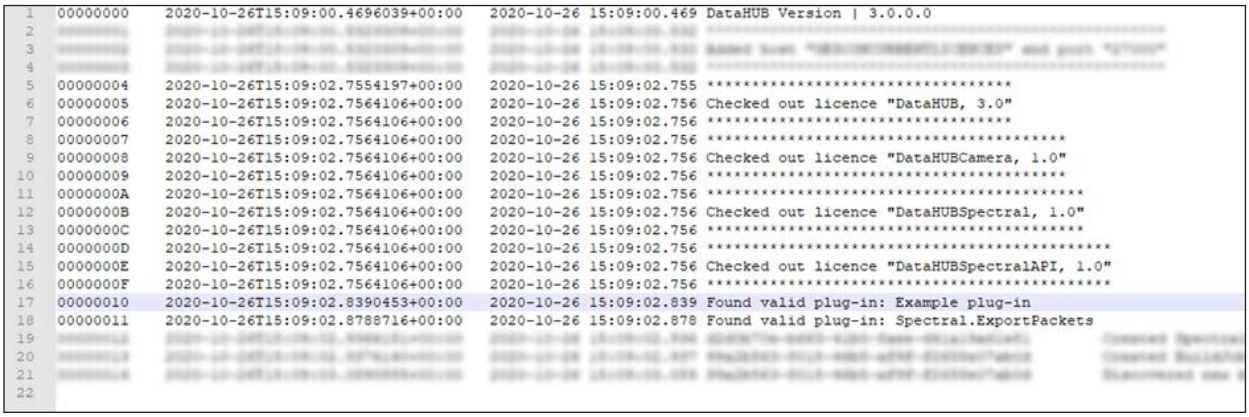


Figure 13 Plug-ins valides trouvés par DataHUB

6.3.5 Délai d’expiration

Comme DataHUB ne peut pas garantir que chaque appel à une instance de plug-in fera l’objet d’un renvoi en temps voulu, il adopte une politique de délai d’expiration. Si un appel au plug-in n’aboutit pas dans un certain délai, il est considéré comme défectueux et il est stoppé. Les délais d’expiration par défaut sont :

- Pour *Initialise*, 20 secondes
- Pour *ProcessPackets*, 200 secondes
- Pour *Shutdwon*, 20 secondes

Pendant le débogage, il arrive souvent que ces délais soient dépassés, ce qui entraîne l’arrêt prématuré du plug-in par DataHUB. Pour permettre plus de temps de débogage, les délais d’expiration peuvent être augmentés en modifiant le fichier <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns\PlugInsConfiguration.xml> dans un éditeur XML ou un éditeur de texte approprié.

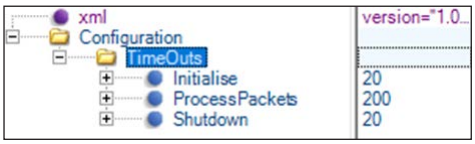


Figure 14 Délais d’expiration par défaut des plug-ins

REMARQUE : DataHUB Service doit être redémarré pour que ces changements de délai d’expiration prennent effet.

6.3.6 Débogage du plug-in

Lorsque le plug-in a été déployé avec succès, il est maintenant possible de déboguer son exécution. Le plug-in lui-même n'est pas directement exécutable, mais en associant le débogueur Visual Studio à DataHUB Service et en lançant une tâche de fabrication, la mise en œuvre du plug-in sera invoquée par le service, et le jeu de points d'arrêt défini dans le code sera atteint.

REMARQUE : pour effectuer le débogage via le service, Visual Studio peut exiger des privilèges d'administrateur. Ces droits peuvent être accordés en cliquant avec le bouton droit de la souris sur le raccourci du programme Visual Studio et en sélectionnant **Run as administrator** (Exécuter en tant qu'administrateur).

Dans Visual Studio, placez des points d'arrêt au début des méthodes *Initialise* et *ProcessPackets* :

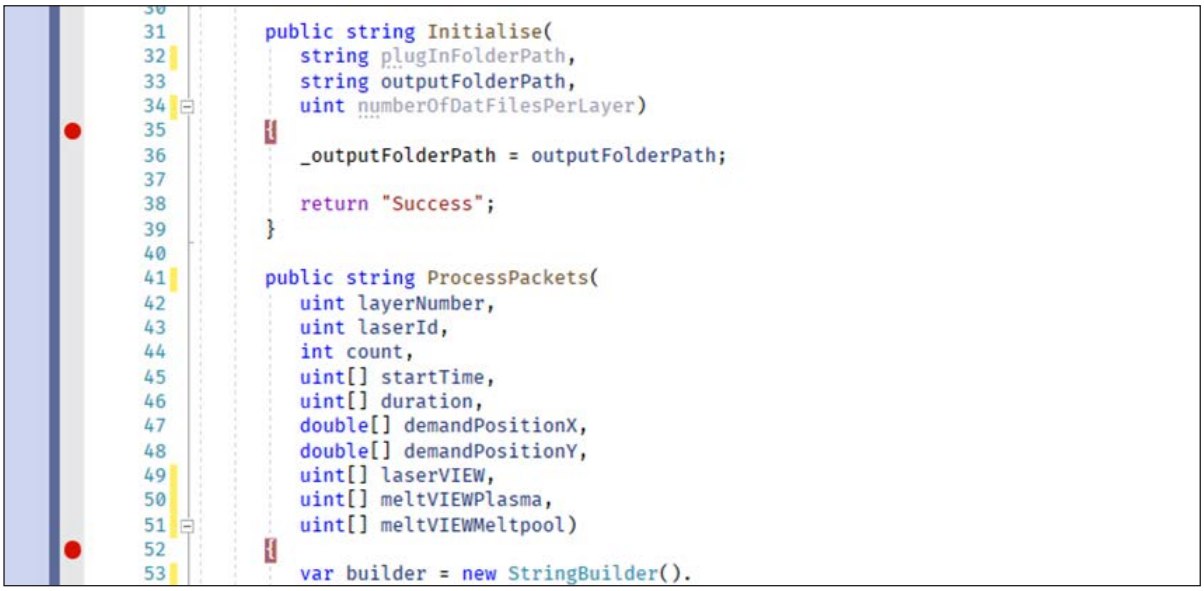


Figure 15 Points d'arrêt définis dans *Initialise* et *ProcessPackets*

Ouvrez le menu **Debug** (Déboguer) et sélectionnez **Attach to Process...** (Attacher au processus) :

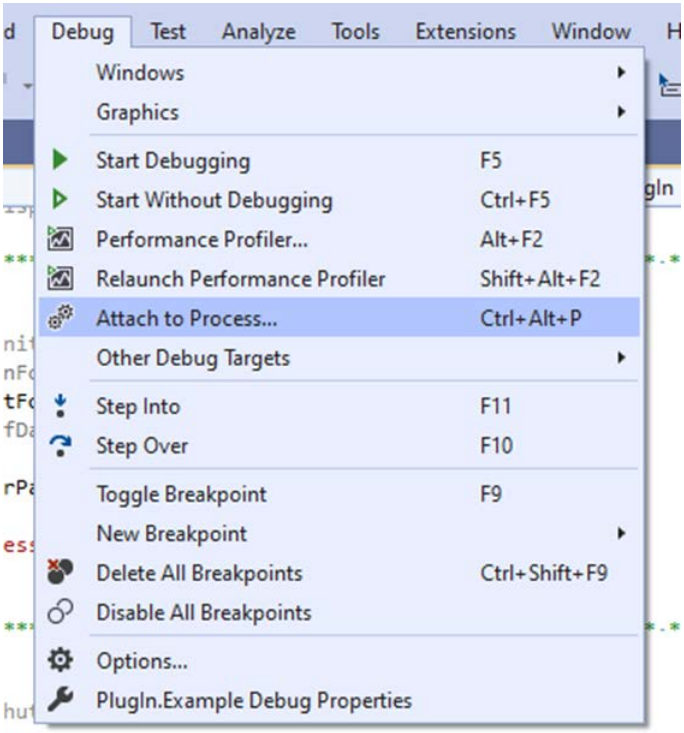


Figure 16 Attachement à un processus pour le débogage

Dans la liste des processus en cours d'exécution, localisez *DataHUB Service.exe*. Vous devrez peut-être cocher la case **Show processes from all users** (Afficher les processus de tous les utilisateurs) :

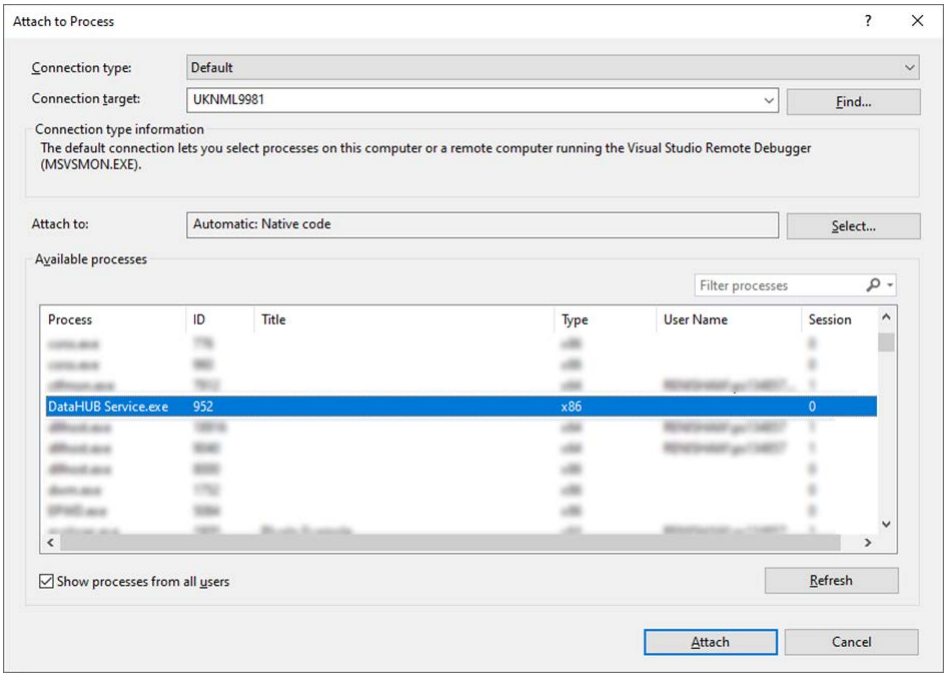


Figure 17 Processus DataHUB Service

Appuyez sur le bouton **Attach** (Attacher) et Visual Studio va démarrer une session de débogage pour le service DataHUB.

Pour que DataHUB appelle le plug-in et donc pour que les points d'arrêt soient atteints, il faut lancer une fabrication. Vous pouvez le faire via DataHUB Generator ou, si vous avez déjà généré des fabrications, simplement en dupliquant un dossier de fabrication dans <$PROGRAMDATA$\Renishaw\DataHUB\Build Jobs>.

DataHUB détectera automatiquement la nouvelle fabrication et commencera à traiter les données. L'une des tâches préliminaires du service consiste à créer (en invoquant le constructeur sans paramètre) une instance de chaque plug-in enregistré. Lorsque DataHUB appelle ensuite la méthode *Initialise* de l'instance, le premier point d'arrêt est atteint. Les valeurs des paramètres de la méthode seront celles de la fabrication en question, par exemple :

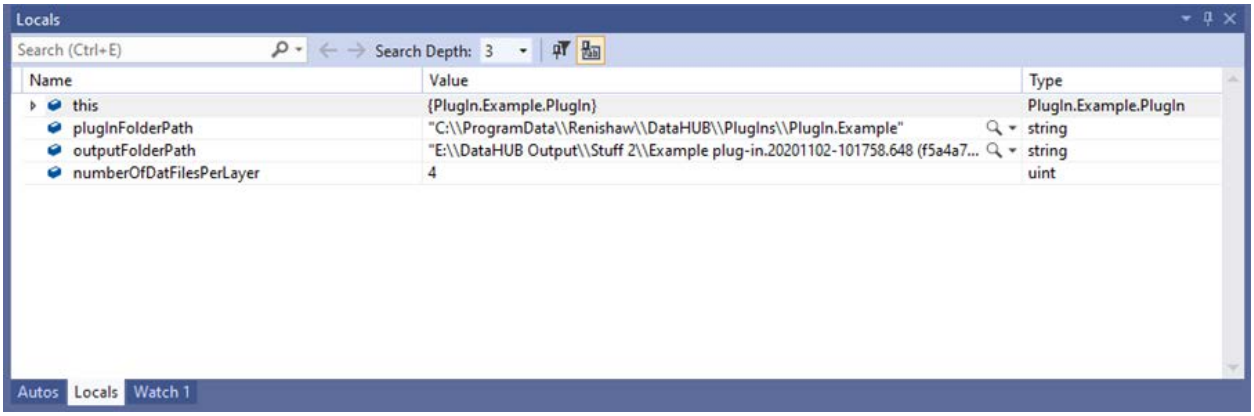


Figure 18 Valeurs des paramètres de la méthode *Initialise*

À partir de ce moment, les données existantes (et, pour une fabrication active, les nouvelles données qui arrivent) sont traitées et la méthode *ProcessPackets* de l'instance est appelée avec des valeurs de paramètres pour les données LaserVIEW et MeltVIEW associées à une couche :

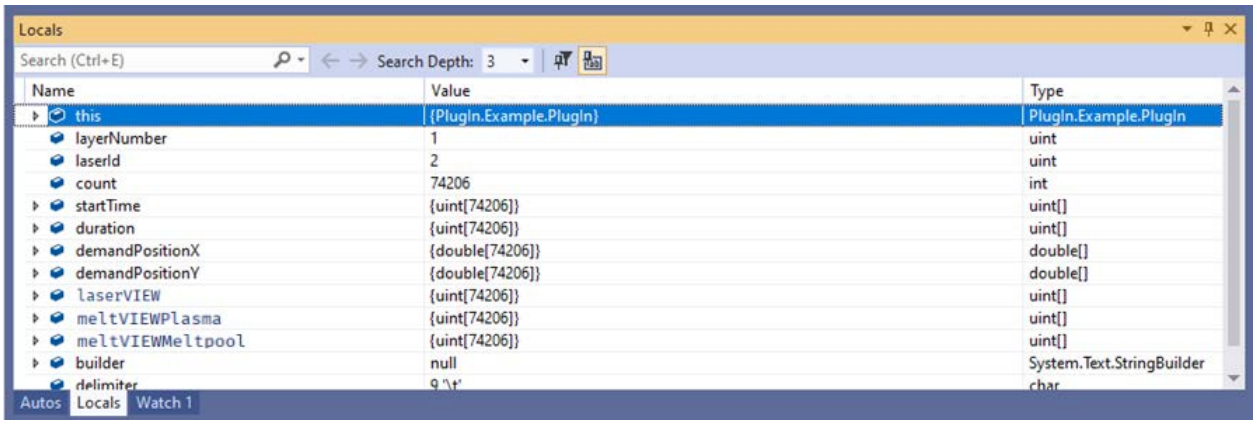


Figure 19 Valeurs des paramètres de la méthode *ProcessPackets*

Le test de l'ajout et de la vérification des points d'arrêt dans la méthode *Shutdown* est laissé à l'appréciation du lecteur.

6.3.7 Déploiement du plug-in pour la production

Pour préparer le déploiement du plug-in sur un PCCD de production, il est essentiel qu'une version finale (*Release*) soit compilée, déployée localement et testée. Une fois la fonctionnalité du plug-in validée et vérifiée, il peut être déployé sur les PCCD de production.

Le processus de déploiement sur un PCCD de production est très similaire à celui du déploiement sur un PC de développement. Le dossier *Release* doit être copié dans le sous-dossier *PlugIns* du dossier de données de DataHUB (<$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>) sur le PCCD et renommé pour correspondre au nom du plug-in. Le service DataHUB doit ensuite être redémarré pour enregistrer le nouveau plug-in : pour ce faire, utilisez l'application *Services*. Le succès ou l'échec de l'enregistrement du nouveau plug-in est consigné dans le journal d'audit de la journée.

REMARQUE : DataHUB est conçu pour reprendre toutes les tâches de traitement de données en cours après un redémarrage, il n'est donc pas nécessaire d'attendre que toutes les tâches en cours soient terminées avant de redémarrer. Cependant, seules les nouvelles tâches de traitement des données utiliseront le plug-in récemment enregistré.

6.3.8 Diagnostiquer les échecs du plug-in en production

Dans l'environnement de production, DataHUB Monitor affichera (avec d'autres tâches de traitement de la fabrication) l'état de traitement propre au plug-in :

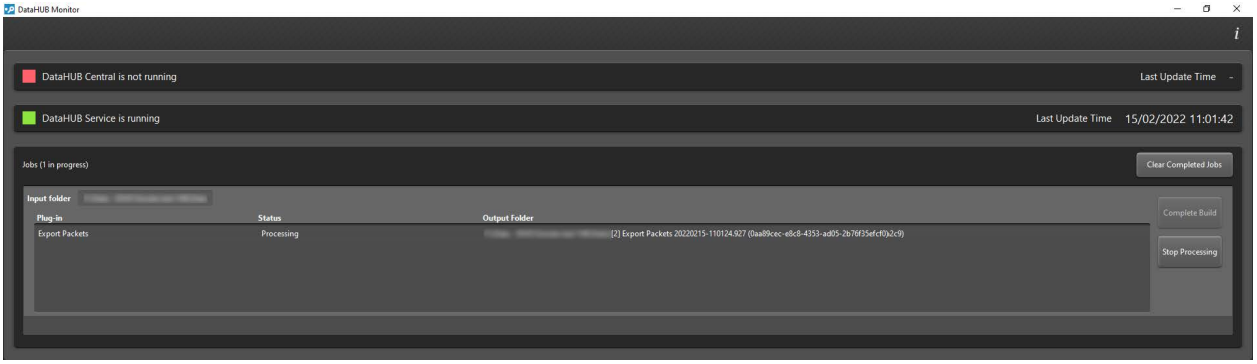


Figure 20 Fabrication en cours d'exécution avec des plug-ins

Si un plug-in est à l'état *Error* (Erreur), le journal d'audit contiendra des enregistrements relatifs à l'échec, soit sous forme d'entrées ajoutées par DataHUB lui-même (p. ex., l'échec du chargement de l'assembly de plug-in en raison de dépendances manquantes), soit par le plug-in via le contenu des valeurs de retour renvoyées par les méthodes *Initialise*, *ProcessPackets* ou *Shutdown* du plug-in.

6.3.9 Résolution des problèmes

Le tableau suivant présente les échecs les plus courants, leur entrée type dans le journal d'audit et les solutions possibles :

Entrée du journal d'audit	Échec	Solution potentielle
L'ensemble ne contient pas de classe nommée PlugIn	Classe PlugIn non trouvée	Renommez la classe du plug-in, recompilez et redéployez.
La classe PlugIn ne contient pas de méthode appellée xyz	La classe PlugIn ne met pas en œuvre une des méthodes de l'API.	Examinez la classe à la recherche des méthodes manquantes ou mal orthographiées.
La méthode xyz ne montre pas la signature attendue…	La classe PlugIn ne met pas correctement en œuvre une des méthodes de l'API.	Examinez la signature de la méthode et assurez-vous que tous les paramètres correspondent à la définition de l'API.
La classe PlugIN utilise une version pas supportée du plugin API « {version} ». Cette version de DataHUB supporte seulement la version « 1.0 » du plugin API.	La classe PlugIn ne renvoie pas une chaîne de version d'API valide.	Assurez-vous que la méthode APIVersion renvoie « 1.0 ».
Failed to create plug-in output folder <path> Echec de l'opération de creation du chemin vers le dossier de sortie du plug-in	Le dossier d'une instance spécifique d'un plug-in n'a pas été créé.	Assurez-vous des droits d'accès en lecture/écriture/modification/ création sont octroyés pour <path>.
Résultat de l'initialisation : timed out Résultat du process : timed out Résultat procédure d'arrêt : timed out	Un appel à l'une de ces méthodes de l'API a dépassé le délai d'expiration pour être renvoyé : le plug-in est supposé avoir échoué.	Envisagez de réduire la quantité de travail effectuée dans la méthode ou d'augmenter la durée du délai d'expiration dans le fichier PlugInsConfiguration. xml.
Exception :	Une erreur inattendue s'est produite.	Examinez les messages d'exception consignés pour déterminer la source de l'échec.

6.3.10 Intégration avec Renishaw Central

Les résultats d'analyse (points de données de séries temporelles, alertes, etc.) d'un plug-in peuvent être transmis à un serveur Renishaw Central. Le document *API de plug-in V2.0 et tutoriels* (Réf. Renishaw H-5800-6784) contient une explication complète sur la manière de structurer le contenu de la chaîne renvoyée par les méthodes *Initialise* et *ProcessPackets* du plug-in pour réaliser cette intégration, ainsi que des détails sur les fonctions d'aide qui compatibles avec l'API V1.0, dans le fichier *PlugIns.API.v2.dll*.

6.3.11 API de plug-in V1.0

6.3.11.1 Dénomination de l'ensemble

Le nom de fichier de l'assembly .NET doit commencer par « PlugIn. » (« PlugIn » suivi d'un point) et doit avoir l'extension « dll ».

6.3.11.2 Dénomination de la classe de plug-in

L'assembly de plug-in doit définir une classe publique nommée « `PlugIn` ».

6.3.11.3 Méthodes de la classe de plug-in

La classe de plug-in doit mettre en œuvre les méthodes publiques suivantes :

```
public PlugIn()
public string APIVersion()
public string DisplayName()
public string Initialise(…)
public string ProcessPackets(…)
public string Shutdown()
```

6.3.11.4 Constructeur sans paramètre

Le type doit avoir un constructeur sans paramètre. Si aucun constructeur avec paramètres n'est défini, C# le fournit automatiquement, c'est-à-dire qu'il n'est pas nécessaire de le définir explicitement.

REMARQUE : un plug-in ne doit pas acquérir de ressources au sein de son constructeur.

Lorsque vous construisez une instance de plug-in, il n'est pas garanti que la méthode *Shutdown* soit appelée, donc ces ressources pourraient ne pas être libérées en temps voulu *Initialise* est l'endroit approprié pour l'acquisition des ressources.

6.3.11.5 La méthode `APIVersion`

Elle identifie la version de l'API par rapport à laquelle ce plug-in doit être validé, ainsi que le format des données de surveillance de procédé qu'il traitera.

Paramètres :

Aucun.

Retour : `string`

Un plug-in qui met en œuvre l'API V1.0 doit renvoyer exactement « 1.0 ».

6.3.11.6 La méthode `DisplayName`

Le contenu de la chaîne renvoyée par cette méthode est utilisé comme nom de plug-in affiché dans DataHUB Monitor, lors de la création du dossier de sortie du plug-in et lors de l'audit des événements pour le plug-in. Bien qu'il ne soit pas obligatoire que ce nom soit unique, cela permet d'identifier le plug-in lorsque, par exemple, plusieurs versions sont installées sur un PCCD.

Paramètres :

Aucun.

Retour : `string`

Texte spécifique à utiliser dans les différentes zones de l'interface utilisateur de DataHUB, pour se connecter, etc.

6.3.11.7 La méthode `Initialise`

Cette méthode est appelée par DataHUB au début d'une fabrication, avant que les données de la couche ne soient envoyées au plug-in, ce qui lui permet de recevoir les valeurs des paramètres *invariants de la fabrication*. Le plug-in peut utiliser cette méthode pour allouer les ressources nécessaires pendant la durée de vie de l'instance, calculer les constantes de fabrication, etc.

Paramètre 1 : `string`

Chemin d'accès du dossier de l'assembly de plug-in, par exemple, <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns\PlugIn.Example>

Paramètre 2 : `string`

Chemin d'accès du dossier dans lequel ce plug-in peut écrire des fichiers de sortie. Notez que pour chaque plug-in, pour chaque fabrication, ce dossier sera unique et nommé en utilisant le nom d'affichage du plug-in (tel que défini par la valeur renvoyée par la méthode `DisplayName`), un horodatage et un GUID. Chaque fois que le plug-in est exécuté pour une fabrication, le nom du dossier change, mais il utilise toujours cette structure :

<Dossier de sortie de fabrication>\[1.0] <Nom d'affichage du plug-in>.aaaaMMjj-HHmmss.fff (GUID)

Paramètre 3 : `uint`

Il s'agit d'une valeur de 1 à 4, définie par la configuration matérielle de la machine qui crée les données.

Retour : `string`

Le plug-in doit renvoyer une chaîne de caractères contenant soit :

- le mot « Success », si la méthode s'est déroulée correctement, ou
- le détail de la cause de l'échec, ou
- une chaîne JSON conforme à l'API de renvoi à Renishaw Central.

Si DataHUB reçoit une chaîne non conforme, le plug-in sera stoppé et le contenu de la chaîne renvoyée sera ajouté au journal d'audit pour un diagnostic ultérieur.

6.3.11.8 La méthode `ProcessPackets`

Cette méthode est appelée par DataHUB lorsque toutes les données d'une couche ont été reçues sur le PCCD. Les paramètres de cette méthode sont les suivants :

Paramètre 1 : `uint`

Numéro de la couche de fabrication concernée par les données. La première couche est la couche 1.

Paramètre 2 : `uint`

Numéro d'identification (« 1 », « 2 », « 3 » ou « 4 ») du laser pour lequel les données ont été collectées.

Paramètre 3 : `int`

Décompte du nombre d'éléments dans les matrices qui suivent.

Paramètre 4 : `uint[ ]`

Heure de début de chaque paquet, en µs, par rapport au démarrage de la couche.

Paramètre 5 : `uint[ ]`

Durée de chaque paquet, en µs.

Paramètre 6 : `double[ ]`

Position demandée en X pour chaque paquet sur le lit de poudre, en mm.

Paramètre 7 : `double[ ]`

Position demandée en Y pour chaque paquet sur le lit de poudre, en mm.

Paramètre 8 : `uint[ ]`

Valeur moyenne de tous les échantillons collectés par le capteur LaserVIEW sur la durée de chaque paquet.

Paramètre 9 : `uint[ ]`

Valeur moyenne de tous les échantillons collectés par le capteur de MeltVIEW Plasma sur la durée de chaque paquet.

Paramètre 10 : `uint[ ]`

Valeur moyenne de tous les échantillons collectés par le capteur de MeltVIEW Melt Pool sur la durée de chaque paquet.

Retour : `string`

Le plug-in doit renvoyer soit :

- une chaîne contenant le mot « Success », si la méthode s'est déroulée correctement, ou
- une chaîne contenant autant de détails que souhaités sur la cause de l'échec, ou
- une chaîne JSON conforme à l'API de renvoi à Renishaw Central.

6.3.11.9 La méthode `Shutdown`

Cette méthode est appelée par DataHUB lorsque toutes les données d'une fabrication ont été envoyées au plug-in, ou lorsque l'appel *Initialise* au plug-in a échoué. Dans cette méthode, le plug-in doit effectuer le traitement final des données de fabrication, libérer les ressources acquises, etc. Cette méthode ne dispose d'aucun paramètre.

Paramètres :

Aucun.

Retour : `string`

Le plug-in doit renvoyer une chaîne de caractères contenant soit :

- le mot « Success », si la méthode s'est déroulée correctement, ou
- autant de détails que souhaités sur la cause de l'échec.

Si DataHUB reçoit une chaîne non conforme, le contenu de la chaîne renvoyée est ajouté au journal d'audit pour un diagnostic ultérieur.

6.4 API de plug-in V2

Cette section sert de guide pour créer, tester et déployer un composant logiciel plug-in V2.0 pour DataHUB. Le cycle de développement du plug-in commence par l'installation de DataHUB sur le PC de développement, puis par la création, à l'aide de Visual Studio, d'un ensemble .NET contenant le code du plug-in. Grâce à la configuration de débogage Visual Studio appropriée, le plug-in peut être testé et débogué pour vérifier son efficacité avant son déploiement final sur un PC de collecte de données (PCCD) sur un système AM de production.

Cette section définit la structure d'un *flux de jetons* de fabrication. Un exemple de code montre ensuite comment écrire un plug-in de traitement de flux de jetons à l'aide de l'API V2.0. Le deuxième exemple de code illustre l'utilisation de *filtres* pour traiter un flux de jetons. Ensuite, la mise en œuvre du plug-in *ExportPackets* (tel qu'il est distribué avec le logiciel DataHUB) est recréée. Les derniers exemples de codes montrent comment les résultats d'un plug-in peuvent être envoyés à Renishaw Central pour apparaître avec d'autres données collectées par les systèmes de fabrication additive de Renishaw.

La section se termine par une définition de l'API V2.0.

6.4.1 Activités préalables

Les documents suivants doivent être lus parallèlement au présent document :

- Guide d'installation des logiciels InfiniAM® et DataHUB (Réf. Renishaw H-5800-6843)
- Manuel d'utilisation de DataHUB (Réf. Renishaw H-5800-6851)
- Manuel de l'utilisateur de Renishaw Licence Manager v2.x

Avant de poursuivre, les actions suivantes doivent être effectuées sur le PC qui sera utilisé pour le développement du plug-in :

- Assurez-vous que le PC est équipé d'un GPU correspondant aux exigences requises par DataHUB (voir la spécification matérielle du PC de collecte de données dans le guide d'installation des logiciels InfiniAM et DataHUB).
- Vérifiez que les dernières versions des pilotes pour le GPU sont installées.
- Assurez-vous que le serveur de licences dispose de suffisamment de licences appropriées pour DataHUB et l'API Spectral.
- Vérifiez l'accès au réseau du serveur de licences.
- Assurez-vous qu'une version de Microsoft Visual Studio prenant en charge .NET Framework 4.7.2 est installée.

6.4.2 Flux de jetons

6.4.2.1 Structure du flux de jetons

Le service DataHUB transforme les données de surveillance de procédé d'une fabrication en un flux de jetons qui sera consommé par des plug-ins V2.0. L'ordre et le type des jetons dans un flux de jetons forment une description pseudo-hiérarchique des données de surveillance de procédé de fabrication.

Les jetons d'un flux se répartissent en deux catégories : les jetons de début et de fin par paires et les jetons délimités par les jetons de début et de fin par paires.

6.4.2.2 Le flux pour un laser, pour une couche

Les données de surveillance de procédé d'un laser pour une couche d'une fabrication sont représentées dans le flux de jetons suivant :

```
StartOfLayerLaserData (Layer, Laser)
   DataSourceInformation
   Sample
   Sample
   …
   Sample
EndOfLayerLaserData (Layer, Laser)
```

Notez que la paire de jetons `StartOfLayerLaserData` et `EndOfLayerLaserData` marquent une limite conceptuelle autour des jetons spécifiques au laser.

Le jeton `DataSourceInformation` contient des valeurs issues des sources de données de surveillance de procédé (c'est-à-dire des fichiers DAT) contenant les données des échantillons.

Chaque jeton `Sample` contient les valeurs des données de surveillance de procédé recueillies par les modules matériel LaserVIEW et MeltVIEW pour un seul échantillon de 100 KHz.

6.4.2.3 Le flux pour une couche

Le jeu reprenant toutes les données de surveillance de procédé pour une couche est combiné dans le flux de jetons, en répétant la structure par laser ci-dessus une fois pour chaque laser de la machine :

```
StartOfLayerData (Layer)
   StartOfLayerLaserData (Layer, Laser)
   …
   EndOfLayerLaserData (Layer, Laser)
   StartOfLayerLaserData (Layer, Laser)
      …
   EndOfLayerLaserData (Layer, Laser)
   StartOfLayerLaserData (Layer, Laser)
      …
```

```
    EndOfLayerLaserData (Layer, Laser)
    StartOfLayerLaserData (Layer, Laser)
       …
    EndOfLayerLaserData (Layer, Laser)
EndOfLayerData (Layer)
```

Là encore, les jetons spécifiques à la couche sont délimités par la paire de jetons `StartOfLayerData` et `EndOfLayerData`.

6.4.2.4 Le flux pour une fabrication

L'ensemble des données relatives à une fabrication forme un flux de jetons répétant la structure par couche susmentionnée pour chaque couche :

```
StartOfLayerData (Layer)
   …
EndOfLayerData (Layer)
StartOfLayerData (Layer)
   …
EndOfLayerData (Layer)
StartOfLayerData (Layer)
   …
EndOfLayerData (Layer)
StartOfLayerData (Layer)
   …
EndOfLayerData (Layer)
```

6.4.2.5 Fractionnements (Splits)

Un détail mineur a été omis dans la structure du flux de jetons que nous venons de décrire, à savoir les jetons `Split`. Les jetons `Split` sont ajoutés au flux de jetons pour marquer les points où les données changent de caractère. Ils sont ajoutés en tant que signal générique, en plus des jetons `StartOf/EndOf`… plus spécifiques. Lors du traitement d'un flux de jetons, il est souvent plus simple d'agir sur ce jeton générique plutôt que sur des marqueurs spécifiques.

6.4.2.6 Définition complète d'un flux de jetons de fabrication

La définition complète d'un flux de jetons pour une fabrication est la suivante :

```
StartOfLayerData (Layer)
   StartOfLayerLaserData (Layer, Laser)
      DataSourceInformation
      Split
      Sample
      Sample
      …
      Sample
      Split
   EndOfLayerLaserData (Layer, Laser)
   Split
   …
EndOfLayerData (Layer)
StartOfLayerData (Layer)
   …
EndOfLayerData (Layer)
```

6.4.2.7 Ajout d'un type de jeton

L'API définit une classe de base `Token` à partir de laquelle tous les jetons sont dérivés. Pour ajouter un nouveau type de jeton au flux de jetons, il faut le dériver de cette classe :

```
public class NewTokenType : Token
{
}
```

Un catalogue complet des jetons d'API est disponible à la section 6.4, « API de plug-in V2 ».

6.4.3 Installation de DataHUB pour les développement de plug-ins

Le service DataHUB doit être installé sur le PC de développement. Lors de l'installation, vous avez la possibilité de choisir les licences à récupérer. Assurez-vous que l'option **LaserVIEW/MeltVIEW Pluggability** (Ajout de plug-ins LaserVIEW/MeltVIEW) est cochée :

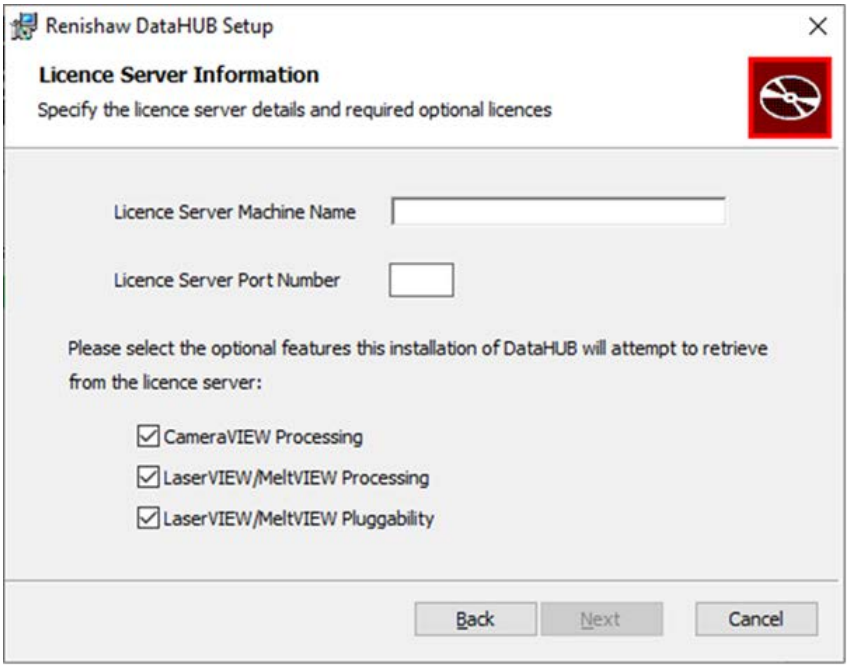


Figure 21 Octroi de licences DataHUB

6.4.4 Création d'un plug-in dans Visual Studio

Les sections suivantes de ce document expliquent comment créer une solution de plug-in à l'aide de Visual Studio.

Les plug-ins sont écrits en langage C# et définissent une classe publique mettant en œuvre le jeu spécifique de méthodes publiques que DataHUB utilise pour identifier, valider et utiliser les instances de plug-in lors du traitement des données de surveillance de procédé d'une fabrication.

REMARQUE : bien que le flux de travail présenté ci-dessous utilise Microsoft Visual Studio 2019, il est globalement similaire dans les autres versions de Microsoft Visual Studio.

6.4.4.1 Création de la solution

Lancez Visual Studio et choisissez *Create a new project* (Créer un nouveau projet). Parmi les options disponibles, choisissez *Class Library (.NET Framework)* (Bibliothèques de classes (.NET Framework))

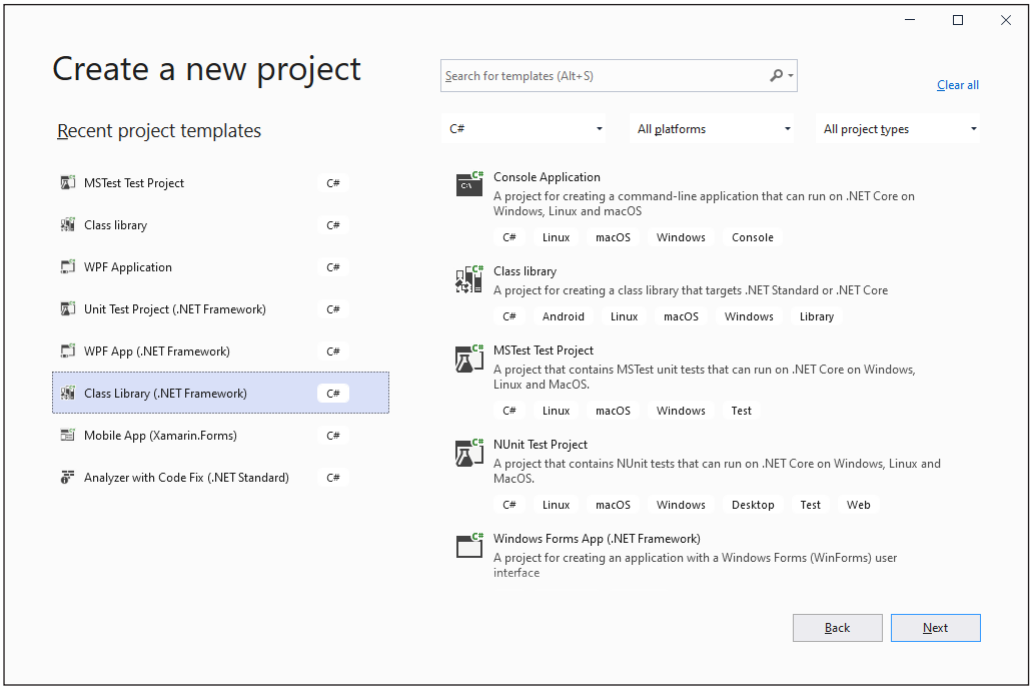


Figure 22 Création de la solution

Ensuite, configurez le projet :

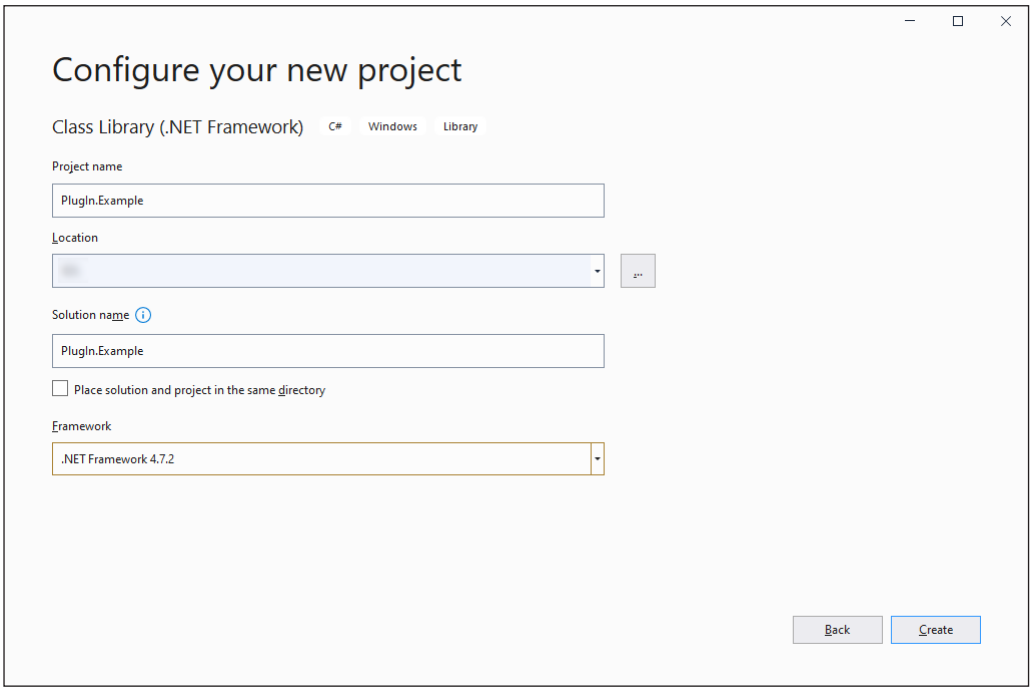


Figure 23 Configuration du projet

DataHUB exige qu'un assembly de plug-in commence par **PlugIn**. (Le mot « PlugIn » suivi d'un point). Sélectionnez un dossier de votre choix dans la zone *Location* (Emplacement) et créez la solution.

REMARQUE : DataHUB prend en charge les plug-ins développés pour les architectures x86/x64/tout processeur. Il est recommandé d'utiliser Framework 4.7.2 ou une version ultérieure.

6.4.4.2 Ajout d'une référence à l'assembly de l'API du plug-in

Pour développer un plug-in pour l'API V2.0, une référence à l'assembly d'API du plug-in est nécessaire. Cliquez avec le bouton droit sur la solution **References** (Références), puis sélectionnez **Add Reference...** (Ajouter une référence).

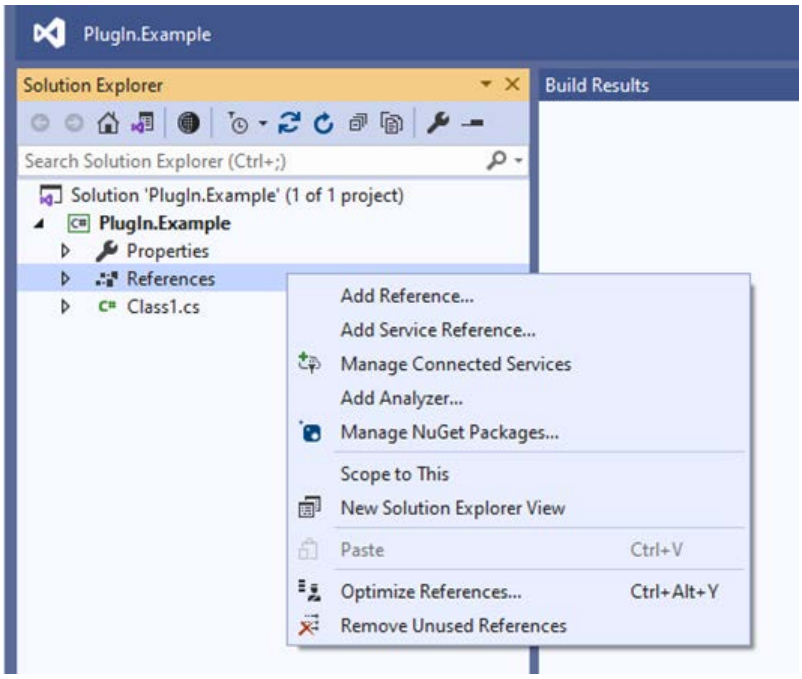


Figure 24 Ajouter une référence

Dans le **Reference Manager** (Gestionnaire de références), cliquez sur **Browse…** (Parcourir) et localisez le fichier « PlugIns.API.v2.dll » dans le dossier d'installation de DataHUB (soit <$PROGRAMFILES$\Renishaw\DataHUB>) :

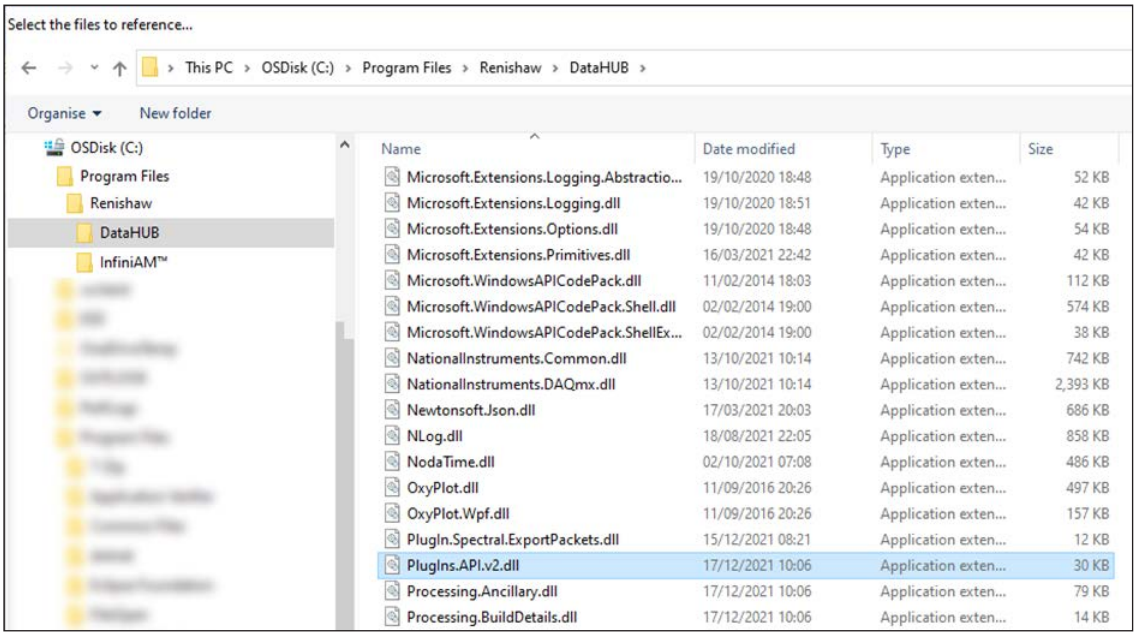


Figure 25 Assembly d'API

L'assembly sera ensuite répertorié dans la liste des dépendances du projet de plug-in :

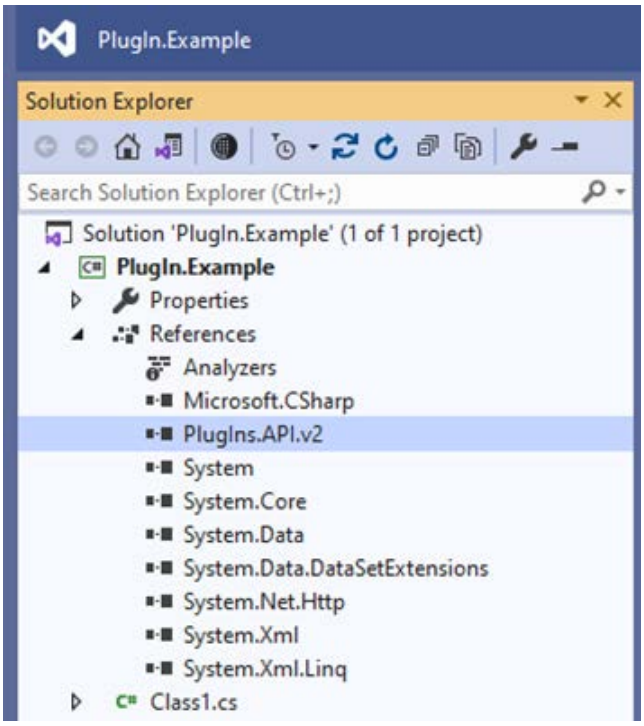


Figure 26 Assembly d'API du plug-in référencé en tant que dépendance

6.4.4.3 Dénomination du type de plug-in

Le seul fichier source par défaut Class1.cs contient la définition de Class1, et nous l'utiliserons comme point de départ.

```
namespace PlugIn.Example
{
    public class Class1
    {
    }
}
```

Une fois que DataHUB a détecté un assembly de plug-in potentiel, il recherche un type public `PlugIn`, et il faut donc renommer `Class1` en `PlugIn`.

```
namespace PlugIn.Example
{
    public class PlugIn
    {
    }
}
```

6.4.4.4 Ajout de l'API

En présence d'un type de plug-in potentiel, DataHUB exige qu'il mette en œuvre les méthodes publiques suivantes :

```
public PlugIn() (generated automatically)
public static string APIVersion()
public static string DisplayName()
public string Initialise(…)
public string Process(…)
public string Shutdown()
```

Ajoutez les déclarations **using** et mises en œuvre de méthodes suivantes à la classe :

```
using System.Collections.Generic;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types.Tokens;
public class PlugIn
{
   public static string APIVersion() => "2";
   public static string DisplayName() => "Example plug-in";
   public string Initialise(string initialisationData) => InitialiseReturns.Success();
   public string Process(IReadOnlyCollection<Token> tokens) => ProcessReturns.Success();
   public string Shutdown() => ShutdownReturns.Success();
}
```

Notez que toutes les méthodes de l'API renvoient une `string`. Le contenu de la chaîne renvoyée est utilisé pour signaler à DataHUB le résultat de la méthode. Le rôle de cette valeur de retour sera expliqué plus loin, au fur et à mesure que chaque méthode sera détaillée.

6.4.4.5 La méthode `APIVersion` statique

Cette méthode indique à DataHUB la version de l'API utilisée par le plug-in. Pour utiliser l'API V2.0, cette méthode doit renvoyer exactement « 2 » :

```
   public static string APIVersion() => "2";
```

6.4.4.6 La méthode `DisplayName` statique

Le contenu de la chaîne renvoyée par cette méthode est utilisé comme nom de plug-in affiché dans DataHUB Monitor et pour la création du dossier de sortie du plug-in, etc. Bien qu'il ne soit pas obligatoire que ce nom soit unique, cela permet d'identifier le plug-in spécifique lorsque, par exemple, plusieurs versions sont installées sur un PCCD.

```
   public static string DisplayName() => "Example plug-in";
```

6.4.4.7 La méthode `Initialise`

Cette méthode est appelée par DataHUB au début d'une fabrication, avant que les données de la couche ne soient envoyées au plug-in. Elle transmet donc les données des *invariants de la fabrication* au plug-in. La mise en œuvre minimale consiste à renvoyer que le plug-in s'est initialisé sans erreur :

```
public string Initialise(string initialisationData) => InitialiseReturns.Success();
```

6.4.4.8 La méthode `Process`

Cette méthode est appelée par DataHUB lorsque toutes les données d'une couche ont été reçues. La mise en œuvre minimale consiste à renvoyer le fait que le plug-in a traité les données sans erreur :

```
public string Process(IReadOnlyCollection<Token> tokens) => ProcessReturns.Success();
```

6.4.4.9 La méthode `Shutdown`

Cette méthode est appelée par DataHUB lorsque toutes les données d'une fabrication ont été envoyées au plug-in, ou lorsque tout appel précédent au plug-in a échoué. Une fois de plus, la mise en œuvre minimale consiste en un renvoi sans échec :

```
public string Shutdown() => ShutdownReturns.Success();
```

6.4.4.10 Déploiement du plug-in pour le débogage

Construisez la solution en mode *Debug*, ouvrez un navigateur de fichiers et localisez le dossier de sortie qui contiendra un fichier *.dll* nommé conformément au nom donné au projet, par exemple, *PlugIn.Example.dll*. (Des fichiers auxiliaires tels que des symboles de débogage, etc. peuvent également être présents.)

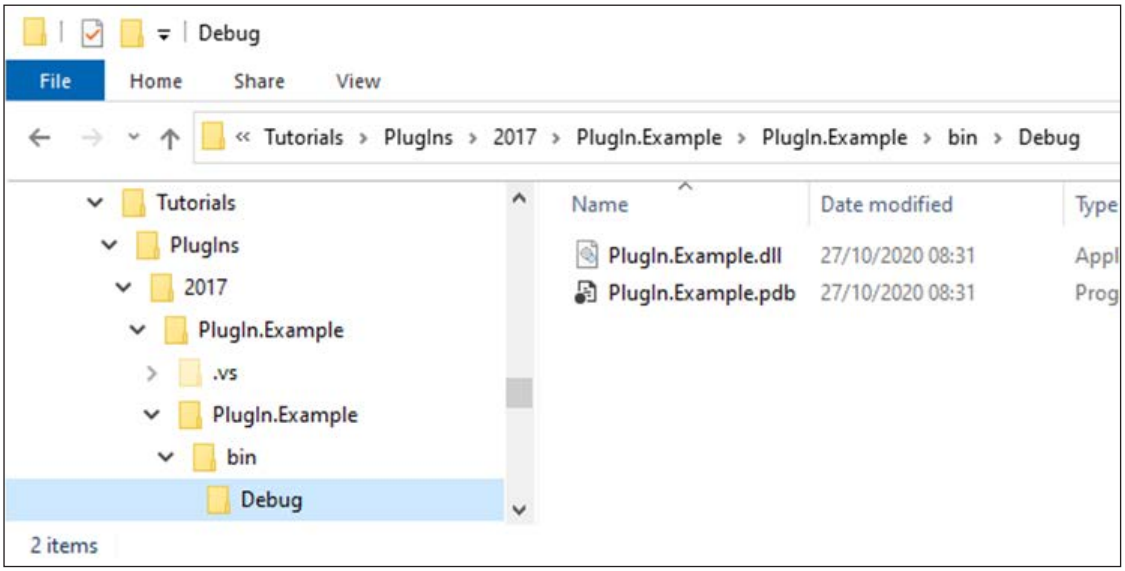


Figure 27 Fichiers binaires d'un plug-in

Copiez ce dossier et collez-le dans le sous-dossier PlugIns dans le dossier de données de DataHUB, soit <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>.

REMARQUE : en général, $PROGRAMDATA$ se trouve sous <C:\ProgramData>, mais il peut être masqué, dans ce cas, il faudra cocher l'option en suivant le chemin **View, Show/hide, Hidden items** (Afficher, Afficher/masquer, Éléments masqués).

Des privilèges d'administrateur peuvent être requis pour modifier le contenu de ce dossier.

Renommez le dossier *Debug* afin qu'il corresponde à celui du plug-in (dans l'image ci-dessous, le dossier à été nommé *PlugIn.Example.*).

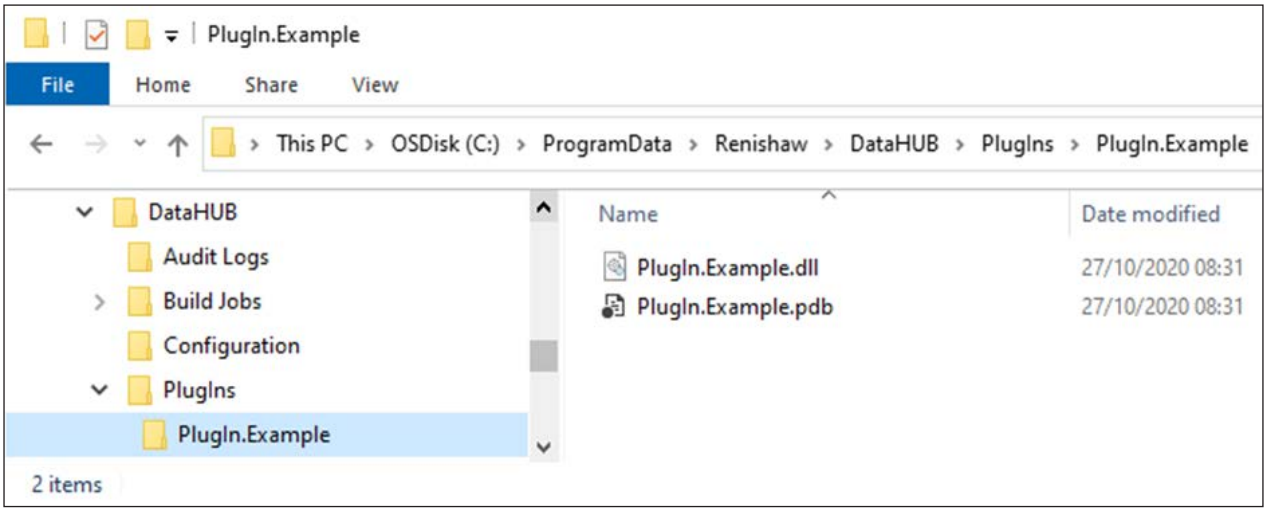


Figure 28 Plug-in déployé

6.4.4.11 Enregistrement du plug-in

DataHUB Service doit être redémarré pour enregistrer le nouveau plug-in. Pour ce faire, utilisez l'application *Services* en cliquant avec le bouton droit de la souris sur *DataHUB Service* et cliquez sur *Restart* (Redémarrer) :

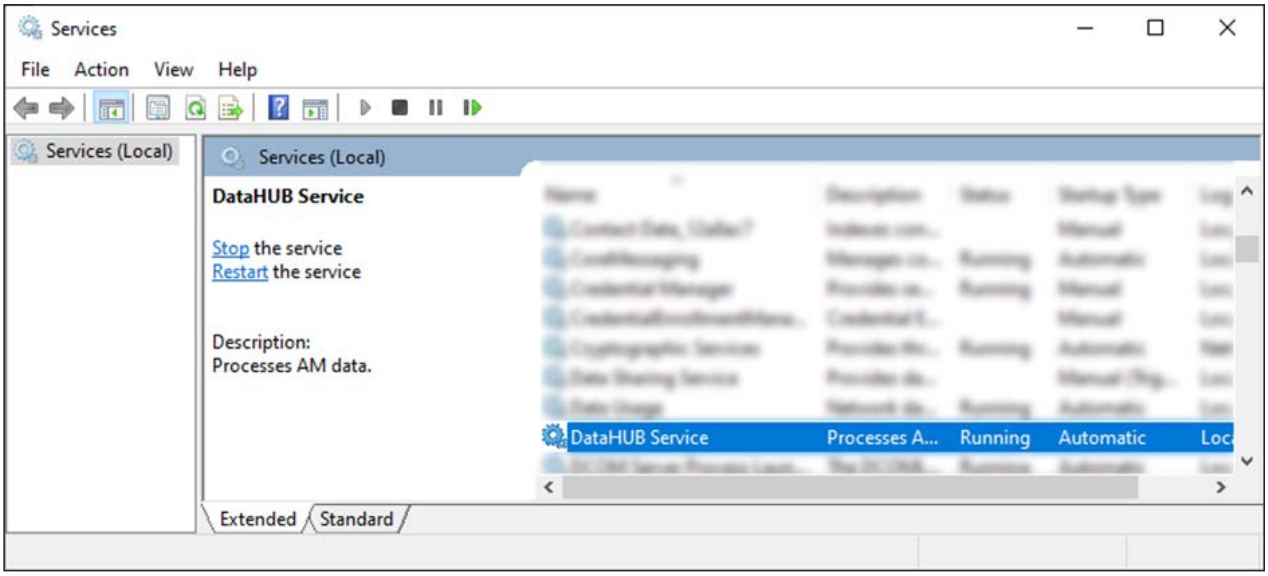


Figure 29 Application Services de Windows

Après quelques secondes, le service DataHUB s'affichera à l'état *Running* (En cours d'exécution).

6.4.4.12 Vérification de l’enregistrement

DataHUB génère un fichier journal au fur et à mesure de son exécution, détaillant les événements importants tels que les démarrages de fabrication, les erreurs de fabrication et, bien sûr, l’audit des plug-ins. Dans l’Explorateur de fichiers, localisez le dossier des journaux <$PROGRAMDATA$\Renishaw\DataHUB\Audit Logs>.

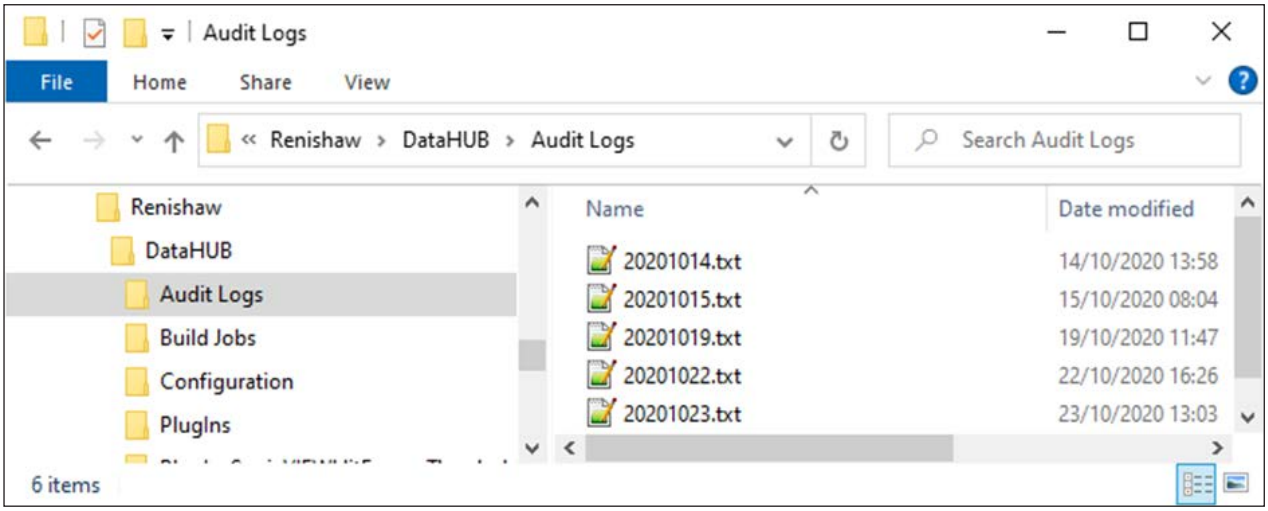


Figure 30 Dossier des journaux de DataHUB

Ouvrez le fichier journal le plus récent (il doit porter la date d’aujourd’hui) faites défiler jusqu’à la fin et recherchez les lignes détaillant la découverte de plug-ins valides :

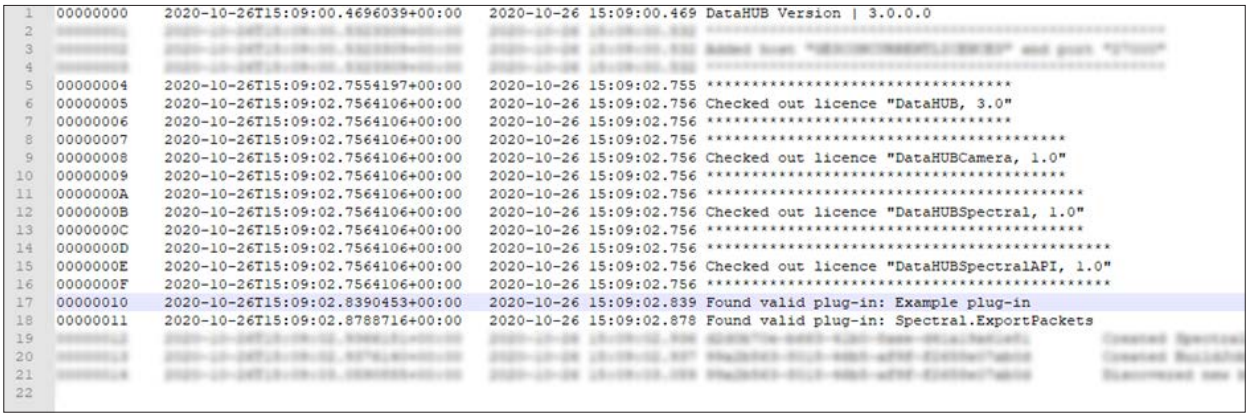


Figure 31 Plug-ins valides trouvés par DataHUB

6.4.4.13 Délais d'expiration

Comme DataHUB ne peut pas garantir que chaque appel à une instance de plug-in fera l'objet d'un renvoi en temps voulu, il adopte une politique de délai d'expiration. Si un plug-in n'aboutit pas dans un certain délai, il est considéré comme défectueux et il est stoppé. Les délais d'expiration par défaut sont :

- Pour *Initialise*, 20 secondes
- Pour *Process* (méthode partagée avec la méthode *ProcessPackets* de l'API V1.0), 200 secondes
- Pour *Shutdwon*, 20 secondes

Pendant le débogage, il arrive souvent que ces délais soient dépassés, ce qui entraîne l'arrêt prématuré du plug-in par DataHUB. Pour permettre plus de temps de débogage, les délais d'expiration peuvent être augmentés en modifiant le fichier <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns\PlugInsConfiguration.xml> dans un éditeur XML ou un éditeur de texte approprié.

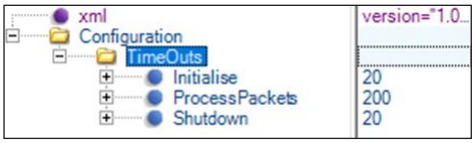


Figure 32 Délais d'expiration par défaut des plug-ins

REMARQUE : DataHUB Service doit être redémarré pour que ces changements de délai d'expiration prennent effet.

6.4.4.14 Débogage du plug-in

Lorsque le plug-in a été déployé avec succès, il est maintenant possible de déboguer son exécution. Le plug-in lui-même n'est pas directement exécutable, mais en associant le débogueur Visual Studio à DataHUB Service et en lançant une tâche de fabrication, la mise en œuvre du plug-in sera invoquée par le service, et le jeu de points d'arrêt défini dans le code sera atteint.

REMARQUE : pour effectuer le débogage via le service, Visual Studio peut exiger des privilèges d'administrateur. Ces droits peuvent être accordés en cliquant avec le bouton droit de la souris sur le raccourci du programme Visual Studio et en sélectionnant **Run as administrator** (Exécuter en tant qu'administrateur).

Dans Visual Studio, placez des points d'arrêt au début des méthodes *Initialise*, *Process* et *Shutdown* :

Figure 33 Points d'arrêt définis dans *Initialise* et *ProcessPackets*

Ouvrez le menu **Debug** (Déboguer) et sélectionnez **Attach to Process...** (Attacher au processus) :

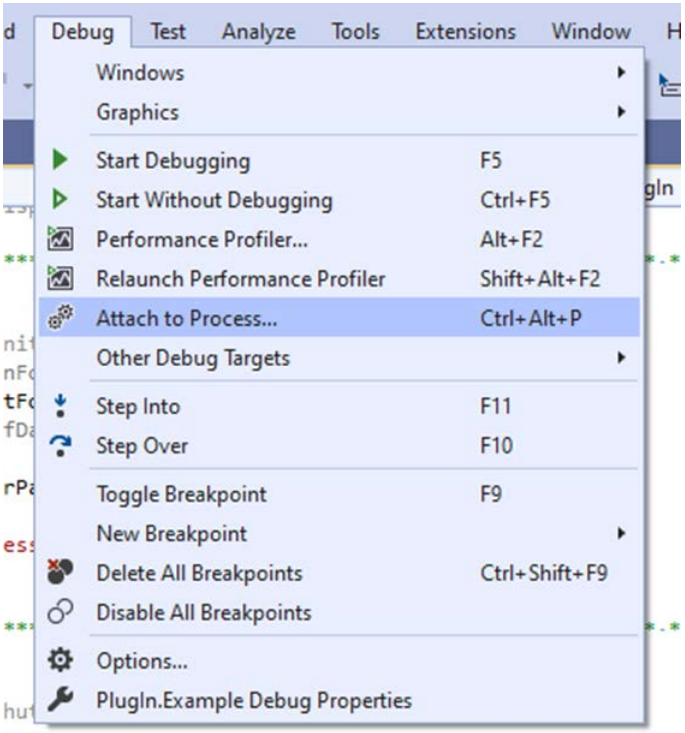


Figure 34 Attachement à un processus pour le débogage

Dans la liste des processus en cours d'exécution, localisez *DataHUB Service.exe*. Vous devrez peut-être cocher la case **Show processes from all users** (Afficher les processus de tous les utilisateurs) :

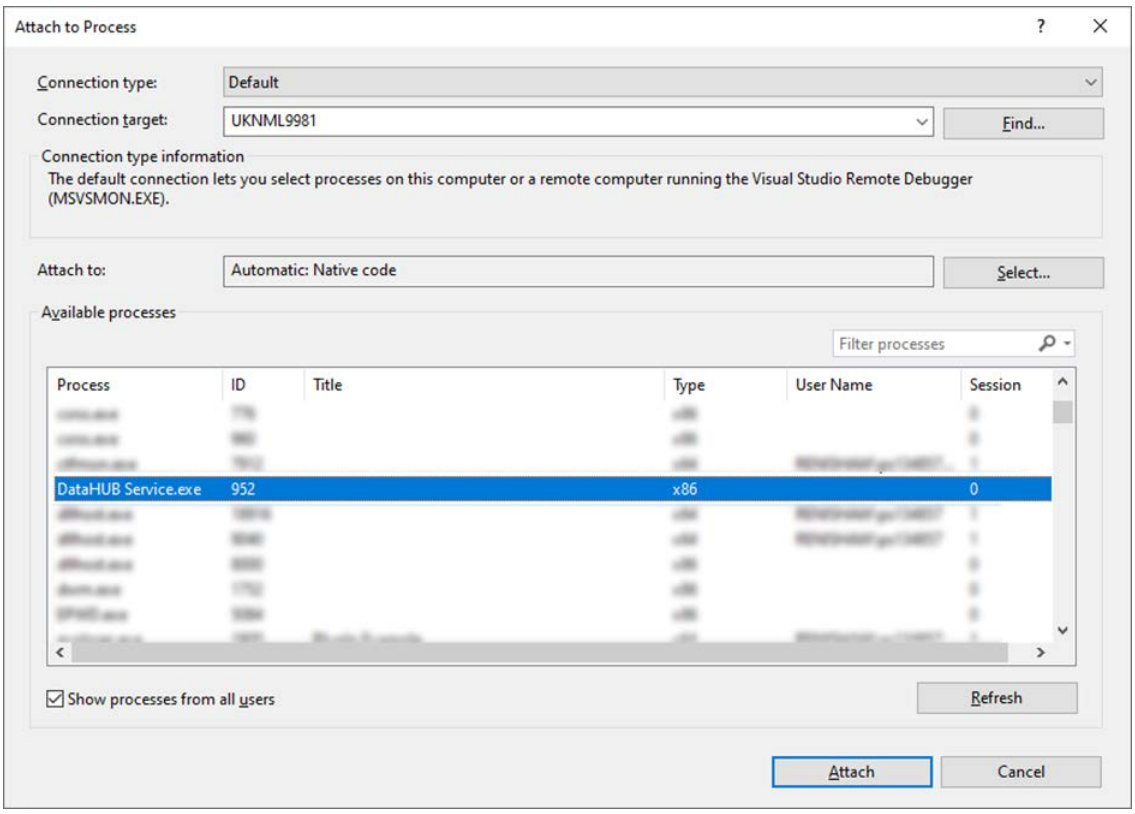


Figure 35 Processus DataHUB Service

Appuyez sur le bouton **Attach** (Attacher) et Visual Studio va démarrer une session de débogage pour le service DataHUB.

Pour que DataHUB appelle le plug-in et donc pour que les points d'arrêt soient atteints, il faut lancer une fabrication. Vous pouvez le faire via DataHUB Generator ou, si vous avez déjà généré des fabrications, simplement en dupliquant un dossier de fabrication dans <$PROGRAMDATA$\Renishaw\DataHUB\Build Jobs>.

DataHUB détectera automatiquement la nouvelle fabrication et commencera à traiter les données. L'une des tâches préliminaires du service consiste à créer (en invoquant le constructeur sans paramètre) une instance de chaque plug-in enregistré. Lorsque DataHUB appelle ensuite la méthode *Initialise* de l'instance, le premier point d'arrêt est atteint. Les valeurs des paramètres de la méthode seront celles de la fabrication en question, par exemple :

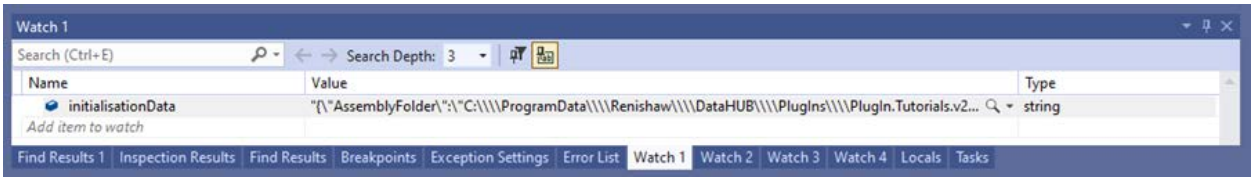


Figure 36 Valeurs des paramètres de la méthode *Initialise*

À partir de ce moment, les données existantes (et, pour une fabrication active, les nouvelles données telles qu'elles arrivent) sont traitées et la méthode *Process* de l'instance est appelée avec des valeurs de paramètres pour les données LaserVIEW et MeltVIEW associées à une couche :

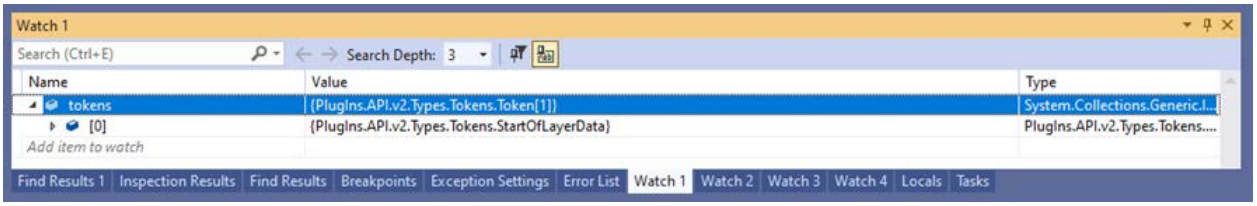


Figure 37 Valeurs des paramètres de la méthode *Process*

6.4.4.15 Déploiement du plug-in pour la production

Pour préparer le déploiement du plug-in sur un PCCD de production, il est essentiel qu'une version finale (*Release*) soit compilée, déployée localement et testée. Une fois la fonctionnalité du plug-in validée et vérifiée, il peut être déployé sur les PCCD de production.

Le processus de déploiement sur un PCDC de production est très similaire à celui du déploiement sur un PC de développement. Le dossier *Release* doit être copié dans le sous-dossier *PlugIns* du dossier de données de DataHUB (<$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>) sur le PCCD et renommé pour correspondre au nom du plug-in. Le service DataHUB doit ensuite être redémarré pour enregistrer le nouveau plug-in : pour ce faire, utilisez l'application *Services*. Le succès ou l'échec de l'enregistrement du nouveau plug-in est consigné dans le journal d'audit.

REMARQUE : DataHUB est conçu pour reprendre toutes les tâches de traitement de données en cours après un redémarrage, il n'est donc pas nécessaire d'attendre que toutes les tâches en cours soient terminées avant de redémarrer. Cependant, seules les nouvelles tâches de traitement des données utiliseront le plug-in récemment enregistré.

6.4.4.16 Diagnostiquer les échecs du plug-in en production

Dans l'environnement de production, DataHUB Monitor affichera (avec d'autres tâches de traitement de la fabrication) l'état de traitement propre au plug-in :

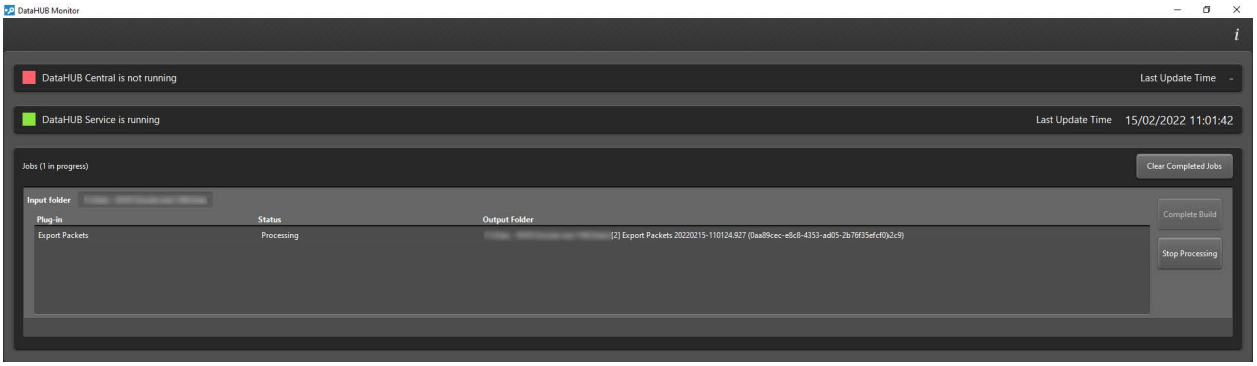


Figure 38 Fabrication en cours d'exécution avec des plug-ins

Si un plug-in est à l'état *Error* (Erreur), le journal d'audit contiendra des enregistrements relatifs à l'échec, soit sous forme d'entrées ajoutées par DataHUB lui-même (p. ex., l'échec du chargement de l'assembly de plug-in en raison de dépendances manquantes), soit par le plug-in via le contenu des valeurs de retour renvoyées par les méthodes *Initialise*, *Process* ou *Shutdown* du plug-in.

6.4.5 Exemple 1 – Traitement d'un flux de jetons

Ce premier exemple de code montre comment traiter un flux de jetons pour extraire les valeurs minimales et maximales pour chaque canal de données de surveillance de procédé, pour chaque laser, pour chaque couche. Plusieurs concepts clés seront abordés :

- Décodage des données d'initialisation
- Traitement des jetons
- Dossiers de sortie de l'instance

6.4.5.1 Création

La mise en œuvre minimale du plug-in est la suivante :

```
using System;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types.Tokens;
namespace PlugIn.Tutorials.v2.Example1
{
   public class PlugIn
   {
      public static string APIVersion() => "2";
      public static string DisplayName() => "Tutorial Example 1";
      public string Initialise(string initialisationData) => InitialiseReturns.Success();
      public string Process(IreadOnlyCollection<Token> tokens) => ProcessReturns.Success();
      public string Shutdown() => ShutdownReturns.Success();
   }
}
```

Les classes utilitaires (*helper*) `InitialiseReturns`, `ProcessReturns` et `ShutdownReturns` de l'API fournissent des méthodes pratiques pour générer des valeurs de retour d'API ; ici, les méthodes *Success* renvoient une chaîne standard « Success » reconnue par DataHUB.

6.4.5.2 Initialisation

Lorsque DataHUB initialise ce plug-in, la première tâche consiste à enregistrer l'emplacement du dossier de sortie unique de l'instance du plug-in pour une utilisation ultérieure lors de l'écriture dans un fichier. La méthode décode (à l'aide de la bibliothèque Newtonsoft.Json) la chaîne `initialisationData` et extrait le chemin du dossier de sortie :

```
public class PlugIn
{
   …
   public string Initialise(string initialisationData)
   {
      var initialisationValues = Jobject.Parse(initialisationData);
      _outputFolder = initialisationValues[“OutputFolder”].Value<string>();
      return InitialiseReturns.Success();
   }
   private string _outputFolder;
   …
}
```

REMARQUE : les données d'initialisation sont codées sous forme d'une chaîne au format JSON.

6.4.5.3 Traitement du flux de jetons

Si l'initialisation est réussie, l'instance du plug-in recevra le flux de jetons de la fabrication. Comme certains jeux de données de surveillance de procédé peuvent être très volumineux, le flux de jetons est fourni par lots ; chaque fois que la méthode `Process` est appelée, le prochain lot de jetons est réceptionné.

La gestion des jetons `Sample` peut désormais être ajoutée à la méthode `Process` :

```
…
public string Process(IreadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         case Sample sample:
            _demandX             = MinMaxForProperty(_demandX,             sample.DemandX);
            _demandY             = MinMaxForProperty(_demandY,             sample.DemandY);
            _demandFocus         = MinMaxForProperty(_demandFocus,         sample.DemandFocus);
            _demandLaserPower    = MinMaxForProperty(_demandLaserPower,    sample.DemandLaserPower);
            _laserModulate       = MinMaxForProperty(_laserModulate,       sample.LaserModulate);
            _meltVIEWPlasma      = MinMaxForProperty(_meltVIEWPlasma,      sample.MeltVIEWPlasma);
            _meltVIEWMeltpool    = MinMaxForProperty(_meltVIEWMeltpool,    sample.MeltVIEWMeltpool);
```

```
            _laserVIEW           = MinMaxForProperty(_laserVIEW,           sample.LaserVIEW);
            _laserOutputPower    = MinMaxForProperty(_laserOutputPower,    sample.LaserOutputPower);
            _laserBackReflection = MinMaxForProperty(_laserBackReflection, sample.LaserBackReflection);
            break;
      }
   return ProcessReturns.SuccessWithoutInformation();
}
private static (double, double) MinMaxForProperty((double Min, double Max) currentValue, double value)
{
   var (min, max) = currentValue;
   return (Math.Min(min, value), Math.Max(max, value));
}
private static (int, int) MinMaxForProperty((int Min, int Max) currentValue, int value)
{
   var (min, max) = currentValue;
   return (Math.Min(min, value), Math.Max(max, value));
}
private (double Min, double Max) _demandX             = (double.MaxValue, double.MinValue);
private (double Min, double Max) _demandY             = (double.MaxValue, double.MinValue);
private (double Min, double Max) _demandFocus         = (double.MaxValue, double.MinValue);
private (int Min, int Max)       _laserModulate       = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _demandLaserPower    = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _meltVIEWPlasma      = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _meltVIEWMeltpool    = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _laserVIEW           = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _laserOutputPower    = (int.MaxValue, int.MinValue);
private (int Min, int Max)       _laserBackReflection = (int.MaxValue, int.MinValue);
…
```

Plusieurs champs sont initialisés, lesquels stockeront le minimum et le maximum en cours d'exécution pour chacune des propriétés des données de surveillance de procédé d'un jeton `Sample`.

Notez que le renvoi de `Process` est devenu `SuccessWithoutInformation()` ce qui indique à DataHUB que le plug-in a terminé le traitement du lot de jetons mais ne souhaite pas ajouter de message aux journaux de DataHUB ; ce changement garantit que les journaux de DataHUB ne sont pas inondés de messages « Success » sans intérêt.

REMARQUE : les flux de jetons sont traités par lots ; la méthode `Process` sera appelée plusieurs fois.

REMARQUE : il n'est pas nécessaire que chaque appel à `Process` renvoie un message consigné dans le journal au service DataHUB.

La valeur des propriétés des données de surveillance de procédé de chaque jeton `Sample` est utilisée pour mettre à jour le minimum et le maximum en cours d'exécution. Notez que certaines propriétés sont de type *double* et *integer* (entier).

REMARQUE : Un jeton `Sample` dispose des propriétés de données de surveillance de procédé suivantes :

- DemandX (double)
- DemandY (double)
- DemandFocus (double)
- DemandLaserPower (integer)
- LaserModulate (integer)
- MeltVIEWPlasma (integer)
- MeltVIEWMeltpool (integer)
- LaserVIEW (entier)
- LaserOutputPower (integer)
- LaserBackReflection (integer)

6.4.5.4 Écriture dans un fichier

La dernière tâche du plug-in consiste à écrire les résultats dans un fichier lorsque le plug-in traite un jeton `EndOfLayerLaserData` :

```
…
public string Process(IreadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         …
         case EndOfLayerLaserData endOfLayerLaserData:
            var layer = endOfLayerLaserData.Layer.Value;
            var laser = endOfLayerLaserData.Laser.Value;
            var outputFilePath = Path.Combine(_outputFolder, “Min & Max for “
                                                          + $”layer {layer}, “
                                                          + $”laser {laser}”
                                                          + “.txt”);
            File.AppendAllText(outputFilePath, FormatMinAndMax(“DemandX”,            _demandX));
            File.AppendAllText(outputFilePath, FormatMinAndMax(“DemandY”,            _demandY));
```

```
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandFocus",          _demandFocus));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandLaserPower",     _demandLaserPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserModulate",        _laserModulate));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWPlasma",       _meltVIEWPlasma));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWMeltpool",     _meltVIEWMeltpool));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserVIEW",            _laserVIEW));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserOutputPower",     _laserOutputPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserBackReflection", _laserBackReflection));
            _demandX              = (double.MaxValue, double.MinValue);
            _demandY              = (double.MaxValue, double.MinValue);
            _demandFocus          = (double.MaxValue, double.MinValue);
            _laserModulate        = (int.MaxValue, int.MinValue);
            _demandLaserPower     = (int.MaxValue, int.MinValue);
            _meltVIEWPlasma       = (int.MaxValue, int.MinValue);
            _meltVIEWMeltpool     = (int.MaxValue, int.MinValue);
            _laserVIEW            = (int.MaxValue, int.MinValue);
            _laserOutputPower     = (int.MaxValue, int.MinValue);
            _laserBackReflection  = (int.MaxValue, int.MinValue);
            return ProcessReturns.Success($"Processed layer {layer}, laser {laser}");
        …
    }
}
private static string FormatMinAndMax(string propertyName, (double Min, double Max) propertyMinAndMax)
{
   return $"{propertyName}\t"
        + $"{propertyMinAndMax.Min:F3}\t"
        + $"{_ propertyMinAndMax.Max:F3}\t"
        + $"{Environment.NewLine}";
}
private static string FormatMinAndMax(string propertyName, (int Min, int Max) propertyMinAndMax)
{
   return $"{propertyName}\t"
        + $"{propertyMinAndMax.Min}\t"
        + $"{propertyMinAndMax.Max}\t"
        + $"{Environment.NewLine}";
}
…
```

Le chemin de sortie du fichier de résultats est construit en combinant le dossier de sortie mis en cache lors de l'initialisation et les numéros de couche et de laser des jetons qui viennent d'être traités. Les valeurs minimales et maximales de chaque propriété `Sample` sont écrites dans le fichier, puis sont réinitialisées en vue du traitement des jetons de la couche suivante.

Le plug-in renvoie un résultat « Success » avec un message qui sera audité par DataHUB.

REMARQUE : DataHUB attribue à chaque instance d'un plug-in un chemin d'accès unique au dossier de sortie, qui doit être utilisé pour enregistrer les résultats de sortie du plug-in.

6.4.5.5 Mise en œuvre finale

Le plug-in complet est le suivant :

```
using System;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Types.Tokens;
using PlugIns.API.v2.Helpers;
namespace PlugIn.Tutorials.v2.Example1
{
   public class PlugIn
   {
      public static string APIVersion() => "2";
      public static string DisplayName() => "Tutorial Example 1";
      public string Initialise(string initialisationData)
      {
         var initialisationValues = Jobject.Parse(initialisationData);
         _outputFolder = initialisationValues["OutputFolder"].Value<string>();
         return InitialiseReturns.Success();
      }
      public string Process(IreadOnlyCollection<Token> tokens)
      {
         foreach (var token in tokens)
            switch (token)
            {
               case Sample sample:
                  _demandX             = MinMaxForProperty(_demandX,             sample.DemandX);
                  _demandY             = MinMaxForProperty(_demandY,             sample.DemandY);
                  _demandFocus         = MinMaxForProperty(_demandFocus,         sample.DemandFocus);
                  _demandLaserPower    = MinMaxForProperty(_demandLaserPower,    sample.DemandLaserPower);
```

```
            _laserModulate       = MinMaxForProperty(_laserModulate,       sample.LaserModulate);
            _meltVIEWPlasma      = MinMaxForProperty(_meltVIEWPlasma,      sample.MeltVIEWPlasma);
            _meltVIEWMeltpool    = MinMaxForProperty(_meltVIEWMeltpool,    sample.MeltVIEWMeltpool);
            _laserVIEW           = MinMaxForProperty(_laserVIEW,           sample.LaserVIEW);
            _laserOutputPower    = MinMaxForProperty(_laserOutputPower,    sample.LaserOutputPower);
            _laserBackReflection = MinMaxForProperty(_laserBackReflection, sample.LaserBackReflection);
          break;
        case EndOfLayerLaserData endOfLayerLaserData:
            var layer = endOfLayerLaserData.Layer.Value;
            var laser = endOfLayerLaserData.Laser.Value;
            var outputFilePath = Path.Combine(_outputFolder, "Min & Max for "
                                                           + $"layer {layer}, "
                                                           + $"laser {laser}"
                                                           + ".txt");
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandX",          _demandX));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandY",          _demandY));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandFocus",      _demandFocus));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserModulate",    _laserModulate));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandLaserPower",
                                                               _demandLaserPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWPlasma",
                                                               _meltVIEWPlasma));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWMeltpool",
                                                               _meltVIEWMeltpool));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserVIEW",
                                                               _laserVIEW
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserOutputPower",
                                                               _laserOutputPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserBackReflection",
                                                               _laserBackReflection));
            _demandX             = (double.MaxValue, double.MinValue);
            _demandY             = (double.MaxValue, double.MinValue);
            _demandFocus         = (double.MaxValue, double.MinValue);
            _laserModulate       = (int.MaxValue, int.MinValue);
            _demandLaserPower    = (int.MaxValue, int.MinValue);
            _meltVIEWPlasma      = (int.MaxValue, int.MinValue);
            _meltVIEWMeltpool    = (int.MaxValue, int.MinValue);
            _laserVIEW           = (int.MaxValue, int.MinValue);
            _laserOutputPower    = (int.MaxValue, int.MinValue);
            _laserBackReflection = (int.MaxValue, int.MinValue);
```

```
                return ProcessReturns.Success($"Processed layer {layer}, laser {laser}");
            }
        return ProcessReturns.SuccessWithoutInformation();
    }
    public string Shutdown() => ShutdownReturns.Success();
    private string _outputFolder;
    private (double Min, double Max) _demandX             = (double.MaxValue, double.MinValue);
    private (double Min, double Max) _demandY             = (double.MaxValue, double.MinValue);
    private (double Min, double Max) _demandFocus         = (double.MaxValue, double.MinValue);
    private (int Min, int Max)       _laserModulate       = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _demandLaserPower    = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _meltVIEWPlasma      = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _meltVIEWMeltpool    = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _laserVIEW           = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _laserOutputPower    = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _laserBackReflection = (int.MaxValue, int.MinValue);

    private static string FormatMinAndMax(string propertyName,
                                          (double Min, double Max) propertyMinAndMax)
    {
        return $"{propertyName}\t"
             + $"{propertyMinAndMax.Min:F3}\t"
             + $"{propertyMinAndMax.Max:F3}\t"
             + $"{Environment.NewLine}";
    }
    private static string FormatMinAndMax(string propertyName,
                                          (int Min, int Max) propertyMinAndMax)
    {
        return $"{propertyName}\t"
             + $"{propertyMinAndMax.Min}\t"
             + $"{propertyMinAndMax.Max}\t"
             + $"{Environment.NewLine}";
    }
    private static (double, double) MinMaxForProperty((double Min, double Max) currentValue,
                                                      double value)
    {
        var (min, max) = currentValue;
        return (Math.Min(min, value), Math.Max(max, value));
    }
    private static (int, int) MinMaxForProperty((int Min, int Max) currentValue,
```

```
                                                int value)
        {
            var (min, max) = currentValue;
            return (Math.Min(min, value), Math.Max(max, value));
        }
    }
}
```

6.4.6 Filtres

Un *filtre* dans l'API est une unité de code qui traite un jeton d'entrée et génère zéro ou plusieurs jetons en sortie. Un filtre peut, après un traitement approprié, émettre le jeton d'entrée, émettre de nouveaux jetons ou faire disparaître (avaler) le jeton d'entrée. Via ces méthodes, un filtre transforme un flux de jetons d'entrée en un nouveau flux de jetons de sortie.

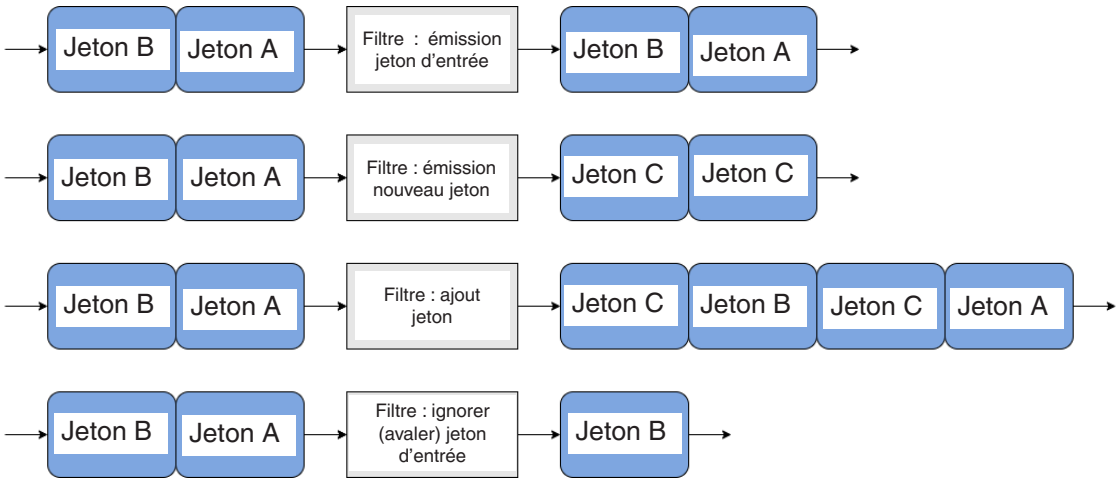


Figure 39 Méthodes de sortie de filtre

6.4.6.1 Pipelines

Pour deux filtres donnés, l'API fournit une méthode permettant de les combiner dans un *pipeline*. Le fait qu'un pipeline ait la même signature qu'un filtre unique signifie que les pipelines et les filtres peuvent se combiner pour former des pipelines plus longs et plus complexes. Lorsqu'un pipeline traite un flux de jetons, la sortie du premier filtre constitue l'entrée du second, et ainsi de suite jusqu'à ce que la sortie du dernier filtre constitue la sortie finale du pipeline.

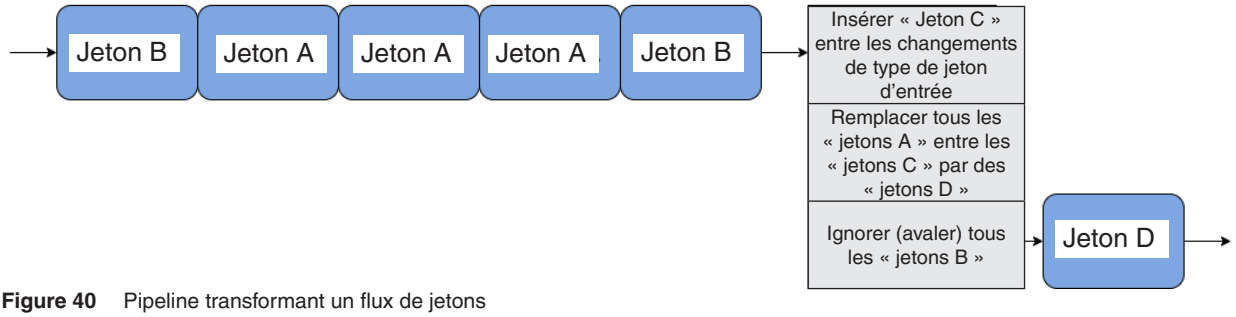


Figure 40 Pipeline transformant un flux de jetons

6.4.6.2 Regroupement des échantillons en paquets à l'aide d'un pipeline de filtres

Cette section illustre la transformation d'un flux de jetons d'entrée composé d'échantillons, via plusieurs étapes de filtrage, en un flux de jetons composé de données synthétisées.

Les filtres qui composent le pipeline sont les suivants :

- Fractionner en cas de changement de **LaserModulate** (Modulation laser)
- Émettre lorsque le laser se déclenche
- Fractionner en cas de changement de **DemandX** (DemandeX)
- Fractionner en cas de changement de **DemandY** (DemandeY)
- Regrouper en paquets d'échantillons
- Synthétiser les paquets d'échantillons

6.4.6.2.1 Flux de jetons d'entrée

Le flux de jetons d'entrée se compose de sept jetons *Sample* :

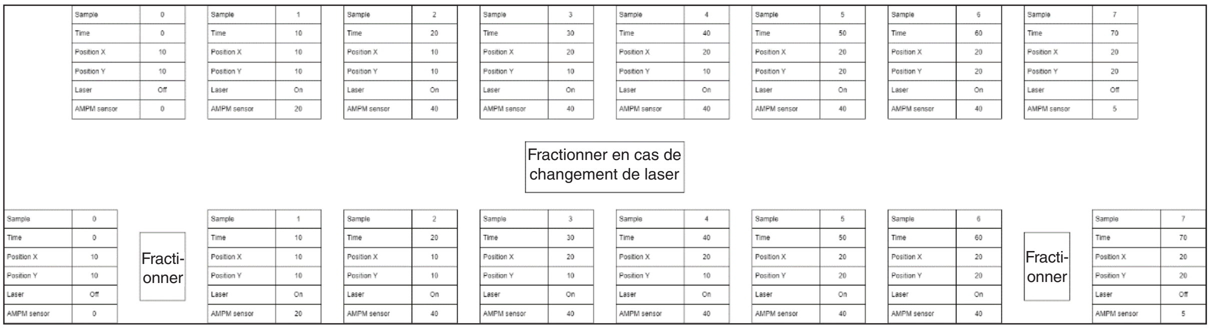
Sample	0	Sample	1	Sample	2	Sample	3	Sample	4	Sample	5	Sample	6	Sample	7
Time	0	Time	10	Time	20	Time	30	Time	40	Time	50	Time	60	Time	70
Position X	10	Position X	10	Position X	10	Position X	20	Position X	20	Position X	20	Position X	20	Position X	20
Position Y	10	Position Y	10	Position Y	10	Position Y	10	Position Y	10	Position Y	20	Position Y	20	Position Y	20
Laser	Off	Laser	On	Laser	On	Laser	On	Laser	On	Laser	On	Laser	On	Laser	Off
AMPM sensor	0	AMPM sensor	20	AMPM sensor	40	AMPM sensor	40	AMPM sensor	40	AMPM sensor	40	AMPM sensor	40	AMPM sensor	5

Le fonctionnement du laser peut être retracé en analysant les différentes valeurs de chaque échantillon :

- Le laser commence à la position (10, 10) au temps 0 et ne se déclenche pas
- Le laser se déclenche au temps 20
- Le laser se déplace à la position (20, 10) au temps 30
- Le laser se déplace à la position (20, 20) au temps 50
- Le laser s'arrête, ne se déclenche plus au temps 70

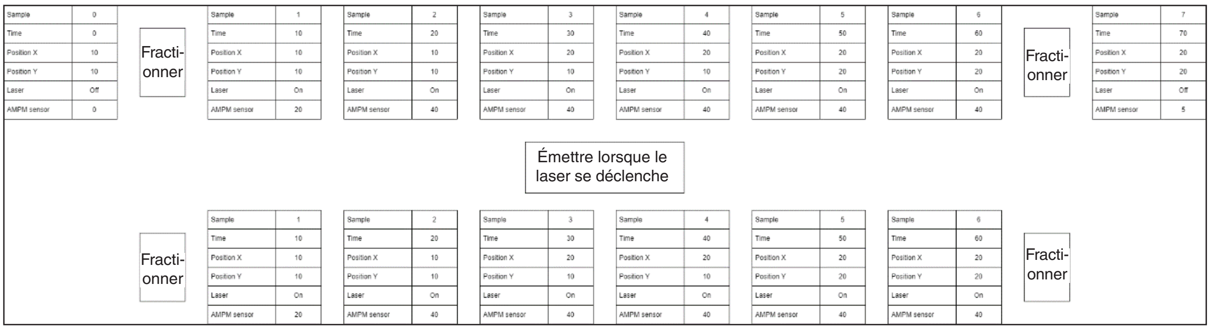
6.4.6.2.2 Filtre 1 – Fractionner en cas de changement de LaserModulate (Modulation laser)

Le premier filtre insère des jetons `Split` entre les changements au niveau du canal de modulation laser :



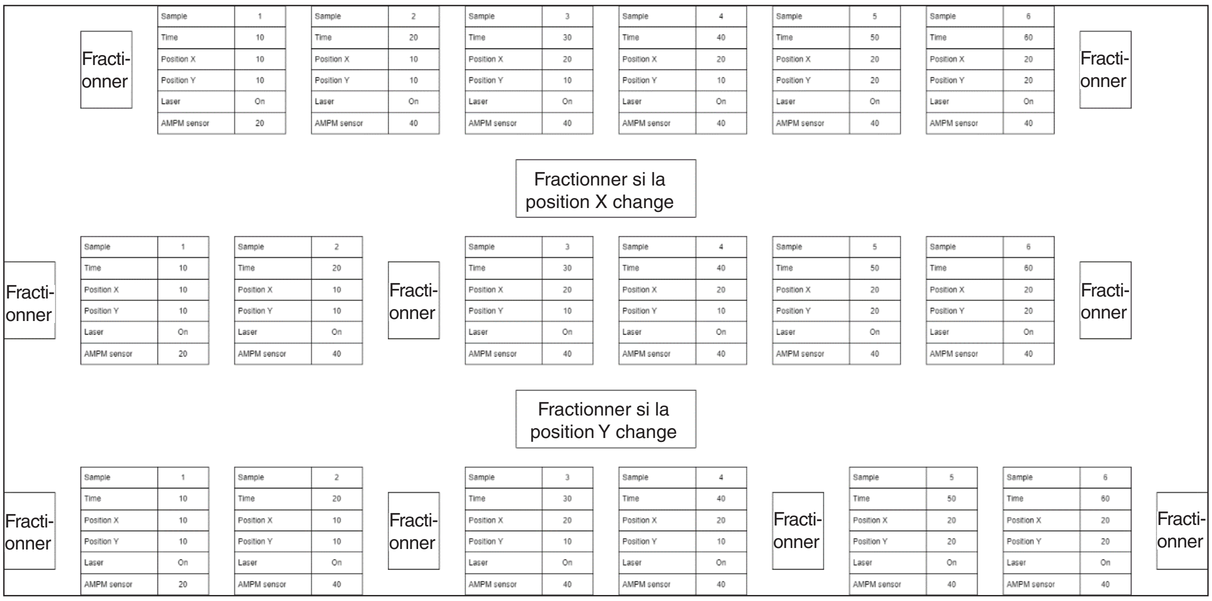
6.4.6.2.3 Filtre 2 – Émettre lorsque le laser se déclenche

Le filtre suivant élimine tous les échantillons où le laser ne se déclenche pas :



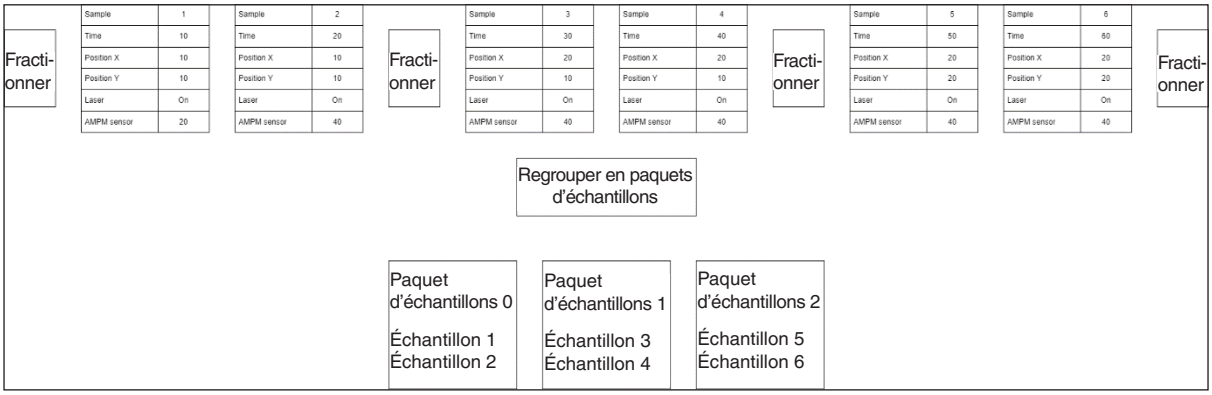
6.4.6.2.4 Filtres 3 et 4 – Fractionner en cas de changement de position

Les deux filtres suivants insèrent des jetons `Split` entre les échantillons pour lesquels la position du laser change :



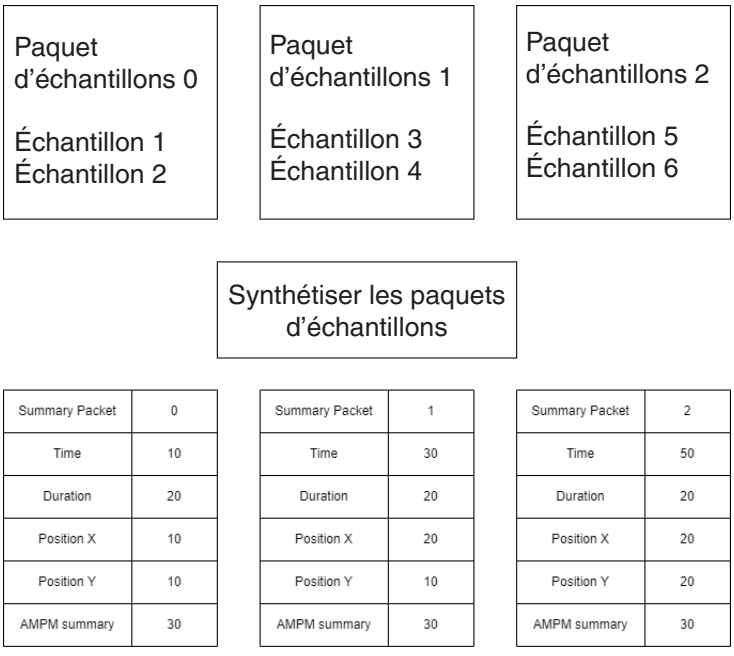
6.4.6.2.5 Filtre 5 – Regrouper en paquets d'échantillons

Les échantillons sont maintenant délimités par des jetons `Split` entre lesquels se trouvent les échantillons où le laser a été activé et était stationnaire (c'est-à-dire les échantillons d'un seul bain de fusion). Ces groupes d'échantillons sont ensuite réunis.



6.4.6.2.6 Filtre 6 – Paquet d'échantillons synthétisés

Enfin, les échantillons réunis sont synthétisés :



Chacun de ces jetons de paquet synthétisé final contient des valeurs décrivant l'heure de début (l'heure du premier échantillon participant), la durée (le nombre d'échantillons participants × la durée d'un échantillon), la position et un résumé des valeurs de surveillance de procédé.

6.4.7 Exemple 2 – Filtres

Cet exemple étend le plug-in minimum/maximum en utilisant les filtres fournis dans l'API. Les concepts clés suivants seront abordés :

- Filtres
- Traitement des jetons à travers un filtre

6.4.7.1 Filtres

Les filtres traitent et modifient un flux de jetons en ajoutant, supprimant, transférant ou modifiant les jetons. Un filtre traite un seul jeton et génère zéro ou plusieurs jetons. Cette correspondance de un à plusieurs représente un mécanisme puissant pour la modification d'un flux de jetons au fur et à mesure de son traitement.

Dans de nombreuses tâches d'analyse de surveillance de procédé, seuls les échantillons enregistrés lorsque le laser se déclenchait sont significatifs (car ils contiennent des données concernant le bain de fusion). Le filtrage des échantillons enregistrés alors que le laser ne se déclenchait pas réduit également le nombre de jetons à prendre en compte dans l'analyse. Le filtre intégré `EmitSampleIf.LaserModulate.IsOn` est conçu dans cette optique. Le plug-in peut faire passer chaque jeton `Sample` à travers ce filtre pour éliminer tous les échantillons sans déclenchement laser, et ainsi produire des valeurs minimales et maximales pertinentes pour la formation du bain de fusion.

En langage C#, un délégué est un type qui représente une méthode avec une signature particulière. La valeur d'un délégué est attribuée à l'aide d'une *instance de méthode* et cette méthode peut être invoquée via le délégué.

Un nouvel espace de noms `PlugIn.Tutorials.v2.Example2` et une classe de plug-in sont ajoutés avec un délégué `FilterOutLaserOffSamples` qui se voit attribuer le filtre `EmitSampleIf.LaserModulate.IsOn()` lors de la création de l'instance de plug-in :

```
using System;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Filters;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types.Tokens;
namespace PlugIn.Tutorials.v2.Example2
{
   public class PlugIn
   {
      public static string APIVersion() => "2";
      public static string DisplayName() => "Tutorial Example 2";
      public string Initialise(string initialisationData)
      {
         var initialisationValues = JObject.Parse(initialisationData);
         _outputFolder = initialisationValues["OutputFolder"].Value<string>();
         return InitialiseReturns.Success();
```

```
    }
    public string Process(IReadOnlyCollection<Token> tokens) => ProcessReturns.Success();
    public string Shutdown() => ShutdownReturns.Success();
    private string _outputFolder;
    private Func<Token, IEnumerable<Token>> FilterOutLaserOffSamples { get; }
        = EmitSampleIf.LaserModulate.IsOn();
  }
}
```

La méthode *Process* est ensuite mise en œuvre, en utilisant une expression LINQ pour faire passer chaque lot de jetons d'abord par le filtre (via le délégué), puis par une méthode qui traite le flux de jetons modifié qui en résulte :

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
  var messages = tokens.FilteredBy(FilterOutLaserOffSamples).
                        CollateMessagesFrom(ProcessTokens);
  return ProcessReturns.Success(messages);
}
private IEnumerable<string > ProcessTokens(IEnumerable<Token> tokens)
{
  foreach (var token in tokens)
    switch (token)
    {
      case Sample sample:
        _demandX             = MinMaxForProperty(_demandX,             sample.DemandX);
        _demandY             = MinMaxForProperty(_demandY,             sample.DemandY);
        _demandFocus         = MinMaxForProperty(_demandFocus,         sample.DemandFocus);
        _laserModulate       = MinMaxForProperty(_laserModulate,       sample.LaserModulate);
        _demandLaserPower    = MinMaxForProperty(_demandLaserPower,    sample.DemandLaserPower);
        _meltVIEWPlasma      = MinMaxForProperty(_meltVIEWPlasma,      sample.MeltVIEWPlasma);
        _meltVIEWMeltpool    = MinMaxForProperty(_meltVIEWMeltpool,    sample.MeltVIEWMeltpool);
        _laserVIEW           = MinMaxForProperty(_laserVIEW,           sample.LaserVIEW);
        _laserOutputPower    = MinMaxForProperty(_laserOutputPower,    sample.LaserOutputPower);
        _laserBackReflection = MinMaxForProperty(_laserBackReflection, sample.LaserBackReflection);
        break;
      case EndOfLayerLaserData endOfLayerLaserData:
        var layer = endOfLayerLaserData.Layer.Value;
        var laser = endOfLayerLaserData.Laser.Value;
        var outputFilePath = Path.Combine(_outputFolder, "Min & Max for "
```

```
                                                    + $"layer {layer}, "
                                                    + $"laser {laser}"
                                                    + ".txt");
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandX",             _demandX));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandY",             _demandY));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandFocus",         _demandFocus));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserModulate",       _laserModulate));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandLaserPower",    _demandLaserPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWPlasma",      _meltVIEWPlasma));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWMeltpool",    _meltVIEWMeltpool));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserVIEW",           _laserVIEW));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserOutputPower",    _laserOutputPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserBackReflection", _laserBackReflection));
            _demandX             = (double.MaxValue, double.MinValue);
            _demandY             = (double.MaxValue, double.MinValue);
            _demandFocus         = (double.MaxValue, double.MinValue);
            _laserModulate       = (int.MaxValue, int.MinValue);
            _demandLaserPower    = (int.MaxValue, int.MinValue);
            _meltVIEWPlasma      = (int.MaxValue, int.MinValue);
            _meltVIEWMeltpool    = (int.MaxValue, int.MinValue);
            _laserVIEW           = (int.MaxValue, int.MinValue);
            _laserOutputPower    = (int.MaxValue, int.MinValue);
            _laserBackReflection  = (int.MaxValue, int.MinValue);
            yield return "Processed layer {layer}, laser {laser}";
        }
}
```

REMARQUE : les différents champs(`_demandX`, etc.) et les méthodes utilitaires (`FormatMinAndMax`, etc.) sont omis par souci de concision et sont mis en œuvre comme dans le premier exemple.

L'expression LINQ inclut deux étapes. Tout d'abord, chaque lot de jetons passe par le filtre `FilterOutLaserOffTokens` via un utilitaire d'API `FilteredBy`. Ensuite, les jetons du flux de jetons résultant (les jetons `Sample` étant uniquement les échantillons enregistrés lors du déclenchement du laser) sont transmis à `ProcessToken` , qui effectue le traitement propre à chaque type de jeton significatif, c'est-à-dire `Sample` et `EndOfLayerLaserData`. Les chaînes résultantes renvoyées par `ProcessTokens` sont collectées et renvoyées à DataHUB par la méthode utilitaire `Success` .

REMARQUE : la syntaxe LINQ de C# est conçue pour traiter des flux de données.

REMARQUE : les flux de jetons peuvent être modifiés à travers un filtre.

REMARQUE : un filtre renvoie zéro ou plusieurs jetons pour chaque jeton d'entrée.

Le plug-in va maintenant renseigner les fichiers de résultats avec les valeurs minimales et maximales des canaux pour les échantillons enregistrés lorsque le laser se déclenchait.

6.4.7.2 Mise en œuvre finale

Le plug-in complet est le suivant :

```
using System;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Filters;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types.Tokens;
namespace PlugIn.Tutorials.v2.Example2
{
   public class PlugIn
   {
      public static string APIVersion() => “2”;
      public static string DisplayName() => “Tutorial Example 2”;
      public string Initialise(string initialisationData)
      {
         var initialisationValues = JObject.Parse(initialisationData);
         _outputFolder = initialisationValues[“OutputFolder”].Value<string>();
         return InitialiseReturns.Success();
      }
      public string Process(IReadOnlyCollection<Token> tokens)
      {
         var messages = tokens.FilteredBy(FilterOutLaserOffSamples).
                              CollateMessagesFrom(ProcessTokens);
         return ProcessReturns.Success(messages);
      }
      private IEnumerable<string> ProcessTokens(IEnumerable<Token> tokens)
      {
        foreach (var token in tokens)
         switch (token)
         {
            case Sample sample:
               _demandX             = MinMaxForProperty(_demandX,             sample.DemandX);
               _demandY             = MinMaxForProperty(_demandY,             sample.DemandY);
               _demandFocus         = MinMaxForProperty(_demandFocus,         sample.DemandFocus);
               _laserModulate       = MinMaxForProperty(_laserModulate,       sample.LaserModulate);
```

```
            _demandLaserPower    = MinMaxForProperty(_demandLaserPower,    sample.DemandLaserPower);
            _meltVIEWPlasma      = MinMaxForProperty(_meltVIEWPlasma,      sample.MeltVIEWPlasma);
            _meltVIEWMeltpool    = MinMaxForProperty(_meltVIEWMeltpool,    sample.MeltVIEWMeltpool);
            _laserVIEW           = MinMaxForProperty(_laserVIEW,           sample.LaserVIEW);
            _laserOutputPower    = MinMaxForProperty(_laserOutputPower,    sample.LaserOutputPower);
            _laserBackReflection = MinMaxForProperty(_laserBackReflection, sample.LaserBackReflection);
            break;
        case EndOfLayerLaserData endOfLayerLaserData:
            var layer = endOfLayerLaserData.Layer.Value;
            var laser = endOfLayerLaserData.Laser.Value;
            var outputFilePath = Path.Combine(_outputFolder, "Min & Max for "
                                                           + $"layer {layer}, "
                                                           + $"laser {laser}"
                                                           + ".txt");
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandX",          _demandX));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandY",          _demandY));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandFocus",      _demandFocus));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserModulate",    _laserModulate));
            File.AppendAllText(outputFilePath, FormatMinAndMax("DemandLaserPower", _demandLaserPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWPlasma",   _meltVIEWPlasma));
            File.AppendAllText(outputFilePath, FormatMinAndMax("MeltVIEWMeltpool", _meltVIEWMeltpool));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserVIEW",        _laserVIEW));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserOutputPower", _laserOutputPower));
            File.AppendAllText(outputFilePath, FormatMinAndMax("LaserBackReflection",
                                                               _laserBackReflection));
            _demandX             = (double.MaxValue, double.MinValue);
            _demandY             = (double.MaxValue, double.MinValue);
            _demandFocus         = (double.MaxValue, double.MinValue);
            _laserModulate       = (int.MaxValue, int.MinValue);
            _demandLaserPower    = (int.MaxValue, int.MinValue);
            _meltVIEWPlasma      = (int.MaxValue, int.MinValue);
            _meltVIEWMeltpool    = (int.MaxValue, int.MinValue);
            _laserVIEW           = (int.MaxValue, int.MinValue);
            _laserOutputPower    = (int.MaxValue, int.MinValue);
            _laserBackReflection = (int.MaxValue, int.MinValue);
            yield return $"Processed layer {layer}, laser {laser}";
            break;
    }
}
public string Shutdown() => ShutdownReturns.Success();
```

```
    private string _outputFolder;
    private Func<Token, IEnumerable<Token>> FilterOutLaserOffSamples { get; }
       = EmitSampleIf.LaserModulate.IsOn();
    private (double Min, double Max) _demandX             = (double.MaxValue, double.MinValue);
    private (double Min, double Max) _demandY             = (double.MaxValue, double.MinValue);
    private (double Min, double Max) _demandFocus         = (double.MaxValue, double.MinValue);
    private (int Min, int Max)       _laserModulate       = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _demandLaserPower    = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _meltVIEWPlasma      = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _meltVIEWMeltpool    = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _laserVIEW           = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _laserOutputPower    = (int.MaxValue, int.MinValue);
    private (int Min, int Max)       _laserBackReflection = (int.MaxValue, int.MinValue);
    private static string FormatMinAndMax(string propertyName, (double Min, double Max) propertyMinAndMax)
    {
       return $"{propertyName}\t"
            + $"{propertyMinAndMax.Min:F3}\t"
            + $"{propertyMinAndMax.Max:F3}\t"
            + $"{Environment.NewLine}";
    }
    private static string FormatMinAndMax(string propertyName, (int Min, int Max) propertyMinAndMax)
    {
       return $"{propertyName}\t"
            + $"{propertyMinAndMax.Min}\t"
            + $"{propertyMinAndMax.Max}\t"
            + $"{Environment.NewLine}";
    }
    private static (double, double) MinMaxForProperty((double Min, double Max) currentValue, double value)
    {
       var (min, max) = currentValue;
       return (Math.Min(min, value), Math.Max(max, value));
    }
    private static (int, int) MinMaxForProperty((int Min, int Max) currentValue, int value)
    {
       var (min, max) = currentValue;
       return (Math.Min(min, value), Math.Max(max, value));
    }
  }
}
```

6.4.8 Exemple 3 – ExportPackets

Bien que les filtres individuels soient utiles, vous profiterez de tout leur potentiel en les combinant en pipelines (voir la section « Pipelines » à la page 6-56). Un pipeline effectue une opération complexe sur un flux de jetons au moyen d'une série de filtres simples et réutilisables. C'est exactement ce que fait le plug-in `ExportPackets` fourni avec le service DataHUB et cet exemple recréera cette mise en œuvre en réalisant les actions suivantes :

- Combinaison de filtres pour former un pipeline.
- Traitement d'un flux de jetons à travers un pipeline.

6.4.8.1 Combinaison de filtres

L'API de plug-in fournit deux utilitaires `First` et `Then` pour faciliter la combinaison de filtres en pipelines. Un pipeline se crée comme suit :

```
First(<filter>).Then(<filter>).Then(<filter>)…Then(<filter>);
```

Il est possible d'enchaîner un nombre quelconque de filtres, et le pipeline qui en résulte se comporte comme un filtre, en faisant correspondre un seul jeton d'entrée à zéro ou plusieurs jetons de sortie.

Le plug-in *ExportPackets* fourni avec DataHUB traite un flux de jetons en paquets, où un paquet est un résumé des données de surveillance de procédé collectées pour des échantillons consécutifs alors qu'un laser se déclenchait à une position X et Y particulière de la demande. Pour créer des paquets à partir d'échantillons, un flux de jetons est traité en suivant ces étapes :

1. Fractionnement du flux à chaque fois que l'état d'activation ou de désactivation du laser change.
2. Retrait de tous les échantillons pour lesquels le laser ne s'est pas déclenché.
3. Fractionnement du flux à chaque fois que la position X change.
4. Fractionnement du flux à chaque fois que la position Y change.
5. Regroupement de chaque ensemble d'échantillons délimité par des fractionnements en un paquet d'échantillons.
6. Production de valeurs récapitulatives moyennes et médianes pour chaque paquet d'échantillons.

REMARQUE : les échantillons du flux pour lesquels le laser ne s'est pas déclenché ne sont pas filtrés en premier, comme cela pourrait sembler judicieux à première vue, car cela pourrait entraîner la création de paquets erronés dans le cas où le laser reste à la même demande de position X et Y tout en s'allumant et s'éteignant de manière répétée.

Le code de départ pour le plug-in est :

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
```

```
using System.Linq;
using System.Text;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Filters;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types.Tokens;
using static PlugIns.API.v2.Helpers.PositionHelpers;
using static PlugIns.API.v2.Helpers.Pipeline;
namespace PlugIn.Tutorials.v2.Example3
{
   public class PlugIn
   {
      public static string APIVersion() => “2”;
      public static string DisplayName() => “Tutorial 3 - Export Packets”;
      public string Initialise(string initialisationData)
      {
          _outputFolder = JObject.Parse(initialisationData)[“OutputFolder”].Value<string>();
          SetColumnHeadings(_builder);
          return InitialiseReturns.Success();
      }
      public string Process(IReadOnlyCollection<Token> tokens) => ProcessReturns.Success();
      public string Shutdown() => ShutdownReturns.Success();
      private static void SetColumnHeadings(StringBuilder builder)
      {
         builder.Clear();
         builder.AppendLine(“Start time\tDuration\t”
                            + “Demand X\t”
                            + “Demand Y\t”
                            + “Demand focus\t”
                            + “Demand laser power (mean)\t”
                            + “MeltVIEW plasma (mean)\t”
                            + “MeltVIEW melt pool (mean)\t”
                            + “LaserVIEW (mean)\t”
                            + “Laser back reflection (mean)\t”
                            + “Laser output power (mean)\t”
                            + “Demand laser power (median)\t”
                            + “MeltVIEW plasma (median)\t”
                            + “MeltVIEW melt pool (median)LaserVIEW (median)\t”
                            + “Laser back reflection (median)\t”
                            + “Laser output power (median)\t”);
```

```
      }
      private readonly StringBuilder _builder = new StringBuilder();
      private string _outputFolder;
   }
}
```

Le plug-in utilise un `StringBuilder` pour cumuler les résumés de tous les paquets d'échantillons ; il ajoute d'abord les en-tête de chaque colonne dans le fichier csv.

Le pipeline mentionné ci-dessus est créé et affecté au délégué `CollateIntoPackets` :

```
…
private Func<Token, IEnumerable<Token>> CollateIntoPackets { get; }
   = First(SplitWhen.LaserModulateChanges()).
     Then(EmitSampleIf.LaserModulate.IsOn()).
     Then(SplitWhen.DemandXChanges()).
     Then(SplitWhen.DemandYChanges()).
     Then(CollateInto.PacketOfSamples()).
     Then(SummarisePacketOfSamples.Summarise(new SummarisePacketOfSamples.SummaryMethod(
                                               "Mean",
                                               values => values.Average(x => x)),
                                            new SummarisePacketOfSamples.SummaryMethod(
                                               "Median",
                                               Median)));
private static double Median(IReadOnlyList<double> values)
{
   var halfwayIndex = values.Count >> 1;
   return values.Count % 2 == 1
            ? values[halfwayIndex]
            : (values[halfwayIndex] + values[halfwayIndex - 1]) * 0.5;
}
…
```

Le pipeline se compose de plusieurs filtres intégrés enchaînés de sorte que la sortie finale du pipeline sera une série de jetons `SummaryPacket` contenant les valeurs moyennes et médianes. Voir la section « Regroupement des échantillons en paquets à l'aide d'un pipeline de filtres » à la page 6-57 pour une vue d'ensemble de la fonction du pipeline.

REMARQUE : des filtres simples enchaînés les uns aux autres forment un pipeline capable d'effectuer une transformation complexe d'un flux de jetons d'entrée.

Le filtre final, `SummarisePacketOfSamples.Summarise`, est initialisé avec des `SummaryMethods` qui nomment les résumés (« Mean » et « Median ») et fournissent également la méthode par laquelle leur valeur doit être calculée. Il s'agit d'un moyen flexible de configurer la manière dont les paquets d'échantillons seront résumés : de nombreuses méthodes de résumé peuvent être fournies, par exemple pour produire des moyennes, des médianes, des minimums et des maximums :

```
SummarisePacketOfSamples.Summarise(
   new SummarisePacketOfSamples.SummaryMethod("Mean",    values => values.Average(x => x)),
   new SummarisePacketOfSamples.SummaryMethod("Median",  Median)),
   new SummarisePacketOfSamples.SummaryMethod("Minimum", values => values.Min(x => x)),
   new SummarisePacketOfSamples.SummaryMethod("Maximum", values => values.Max(x => x)))
```

La dernière étape consiste à mettre en œuvre la méthode *Process* qui ajoutera (grâce à la méthode utilitaire `AddSummaryLine`) au `StringBuilder` une ligne pour chaque jeton `SummaryPacket` et écrira le contenu du `StringBuilder` dans le fichier de sortie :

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   var messages = tokens.FilteredBy(CollateIntoPackets).
                         CollateMessagesFrom(ProcessToken);
   return ProcessReturns.Success(messages);
}
private IEnumerable<string> ProcessToken(IEnumerable<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         case SummaryPacket summaryPacket:
            AddSummaryLine(_builder,
                           summaryPacket.Time,
                           summaryPacket.Duration,
                           summaryPacket.Summary["Mean"],
                           summaryPacket.Summary["Median"]);
         break;
         case EndOfLayerLaserData endOfLayerLaserData:
            var layer = endOfLayerLaserData.Layer.Value;
            var laser = endOfLayerLaserData.Laser.Value;
            File.AppendAllText(Path.Combine(_outputFolder,
                               $"Packet data for layer {layer}, laser {laser}.txt"), _builder.ToString());
            SetColumnHeadings(_builder);
            yield return $"Processed layer {layer}, laser {laser}";
```

```
            break;
        }
}
private void AddSummaryLine(StringBuilder builder,
                            uint time,
                            uint duration,
                            SummaryPacket.SummaryValues means,
                            SummaryPacket.SummaryValues medians)
{
   builder.AppendLine($"{time}\t"
                    + $"{duration}\t"
                    + $"{FormatTo3dp(RoundToMicrons(means.DemandX,     1))}\t"
                    + $"{FormatTo3dp(RoundToMicrons(means.DemandY,     1))}\t"
                    + $"{FormatTo3dp(RoundToMicrons(means.DemandFocus, 1))}\t"
                    + $"{FormatTo3dp(means.DemandLaserPower)}\t"
                    + $"{FormatTo3dp(means.MeltVIEWPlasma)}\t"
                    + $"{FormatTo3dp(means.MeltVIEWMeltpool)}\t"
                    + $"{FormatTo3dp(means.LaserVIEW)}\t"
                    + $"{FormatTo3dp(means.LaserBackReflection)}\t"
                    + $"{FormatTo3dp(means.LaserOutputPower)}\t"
                    + $"{FormatTo3dp(medians.DemandLaserPower)}\t"
                    + $"{FormatTo3dp(medians.MeltVIEWPlasma)}\t"
                    + $"{FormatTo3dp(medians.MeltVIEWMeltpool)}\t"
                    + $"{FormatTo3dp(medians.LaserVIEW)}\t"
                    + $"{FormatTo3dp(medians.LaserBackReflection)}\t"
                    + $"{FormatTo3dp(medians.LaserOutputPower)}\t");
}
private string FormatTo3dp(double meansDemandLaserPower)
{
   return meansDemandLaserPower.ToString("F3", _currentCulture);
}
…
```

Chaque jeton `SummaryPacket` est ajouté sous la forme d'une ligne de texte, avec le formatage approprié à chaque valeur, à l'aide du `StringBuilder`. En présence d'un jeton `EndOfLayerLaserData`, le plug-in écrit les données des paquets regroupées dans un fichier du dossier de sortie.

REMARQUE : le flux de jetons de fabrication peut être traité à travers un pipeline prédéfini.

6.4.8.2 Mise en œuvre finale

Le plug-in complet est le suivant :

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.IO;
using System.Linq;
using System.Text;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Filters;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types.Tokens;
using static PlugIns.API.v2.Helpers.PositionHelpers;
using static PlugIns.API.v2.Helpers.Pipeline;
namespace PlugIn.Tutorials.v2.Example3
{
   public class PlugIn
   {
      public static string APIVersion() => “2”;
      public static string DisplayName() => “Tutorial 3 - Export Packets”;
      public string Initialise(string initialisationData)
      {
         _outputFolder = JObject.Parse(initialisationData)[“OutputFolder”].Value<string>();
         SetColumnHeadings(_builder);
         return InitialiseReturns.Success();
      }
      public string Process(IReadOnlyCollection<Token> tokens)
      {
         var messages = tokens.FilteredBy(CollateIntoPackets).
                              CollateMessagesFrom(ProcessToken);
         return ProcessReturns.Success(messages);
      }
      public string Shutdown() => ShutdownReturns.Success();
      private IEnumerable<string> ProcessToken(IEnumerable<Token> tokens)
      {
         foreach (var token in tokens)
            switch (token)
            {
               case SummaryPacket summaryPacket:
```

```
                AddSummaryLine(_builder,
                               summaryPacket.Time,
                               summaryPacket.Duration,
                               summaryPacket.Summary[“Mean”],
                               summaryPacket.Summary[“Median”]);
                break;
            case EndOfLayerLaserData endOfLayerLaserData:
                var layer = endOfLayerLaserData.Layer.Value;
                var laser = endOfLayerLaserData.Laser.Value;
                File.AppendAllText(Path.Combine(_outputFolder,
                                                $”Packet data for layer {layer}, laser {laser}.txt”),
                                   _builder.ToString());
                SetColumnHeadings(_builder);
                yield return $”Processed layer {layer}, laser {laser}”;
                break;
        }
    }
    private static void SetColumnHeadings(StringBuilder builder)
    {
        builder.Clear();
        builder.AppendLine(“Start time\tDuration\t”
                           + “Demand X\t”
                           + “Demand Y\t”
                           + “Demand focus\t”
                           + “Demand laser power (mean)\t”
                           + “MeltVIEW plasma (mean)\t”
                           + “MeltVIEW melt pool (mean)\t”
                           + “LaserVIEW (mean)\t”
                           + “Laser back reflection (mean)\t”
                           + “Laser output power (mean)\t”
                           + “Demand laser power (median)\t”
                           + “MeltVIEW plasma (median)\t”
                           + “MeltVIEW melt pool (median)LaserVIEW (median)\t”
                           + “Laser back reflection (median)\t”
                           + “Laser output power (median)\t”);
    }
    private void AddSummaryLine(
        StringBuilder builder,
        uint time,
        uint duration,
```

```
    SummaryPacket.SummaryValues means,
    SummaryPacket.SummaryValues medians)
{
    builder.AppendLine($"{time}\t"
                     + $"{duration}\t"
                     + $"{FormatTo3dp(RoundToMicrons(means.DemandX,     1))}\t"
                     + $"{FormatTo3dp(RoundToMicrons(means.DemandY,     1))}\t"
                     + $"{FormatTo3dp(RoundToMicrons(means.DemandFocus, 1))}\t"
                     + $"{FormatTo3dp(means.DemandLaserPower)}\t"
                     + $"{FormatTo3dp(means.MeltVIEWPlasma)}\t"
                     + $"{FormatTo3dp(means.MeltVIEWMeltpool)}\t"
                     + $"{FormatTo3dp(means.LaserVIEW)}\t"
                     + $"{FormatTo3dp(means.LaserBackReflection)}\t"
                     + $"{FormatTo3dp(means.LaserOutputPower)}\t"
                     + $"{FormatTo3dp(medians.DemandLaserPower)}\t"
                     + $"{FormatTo3dp(medians.MeltVIEWPlasma)}\t"
                     + $"{FormatTo3dp(medians.MeltVIEWMeltpool)}\t"
                     + $"{FormatTo3dp(medians.LaserVIEW)}\t"
                     + $"{FormatTo3dp(medians.LaserBackReflection)}\t"
                     + $"{FormatTo3dp(medians.LaserOutputPower)}\t");
}
private Func<Token, IEnumerable<Token>> CollateIntoPackets { get; }
    = First(SplitWhen.LaserModulateChanges()).
      Then(EmitSampleIf.LaserModulate.IsOn()).
      Then(SplitWhen.DemandXChanges()).
      Then(SplitWhen.DemandYChanges()).
      Then(CollateInto.PacketOfSamples()).
      Then(SummarisePacketOfSamples.Summarise(new SummarisePacketOfSamples.SummaryMethod(
                                                "Mean",
                                                values => values.Average(x => x)),
                                              new SummarisePacketOfSamples.SummaryMethod(
                                                "Median",
                                                Median)));
private static double Median(IReadOnlyList<double> values)
{
    var halfwayValue = values.Count >> 1;
    return values.Count % 2 == 1
        ? values[halfwayValue]
        : (values[halfwayValue] + values[halfwayValue - 1]) * 0.5;
}
```

```
      private string FormatTo3dp(double meansDemandLaserPower)
      {
         return meansDemandLaserPower.ToString("F3", _currentCulture);
      }
      private readonly IFormatProvider _currentCulture = CultureInfo.CurrentCulture;
      private readonly StringBuilder _builder = new StringBuilder();
      private string _outputFolder;
   }
}
```

6.4.9 Example 4 – Renvoi de données de séries temporelles à Renishaw Central

L'exemple suivant montre comment un plug-in crée et remplit une série temporelle Renishaw Central avec des points de données. L'interface utilisateur Web de Renishaw Central affiche les données de la série temporelle sous forme de graphique.

L'exemple de plug-in définit une série temporelle et, pour chaque couche, ajoute un point de données pour la valeur maximale du canal LaserVIEW. La définition de la série temporelle et ses points de données sont renvoyés à DataHUB via les valeurs de retour des méthodes *Initialise* et *Process*.

Concepts clés abordés :

- Création d'une série temporelle Renishaw Central
- Ajout de données à une série temporelle Renishaw Central

En commençant par le plug-in standard vierge :

```
using System;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Types.Tokens;
using PlugIns.API.v2.Types;
namespace PlugIn.Tutorials.v2.Example4
{
   public class PlugIn
   {
      public static string APIVersion() => "2";
      public static string DisplayName() => "Tutorial Example 4";
      public string Initialise(string initialisationData) => InitialiseReturns.Success();
      public string Process(IReadOnlyCollection<Token> tokens) => ProcessReturns.Success();
      public string Shutdown() => ShutdownReturns.Success();
   }
}
```

6.4.9.1 Initialisation

Lorsque DataHUB initialise ce plug-in, la tâche doit renvoyer une définition pour la série temporelle « LaserVIEW Max ». La définition d'une série temporelle nécessite un nom, un type d'unité et une unité. Des propriétés supplémentaires peuvent également être définies, telles que la propriété Grouping qui catégorise les séries temporelles en un groupement utilisé par le front end Web de Renishaw Central pour regrouper et gérer l'affichage des séries temporelles apparentées.

La définition de la série temporelle est renvoyée à DataHUB à via la méthode utilitaire `InitialiseReturns SuccessAndCreateTimeSeries()`. DataHUB convertit cette valeur renvoyée en publications sur Renishaw Central contenant les données nécessaires pour créer la série temporelle prête à être remplie avec des points de données :

```
…
public string Initialise(string initialisationData)
{
   _laserViewMax = TimeSeries.Factory().
                              Name(“LaserVIEW Max”)UnitType(“Intensity”).
                              DimensionlessUnit().
                              Grouping(“LaserVIEW”).
                              Create();
   return InitialiseReturns.SuccessAndCreateTimeSeries(_laserViewMax);
}
private TimeSeries _laserViewMax;
…
```

REMARQUE : une ou plusieurs définitions de séries temporelles peuvent être renvoyées par la méthode *Initialise* du plug-in.

REMARQUE : DataHUB gère les détails de la publication des données sur Renishaw Central, ce qui permet aux plug-ins de se concentrer simplement sur les définitions des séries temporelles.

6.4.9.2 Traitement

Lors du traitement de chaque jeton `Sample` dans le flux de jetons, la valeur maximale actuelle est comparée à la valeur du canal LaserVIEW de l'échantillon et est mise à jour si nécessaire :

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
   {
      switch (token)
      {
         case Sample sample:
            if (sample.LaserVIEW >= _max.value)
               _max = (sample.Time, sample.LaserVIEW);
         break;
      }
   }
   return ProcessReturns.SuccessWithoutInformation();
}
private (uint time, int value) _max = (0, int.MinValue);
…
```

À la fin des données laser pour la couche donnée, lorsque le plug-in reçoit le jeton `EndOfLayerLaserData` , un message de réussite peut être renvoyé à DataHUB pour signifier que les données pour la couche et le laser en question ont été traitées :

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         …
         case EndOfLayerLaserData endOfLayerLaserData:
            return ProcessReturns.Success($"Processed layer {endOfLayerLaserData.Layer.Value}, " +
                                          $"laser {endOfLayerLaserData.Laser.Value}");
         …
      }
   return ProcessReturns.SuccessWithoutInformation();
}
…
```

À la fin des données de la couche, le plug-in recevra un jeton `EndOfLayerData`. L'objet `TimeSeries` est utilisé pour produire un objet `UpdateTimeSeries` avec une valeur de série temporelle contenant la valeur maximale finale de la couche. Les utilitaires `ProcessReturns` fournissent une méthode `SuccessAndSendData()` qui prend une matrice d'objets `UpdateTimeSeries`.

La valeur maximale mise en cache pour la couche est réinitialisée, prête pour le prochain lot de jetons, et la mise à jour de la série temporelle est renvoyée :

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   …
   case EndOfLayerData endOfLayerData:
      var laserViewMaxUpdate = _laserViewMax.Update().
                                             WithValues((_max.Time, _max.Value)).
                                             NoLimits();
      _max = (0, int.MinValue);
      return ProcessReturns.SuccessAndSendData($"Processed layer {endOfLayerData.Layer.Value}",
                                               new[] {laserViewMaxUpdate});
   …
}
```

6.4.9.3 Mise en œuvre finale

Le plug-in complet est le suivant :

```
using System.Collections.Generic;
using PlugIns.API.v2.Helpers;
using PlugIns.API.v2.Types;
using PlugIns.API.v2.Types.Tokens;
namespace PlugIn.Tutorials.v2.Example4
{
   public class PlugIn
   {
      public static string APIVersion() => "2";
      public static string DisplayName() => "Tutorial Example 4";
      public string Initialise(string initialisationData)
      {
         _laserViewMax = TimeSeries.Factory().
                                    Name("LaserVIEW Max").
                                    UnitType("Intensity").
                                    DimensionlessUnit().
                                    Grouping("LaserVIEW").
```

```
                                    Create();
        return InitialiseReturns.SuccessAndCreateTimeSeries(_laserViewMax);
    }
    public string Process(IReadOnlyCollection<Token> tokens)
    {
        foreach (var token in tokens)
            switch (token)
            {
                case Sample sample:
                    if (sample.LaserVIEW >= _max.Value)
                        _max = (sample.Time, sample.LaserVIEW);
                    break;
                case EndOfLayerLaserData endOfLayerLaserData:
                    return ProcessReturns.Success($"Processed layer {endOfLayerLaserData.Layer.Value}, "
                        + $"laser {endOfLayerLaserData.Laser.Value}");
                case EndOfLayerData endOfLayerData:
                    var laserViewMaxUpdate = _laserViewMax.Update().
                                                           WithValues((_max.Time, _max.Value)).
                                                           NoLimits();
                    _max = (0, int.MinValue);
                    return ProcessReturns.SuccessAndSendData($"Processed layer {endOfLayerData.Layer.Value}",
                                                             new[] {laserViewMaxUpdate});
            }
        return ProcessReturns.SuccessWithoutInformation();
    }
    public string Shutdown() => ShutdownReturns.Success();
    private (uint Time, int Value) _max = (0, int.MinValue);
    private TimeSeries             _laserViewMax;
  }
}
```

6.4.10 Exemple 5 – Lancement d’alertes sur Renishaw Central

Cet exemple montre comment déclencher des alertes sur Renishaw Central lorsque certaines valeurs dépassent des seuils prédéfinis.

Concepts clés abordés :

- Décodage des données d’initialisation
- Lancement d’alertes

6.4.10.1 Configuration de la tâche de plug-in avec des données d’initialisation

Les seuils au-delà desquels cet exemple déclenche des alertes sont contrôlés par les données d’initialisation du plug-in. Dans cet exemple, les données d’initialisation correspondant à la structure suivante sont saisies à l’étape « Specify Plug-in Initialisation Data » (Préciser les données d’initialisation du plug-in) de DataHUB Generator.

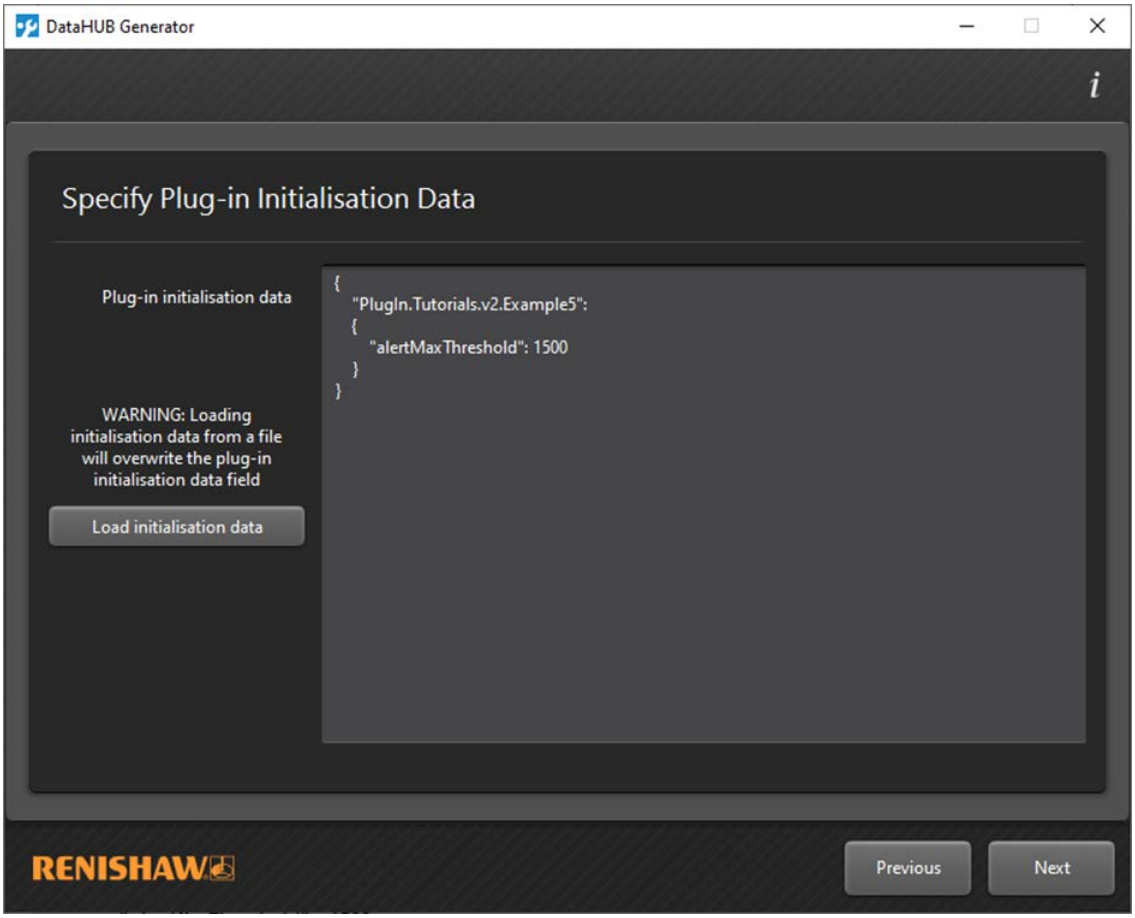


Figure 41 Saisie des données d’initialisation du plug-in dans DataHUB Generator

```
{
  “PlugIn.Tutorials.v2.Example5”:
  {
    “alertMaxThreshold”: 1500
  }
}
```

En commençant par le plug-in standard vierge :

```
using System;
using System.Collections.Generic;
using System.IO;
using Newtonsoft.Json.Linq;
using PlugIns.API.v2.Types.Tokens;
namespace PlugIn.Tutorials.v2.Example5
{
   public class PlugIn
   {
      public static string APIVersion() => “2”;
      public static string DisplayName() => “Tutorial Example 5”;
      public string Initialise(string initialisationData) => InitialiseReturns.Success();
      public string Process(IReadOnlyCollection<Token> tokens) => ProcessReturns.Success();
      public string Shutdown() => ShutdownReturns.Success();
   }
}
```

6.4.10.2 Initialisation

Lors de l'initialisation, les données d'initialisation spécifiques au plug-in sont accessibles à partir de l'objet JSON « initialisationData » dont le nom a été fourni lors de la configuration de la tâche. Il s'agit de « PlugIn.Tutorials.v2.Example5 » dans cet exemple :

```
…
public string Initialise(string initialisationData)
{
   var iv               = JObject.Parse(initialisationData);
   var id               = JObject.Parse(iv["InitialisationData"].Value<string>());
   var example5InitData = id["PlugIn.Tutorials.v2.Example5"];
   _alertMaxThreshold = example5InitData["alertMaxThreshold"].Value<int>();
   InitialiseReturns.Success();
}
private int _alertMaxThreshold;
…
```

REMARQUE : la chaîne d'initialisation du plug-in spécifiée lors de la configuration de la tâche de traitement dans DataHUB Generator est disponible pour le plug-in lors de l'initialisation.

6.4.10.3 Traitement

Ce plug-in consiste à déclencher une alerte dans Renishaw Central si la valeur maximale de LaserVIEW pour la couche dépasse le seuil défini. Tout d'abord, la valeur maximale actuelle de la couche et l'heure de l'échantillon associée sont mises en cache et initialisées. Lors du traitement des jetons Sample, le canal « LaserVIEW » est comparé à la valeur maximale actuelle, et cette valeur est écrasée si nécessaire. L'heure de l'échantillon est également enregistrée avec la valeur maximale :

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         case Sample sample:
            if (sample.LaserVIEW >= _max.value)
               _max = (sample.Time, sample.LaserVIEW);
            break;
      }
   return SuccessWithoutInformation();
}
private (uint time, int value) _max = (0, int.MinValue);
…
```

À la fin des données laser pour la couche donnée, le plug-in recevra le jeton `EndOfLayerLaserData`. Un message de réussite peut être renvoyé à DataHUB pour indiquer que les données pour la couche/le laser donné(e) ont été traitées avec succès :

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         …
         case EndOfLayerLaserData endOfLayerLaserData:
            return Success($"Processed layer {endOfLayerLaserData.Layer}, " +
                           $"laser {endOfLayerLaserData.Laser}");
         …
      }
   return SuccessWithoutInformation();
}
…
```

À la fin de tous les échantillons de la couche, le plug-in reçoit un jeton `EndOfLayerData` , signifiant qu'il n'y aura plus de données pour la couche donnée. En présence de ce jeton, le traitement peut être finalisé et les résultats envoyés à Renishaw Central.

Si le maximum actuel a dépassé le seuil défini, une alerte est créée et ajoutée à la liste des alertes. L'alerte est créée avec une description, un niveau de gravité, un nom et un horodatage.

Les utilitaires `ProcessReturns` fournissent une méthode `SuccessAndSendData` qui prend un message et un tableau d'alertes et encode ces données pour les renvoyer à DataHUB :

```
…
public string Process(IReadOnlyCollection<Token> tokens)
{
   foreach (var token in tokens)
      switch (token)
      {
         …
         case EndOfLayerData endOfLayerData:
            var alerts = new List<Alert>();
            if (_max.value > _alertMaxThreshold)
               alerts.Add(Alert.Factory().
                                 Description(“LaserVIEW intensity has exceeded the maximum threshold”).
                                 Warning().
                                 Name(“LaserVIEW Max Threshold”).
                                 Time(_max.time));

            _max = (0, int.MinValue);
            return ProcessReturns.
                        SuccessAndSendData($”Processed layer {endOfLayerData.Layer.Value}”,
                                           alerts);
        …
      }
…
```

REMARQUE : les descriptions d'alertes renvoyées à DataHUB sont transmises à Renishaw Central qui peut déclencher des actions telles que l'envoi d'un SMS.

6.4.10.4 Mise en œuvre finale

Le plug-in complet est le suivant :

```
public class PlugIn
{
   public static string APIVersion() => “2”;
   public static string DisplayName() => “Tutorial Example 5”;
   public string Initialise(initialisationData)
   {
      var iv               = JObject.Parse(initialisationData);
      var id               = JObject.Parse(iv[“InitialisationData”].Value<string>());
      var example5InitData = id[“PlugIn.Tutorials.v2.Example5”].ToObject<InitialisationData>();
      _alertMaxThreshold = example5InitData.AlertMaxThreshold;
      return InitialiseReturns.Success();
   }
   public string Process(IReadOnlyCollection<Token> tokens)
   {
      foreach (var token in tokens)
         switch (token)
         {
            case Sample sample:
               if (sample.LaserVIEW >= _max.value)
                  _max = (sample.Time, sample.LaserVIEW);
               break;
            case EndOfLayerLaserData endOfLayerLaserData:
               return ProcessReturns.Success($”Processed layer {endOfLayerLaserData.Layer.Value}, “
                                             + $”laser {endOfLayerLaserData.Laser.Value}”);
            case EndOfLayerData endOfLayerData:
               var alerts = new List<Alert>();
               if (_max.value > _alertMaxThreshold)
                  alerts.Add(Alert.Factory().
                                    Description(“LaserVIEW intensity has exceeded the maximum threshold”).
                                    Warning().
                                    Name(“LaserVIEW Max Threshold”).
                                    Time(_max.time));
               _max = (0, int.MinValue);
               return ProcessReturns.
                       SuccessAndSendData($”Processed layer {endOfLayerData.Layer.Value}”, alerts);
         }
      return ProcessReturns.SuccessWithoutInformation();
```

```
    }
    public string Shutdown() => ShutdownReturns.Success();
    private (uint time, int value) _max = (0, int.MinValue);
    private int _alertMaxThreshold;
}
```

6.4.11 API de plug-in V2.0

Pour qu'un assembly puisse être utilisé par le service DataHUB en tant que plug-in V2.0, l'API publique suivante doit être mise en œuvre.

6.4.11.1 Dénomination de l'assembly

Le nom de fichier de l'assembly .NET doit commencer par « PlugIn. » (« PlugIn » suivi d'un point) et doit avoir l'extension « dll ».

6.4.11.2 Dénomination de la classe de plug-in

L'assembly de plug-in doit définir une classe publique nommée « PlugIn ».

6.4.11.3 Méthodes de la classe de plug-in

La classe de plug-in doit mettre en œuvre les méthodes publiques suivantes :

```
public PlugIn()
public static string APIVersion()
public static string DisplayName()
public string Initialise(string data)
public string Process(IReadOnlyCollection<Token> data)
public string Shutdown()
```

6.4.11.4 Constructeur sans paramètre

Le type doit avoir un constructeur sans paramètre. Si aucun constructeur avec paramètres n'est défini, C# le fournit automatiquement, c'est-à-dire qu'il n'est pas nécessaire de le définir explicitement.

REMARQUE : un plug-in ne doit pas acquérir de ressources au sein de son constructeur.

Lorsque vous construisez une instance de plug-in, il n'est pas garanti que la méthode *Shutdown* soit appelée, donc ces ressources pourraient ne pas être nettoyées en temps voulu *Initialise* est l'endroit approprié pour l'acquisition des ressources.

6.4.11.5 La méthode `APIVersion` statique

La méthode `APIVersion` statique est utilisée par DataHUB pour identifier la version de l'API par rapport à laquelle ce plug-in doit être validé, ainsi que le format des données de surveillance de procédé qu'il traitera.

Paramètres :

Aucun.

Retour : `string`

Un plug-in qui met en œuvre l'API V2.0 doit renvoyer exactement « 2 ».

6.4.11.6 La méthode `DisplayName` statique

Le contenu de la chaîne renvoyée par la méthode statique `DisplayName` est utilisé comme nom de plug-in affiché dans DataHUB Monitor, lors de la création du dossier de sortie du plug-in, lors de l'audit des événements, etc. Bien qu'il ne soit pas obligatoire que ce nom soit unique, cela permet d'identifier le plug-in lorsque, par exemple, plusieurs versions sont installées sur un PCCD.

Paramètres :

Aucun.

Retour : `string`

Libellé à utiliser pour nommer le plug-in dans les différents interfaces utilisateur et journaux de DataHUB.

6.4.11.7 La méthode `Initialise`

Cette méthode est appelée par DataHUB au début d'une fabrication, avant que les données de la couche ne soient envoyées au plug-in. Elle transmet donc les données des *invariants de la fabrication* au plug-in. Le plug-in peut utiliser cette méthode pour allouer les ressources nécessaires pendant la durée de vie de l'instance, calculer les constantes de fabrication, etc.

Paramètre : `string`

Chaîne contenant un objet JSON conforme au schéma suivant :

```
{
   "AssemblyFolder":     string
   "OutputFolder":       string
   "NumberOfLasers":     unsigned integer
   "InitialisationData": JSON object
}
```

`AssemblyFolder: string`

Chemin d'accès au dossier de l'assembly qui contient ce plug-in :

"C:\ProgramData\Renishaw\DataHUB\PlugIns\PlugIn.Example"

`OutputFolder: string`

Chemin d'accès au dossier dans lequel l'instance de plug-in doit écrire les fichiers de sortie. Chaque instance de plug-in exécutée pour une tâche aura un dossier unique généré pour elle dans le dossier de sortie de la tâche. Le nom de ce dossier sera construit en utilisant le nom d'affichage et la version de l'API de plug-in, un horodatage et un GUID, en respectant la structure suivante :

« \[2] <Nom complet>.<Horodatage au format aaaaMMjj-hhmmss.fff> (<GUID>) »

`NumberOfLasers: unsigned integer`

Il s'agit d'une valeur de 1 à 4, définie par la configuration matérielle de la machine qui crée les données.

InitialisationData: Objet JSON

Contenu fourni au cours de l'étape « Specify Plug-in Initialisation Data » (Préciser les données d'initialisation du plug-in) du flux de travail de création de tâches dans DataHUB Generator. Voir la section « La chaîne de données d'initialisation » à la page 6-97, pour plus de détails sur la structure et l'utilisation de cette chaîne.

Retour : string

Chaîne contenant un objet JSON conforme au schéma suivant :

```
{
   “result”: string
   “message”: string
   “timeSeries”:
   [
      {
         “name”:                string
         “unitType”:            string
         “unit”:                string
         “absoluteOrRelative”: string
         “component”:           string
         “displayPrecision”:    unsigned integer
         “graphType”:           string
         “grouping”:            string
         “precision”:           unsigned integer
         [Any number of additional custom properties]
       }
    ]
 }
```

result: string

Indique si le plug-in a été initialisé avec succès ou non. Il doit s'agir de l'un ou l'autre :

- la valeur littérale « Success » qui indique que le plug-in a été initialisé avec succès et peut commencer le traitement, ou
- la valeur littérale « Failure » qui indique que le plug-in a rencontré une erreur et qu'il sera arrêté sans recevoir de données.

Cette propriété est obligatoire.

message: string

Message contenant plus de détails qui sera ajouté au résultat et enregistré dans le journal d'audit. Cette propriété est facultative : si elle n'est pas présente, seul le résultat sera enregistré dans le journal d'audit.

`timeSeries: JSON array`

Tableau contenant les définitions des séries temporelles dans lequel le plug-in publiera des données pendant le traitement. Cette propriété est facultative : si elle n'est pas présente ou s'il s'agit d'un tableau vide, DataHUB ne créera aucune série temporelle dans Renishaw Central.

`timeSeries[n].name: string`

Nom de la série temporelle. Cette propriété est obligatoire.

`timeSeries[n].unitType: string`

Famille d'unités que les données représentent. Cela permet une conversion en aval vers un type d'unité préféré pour l'affichage. Les types d'unités prédéfinis sont les suivants :

- **Acceleration** (Accélération)
- **Angle**
- **Binary** (Binaire)
- **Distance**
- **Flowrate** (Débit)
- **Humidity** (Humidité)
- **MicroDistance**
- **Percentage** (Pourcentage)
- **Pressure** (Pression)
- **Speed** (Vitesse)
- **Temperature** (Température)
- **Dimensionless** (Sans dimension)

Cette propriété est obligatoire.

`timeSeries[n].unit: string`

Par convention, Renishaw Central préfère les unités du SI, s'attend à ce que les relations « par » soient indiquées à l'aide du caractère « / » et présente les exposants numériques sous forme de nombres. Par exemple, l'accélération est généralement représentée par m/s^2. Pour la température et les degrés, utilisez le caractère « ° » le cas échéant, par exemple °C pour Celsius. Lorsque les valeurs n'ont pas de dimension, utiliser la valeur littérale `Dimensionless`. Cette propriété est obligatoire.

`timeSeries[n].absoluteOrRelative: string`

Indication destinée aux applications pour savoir si les valeurs sont absolues ou relatives par rapport à la valeur précédente. Il doit s'agir de l'un ou l'autre :

- la valeur littérale « Absolute » qui indique que les valeurs sont absolues, ou
- la valeur littérale « Relative » qui indique que les valeurs sont relatives par rapport à la valeur précédente.

Cette propriété est facultative : si elle n'est pas présente, la valeur « Absolute » lui sera attribuée.

`timeSeries[n].component: string`

Associe la série temporelle à un aspect distinct d'un dispositif, généralement une entité physique telle qu'une station météorologique, un outil ou un système logiciel. Cette propriété est facultative : si elle n'est pas présente, la valeur « Machine » lui sera attribuée.

`timeSeries[n].description:` `string`

Description de la série temporelle. Cette propriété est facultative.

`timeSeries[n].displayHintPrecision:` `unsigned integer`

Indication destinée aux applications qui s'intègrent à Renishaw Central à propos de la précision avec laquelle les valeurs de la série temporelle doivent être affichées. Cette propriété est facultative.

`timeSeries[n].displayHintSampleRate:` `string`

Indication destinée aux applications qui s'intègrent à Renishaw Central à propos du taux d'échantillonnage à utiliser. En cas d'omission, la plupart des applications en aval utiliseront la valeur « default ». Les taux d'échantillonnage les plus connus sont les suivants :

- **Raw** (Brut)
- **Default** (Par défaut)
- **Hour** (Heure)
- **Day** (Jour)
- **Week** (Semaine)
- **Month** (Mois)

Cette propriété est facultative.

`timeSeries[n].graphType:` `string`

Indication destinée aux applications qui s'intègrent à Renishaw Central afin de spécifier le type de graphique sur lequel la série temporelle devrait être tracée. Les types de graphique les plus connus sont les suivants :

- **Bar** (Barre)
- **Binary** (Binaire) (spécifiquement pour afficher les données On-Off)
- **Line** (Ligne)

Cette propriété est facultative.

`timeSeries[n].grouping:` `string`

Groupe auquel cette série temporelle doit être associée. Cette propriété est facultative.

`timeSeries[n].precision:` `unsigned integer`

Enregistrement de la précision avec laquelle les données ont été saisies/calculées. Cette propriété est facultative.

`timeSeries[n]` *Additional Properties*

Le plug-in peut inclure un nombre quelconque de propriétés supplémentaires qui seront stockées dans Renishaw Central et auxquelles un logiciel personnalisé pourra accéder. Elles ne peuvent pas utiliser les noms réservés « machine » ou « source ».

Si la chaîne renvoyée n'est pas conforme au schéma ci-dessus, ceci est considéré comme une erreur et la chaîne complète est enregistrée dans le journal d'audit.

6.4.11.8 La méthode *Process*

DataHUB transmet des sous-sections des données de fabrication dans appels séquentiels : le nombre total d'appels varie en fonction de la taille des données. Dans cette méthode, le plug-in doit effectuer la majeure partie de son travail d'analyse.

Paramètre : IReadOnlyCollection<Token>

Section du flux de jetons de fabrication.

Retour : `string`

Chaîne vide, ou une chaîne contenant « Success » ou « Failure », ou une chaîne contenant un objet JSON conforme au schéma suivant :

```
{
   “result”:  string
   “message”: string
   “alerts”:
   [
      {
         “description”: string
         “level”:    string
         “name”:    string
         “time”:    unsigned integer
      }
   ],
   “analysisResults”:
   [
      {
         “name”: string
         “time”: unsigned integer
         Any number of additional custom properties
      }
   ],
   “timeSeries”:
   [
      {
         “name”: string
         “unitType”: string
         “unit”: string
         “values”:
         [
            {
```

```
                "time":  unsigned integer
                "value": number
            }
        ],
        "limits":
         [
            {
               "time":        unsigned integer
               "lowerDisplayLimit":  number
               "upperDisplayLimit":  number
               "lowerWarningLimit":  number
               "upperWarningLimit":  number
               "lowerOperatingLimit": number
               "upperOperatingLimit": number
            }
        ]
     }
   ]
}
```

result: string

Cela indique si le plug-in a traité les données avec succès ou non. Il doit s'agir soit de

- la valeur littérale « Success » qui indique que le plug-in a exécuté le traitement avec succès, ou
- la valeur littérale « Failure » qui indique que le plug-in a rencontré une erreur. Le plug-in reste actif et continue à recevoir les données suivantes.

Cette propriété est obligatoire.

message: string

Message contenant plus de détails qui sera ajouté au résultat et enregistré dans le journal d'audit. Cette propriété est facultative : si elle n'est pas présente, rien ne sera enregistré dans le journal d'audit.

alertes: Matrice JSON

Une matrice contenant les alertes que le plug-in souhaite publier dans Renishaw Central au cours du traitement. Cette propriété est facultative : si elle n'est pas présente ou s'il s'agit d'un tableau vide, DataHUB ne tentera pas de publier d'alertes dans Renishaw Central.

alerts[n].description : string

Description qui fournit des détails sur l'alerte. Cette propriété est obligatoire.

alerts[n].level : string

Une des valeurs littérales suivantes indiquant la gravité de l'alerte :

- **Info**
- **Warning** (Avertissement)
- **Error** (Erreur)

Cette propriété est obligatoire.

alerts[n].name : string

Nom de l'alerte. Cette propriété est obligatoire.

alerts[n].time: unsigned integer

Heure à laquelle l'alerte s'est produite, en microsecondes par rapport à l'heure de début de la couche. Cette propriété est obligatoire.

analysisResults : Matrice JSON

Une matrice contenant les résultats d'analyse que le plug-in souhaite publier dans Renishaw Central au cours du traitement. Cette propriété est facultative : si elle n'est pas présente ou s'il s'agit d'un tableau vide, DataHUB ne tentera pas de publier de résultats d'analyse dans Renishaw Central.

analysisResults [n].name: string

Nom du résultat de l'analyse. Cette propriété est obligatoire.

analysisResults [n].time: unsigned integer

Heure à laquelle le résultat d'analyse est arrivé, en microsecondes par rapport à l'heure de début de la couche. Cette propriété est obligatoire.

analysisResults[n] *Additional Properties*

Le résultat d'analyse peut inclure un nombre quelconque de propriétés supplémentaires qui seront stockées dans Renishaw Central et auxquelles un logiciel personnalisé pourra accéder. L'un des noms réservés suivants ne peut pas être utilisé :

- **created** (créé)
- **machine**
- **source**
- **type**

`timeSeries: JSON Array`

Une matrice contenant les données de la série temporelle que le plug-in souhaite ajouter à la série temporelle définie dans la méthode *Initialise*. Cette propriété est facultative : si elle n'est pas présente ou s'il s'agit d'un tableau vide, DataHUB ne mettra à jour les données d'aucune série temporelle dans Renishaw Central.

`timeSeries[n].name: string`

Nom de la série temporelle tel qu'il a été défini lors de l'initialisation. Cette propriété est obligatoire.

`timeSeries[n].unitType: string`

Type de la série temporelle tel qu'il a été défini lors de l'initialisation. Cette propriété est obligatoire.

`timeSeries[n].unit: string`

Unité de la série temporelle telle qu'elle a été définie lors de l'initialisation. Cette propriété est obligatoire.

`timeSeries[n].values: Matrice JSON`

Une matrice contenant les paires temps-valeur à ajouter à la série temporelle. Une série temporelle doit avoir une propriété *values*, une propriété *limits* ou les deux.

`timeSeries[n].values[m].time: unsigned integer`

Temps auquel correspond la valeur, en µs par rapport à l'heure de début de la couche. Cette propriété est obligatoire.

`timeSeries[n].values[m].value: number`

Valeur. Cette propriété est obligatoire.

`timeSeries[n].limits: Matrice JSON`

Une matrice contenant les limites des données. Il existe trois familles de limites :

1. **Operating** (En fonctionnement)
2. **Warning** (Avertissement)
3. **Display** (Affichage)

Les applications qui s'intègrent à Renishaw Central peuvent utiliser ces limites pour faciliter l'affichage et déclencher des actions lorsque les points de données de la série temporelle dépassent ces différentes limites.

Les limites prennent effet à partir de l'heure indiquée par la propriété *time* jusqu'à l'heure indiquée par les limites suivantes ajoutées (les dernières limites définies prennent effet à perpétuité).

Il n'est pas nécessaire qu'un plug-in fournisse des limites pour une série temporelle ; il se peut que les données n'aient pas de plage de fonctionnement naturelle ou de plage d'affichage pertinente, par exemple. Il n'est pas non plus nécessaire de modifier les limites une fois qu'elles ont été fixées. Il est prévu qu'elles puissent être modifiées dans les cas où cela s'avérerait approprié.

Une série temporelle doit avoir une propriété *values*, une propriété *limits* ou les deux.

timeSeries[n].limits[m].time: unsigned integer

Heure à partir de laquelle la limite doit prendre effet, en µs, par rapport à l'heure de début de la couche. Cette propriété est obligatoire.

timeSeries[n].limits[m].lowerDisplayLimit: number

Valeur de la limite inférieure d'affichage. Cette propriété est facultative : si elle n'est pas présente, la limite inférieure d'affichage ne sera pas modifiée par rapport à sa valeur actuelle.

timeSeries[n].limits[m].upperDisplayLimit: number

Valeur de la limite supérieure d'affichage. Cette propriété est facultative : si elle n'est pas présente, la limite supérieure d'affichage ne sera pas modifiée par rapport à sa valeur actuelle.

timeSeries[n].limits[m].lowerWarningLimit: number

Valeur de la limite inférieure d'avertissement. Cette propriété est facultative : si elle n'est pas présente, la limite inférieure d'avertissement ne sera pas modifiée par rapport à sa valeur actuelle.

timeSeries[n].limits[m].upperWarningLimit: number

Valeur de la limite supérieure d'avertissement. Cette propriété est facultative : si elle n'est pas présente, la limite supérieure d'avertissement ne sera pas modifiée par rapport à sa valeur actuelle.

timeSeries[n].limits[m].lowerOperationalLimit: number

Valeur de la limite inférieure de fonctionnement. Cette propriété est facultative : si elle n'est pas présente, la limite inférieure de fonctionnement ne sera pas modifiée par rapport à sa valeur actuelle.

timeSeries[n].limits[m].upperOperationalLimit: number

Valeur de la limite supérieure de fonctionnement. Cette propriété est facultative : si elle n'est pas présente, la limite supérieure de fonctionnement ne sera pas modifiée par rapport à sa valeur actuelle.

Si la chaîne renvoyée n'est pas conforme au schéma ci-dessus, ceci est considéré comme une erreur et la chaîne complète est enregistrée dans le journal d'audit.

6.4.11.9 La méthode *Shutdown*

La méthode *Shutdown* est appelée exactement une fois et sera appelée soit lorsque la méthode *Initialise* renvoie une erreur, soit lorsqu'il n'y a plus de données à traiter (parce que toutes les données attendues ont été traitées ou que la tâche a été annulée). Une fois appelée, le plug-in ne recevra plus d'autres appels de quelque nature que ce soit.

Dans cette méthode, le plug-in doit effectuer le traitement final des données de fabrication, libérer les ressources acquises, etc.

Paramètres :

Aucun.

Retour : `string`

Si la chaîne contient le mot « Success », cela indique que le plug-in s'est arrêté avec succès. Si ce n'est pas le cas, cela indique que le plug-in a rencontré une erreur. Dans les deux cas, la chaîne est intégralement enregistrée dans les journaux d'audit.

6.4.11.10 La chaîne de données d'initialisation

DataHUB initialise un plug-in V2.0 sous forme d'une chaîne formatée en JSON contenant les valeurs des paramètres globaux de fabrication et la chaîne d'initialisation du plug-in spécifiée dans le flux de production de DataHUB Generator pour la tâche de traitement du plug-in pour la fabrication.

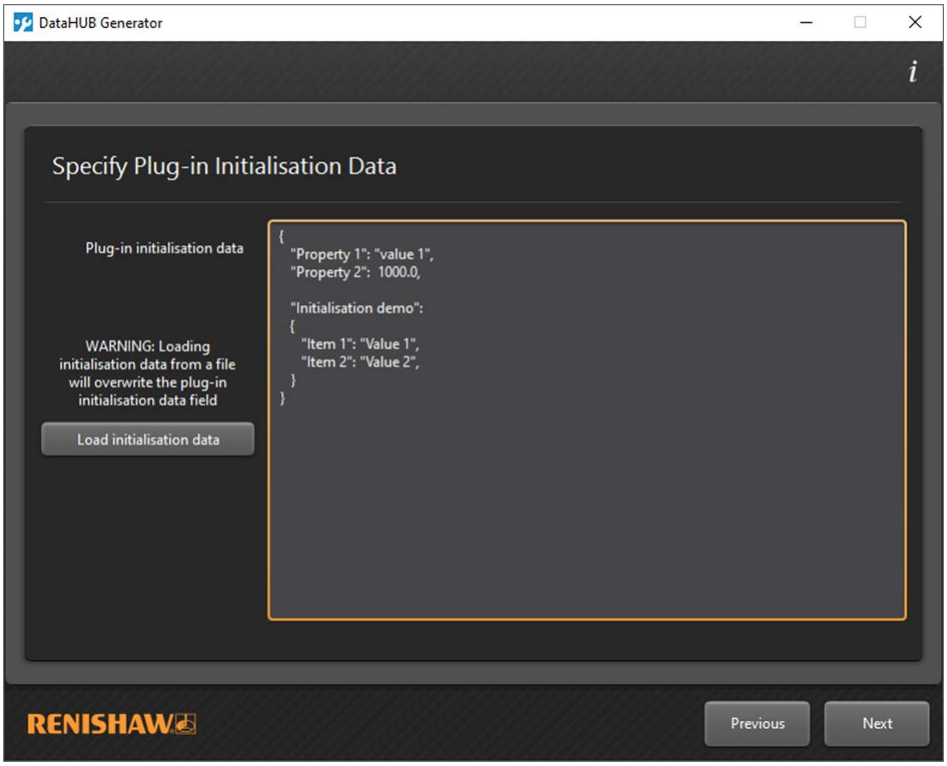


Figure 42 Chaîne JSON d'entrée du plug-in

La chaîne saisie dans le flux de production de DataHUB Generator est elle-même au format JSON. En ajoutant des propriétés et des objets, des données spécifiques à chaque type de plug-in peuvent être envoyées à la méthode *Initialise* de l'instance.

Un exemple de chaîne d'initialisation de plug-in pourrait être le suivant :

```
{
   “Property 1”: “value 1”,
   “Property 2”:  1000.0,
   “Initialisation demo”:
   {
      “Item 1”: “Value 1”,
      “Item 2”: “Value 2”,
   }
}
```

L'utilisation d'une bibliothèque d'analyse JSON (par exemple, Newtonsoft.Json ou System.Text.Json) dans le plug-in simplifie le processus d'extraction des valeurs contenues dans la chaîne d'initialisation, par exemple :

```
public string Initialise(string initialisationData)
{
   var instanceData         = JObject.Parse(initialisationData);
   var assemblyFolder       = instanceData[“AssemblyFolder”];
   var outputFolder         = instanceData[“OutputFolder”];
   var numberOfLasers       = instanceData[“NumberOfLasers”];
   var dataHUBGeneratorData = JObject.Parse(instanceData[“InitialisationData”].Value<string>());
   var property1            = dataHUBGeneratorData[“Property 1”];
   var property2            = dataHUBGeneratorData[“Property 2”];
   var item1                = dataHUBGeneratorData[“Initialisation demo”][“Item 1”];
   var item2                = dataHUBGeneratorData[“Initialisation demo”][“Item 2”];
```

Les données d'initialisation de l'instance spécifiées dans DataHUB Generator sont elles-mêmes une chaîne JSON, il faut donc noter qu'elles sont analysées en tant que JObject avant d'accéder aux propriétés qu'elles contiennent. Les valeurs des propriétés `Property1` et `Property2`, ainsi que les valeurs des propriétés de l'objet démo `Initialisation` imbriqué sont affichées dans la fenêtre de surveillance de VisualStudio :

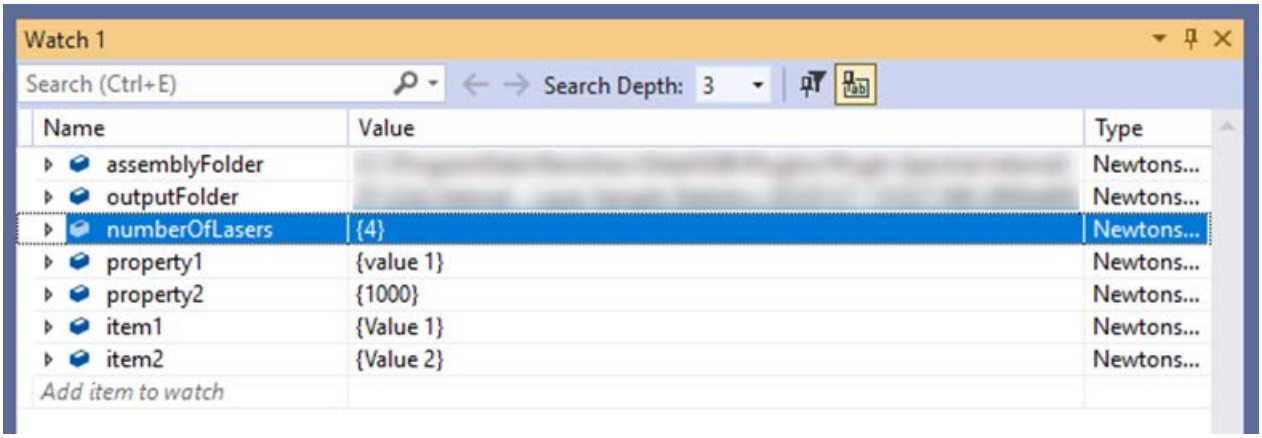


Figure 43 Chaîne de données d'initialisation du plug-in dans l'appel *Initialise*

Plusieurs plug-ins sont généralement utilisés sur un PCCD, c'est pourquoi une stratégie recommandée pour formater la chaîne du DataHUB Generator consiste à spécifier un objet JSON nommé pour chaque type de plug-in :

```
{
   "PlugIn.TypeA":
   {
      "PlugIn parameter 1":1,
      "PlugIn parameter 2":2,
   },
   "PlugIn.TypeB":
  {
      "PlugIn parameter 1":100,
      "PlugIn parameter 2":200,
   }
}
```

Pour accéder aux données d'initialisation propres à un plug-in, il suffit d'accéder à l'objet imbriqué correspondant, c'est-à-dire aux plug-ins TypeA et TypeB respectivement :

```
var dataA = dataHUBGeneratorData["PlugIn.TypeA"];
var dataB = dataHUBGeneratorData["PlugIn.TypeB"];
```

6.4.11.11 Types de jetons

6.4.11.11.1 DataSourceInformation

Ce jeton décrit les caractéristiques d'une source de données de surveillance de procédé.

```
public string      File { get; }
public LayerNumber Layer { get; }
public LaserID     Laser { get; }
public DateTime    LayerStartTime { get; }
public double      MillisecondsPerSample { get; }
public uint        TotalNumberOfSamples { get; }
```

6.4.11.11.2 EndOfLayerData

Ce jeton indique la fin de toutes les données pour une couche.

```
public LayerNumber Layer { get; }
public uint        LastSampleTime { get; }
```

6.4.11.11.3 EndOfLayerLaserData

Ce jeton marque la fin des échantillons pour un laser, pour une couche.

```
public LayerNumber Layer { get; }
public LaserID     Laser { get; }
```

6.4.11.11.4 PacketOfSamples

Ce jeton contient une collection d'échantillons.

```
public IreadOnlyList<Sample> Samples { get; }
```

6.4.11.11.5 Sample

Un jeton d'échantillon contient des données collectées à un moment précis du processus de fabrication de la couche. Pour une couche, pour un laser, il y aura plusieurs jetons Sample.

- `public uint   Time               { get; }`
- `public double DemandX            { get; }`
- `public double DemandY            { get; }`
- `public double DemandFocus        { get; }`
- `public int    DemandLaserPower   { get; }`
- `public int    LaserModulate      { get; }`
- `public int    MeltVIEWPlasma     { get; }`
- `public int    MeltVIEWMeltpool   { get; }`
- `public int    LaserVIEW          { get; }`
- `public int    LaserOutputPower   { get; }`
- `public int    LaserBackReflection { get; }`

6.4.11.11.6 Split

Ce jeton n'a pas de propriétés.

6.4.11.11.7 StartOfLayerData

Ce jeton marque le début de la série de jetons concernant une couche.

```
public LayerNumber Layer { get; }
```

6.4.11.11.8 StartOfLayerLaserData

Ce jeton marque le début de la série de jetons correspondant à un laser pour une couche.

```
public LayerNumber Layer { get; }
public LaserID     Laser { get; }
```

6.4.11.11.9 SummaryPacket

Ce jeton contient les valeurs synthétisées produites par les méthodes *Summary* fournies au filtre `SummarisePacketOfSamples`. L'ensemble des valeurs synthétisées sont contenues dans les instances de la classe imbriquée `SummaryValues`. Comme les méthodes *Summary* peuvent produire des résultats à valeurs fractionnaires, le type de chaque propriété de `SummaryValues` est double.

```
Public uint Time   { get; }
public uint Duration { get; }
public IreadOnlyDictionary<string, SummaryValues> Summary { get; }
public class SummaryValues
{
   public double DemandX             { get; }
   public double DemandY             { get; }
   public double DemandFocus         { get; }
   public double DemandLaserPower    { get; }
   public double MeltVIEWPlasma      { get; }
   public double MeltVIEWMeltpool    { get; }
   public double LaserVIEW           { get; }
   public double LaserOutputPower    { get; }
   public double LaserBackReflectio  { get; }
}
```

6.4.11.11.10 Token

Classe de base dont dérivent tous les jetons. Cette classe ne contient aucune donnée.

6.4.11.11.11 LaserID et LayerNumber

Le type sous-jacent des ID des lasers et des numéros de couche est un nombre entier (*integer*). Ceci permet d'éviter toute erreur d'affectation accidentelle, par exemple, d'un numéro de couche à un ID de laser, car ces derniers assurent la sécurité du type lors de la création des différents jetons mentionnés précédemment.

6.4.11.12 Filtres

L'API fournit un ensemble de filtres conçus pour effectuer des tâches courantes lors du traitement d'un flux de jetons.

6.4.11.12.1 EmitSampleOnlyIf

Ce filtre compare la valeur de l'une des propriétés d'un jeton `Sample` en entrée et n'émet le `Sample` que si la valeur passe le test. Tous les types de jetons autres que `Sample` sont émis vers le flux de sortie.

Les valeurs de propriété suivantes sont proposées :

- DemandX
- DemandY
- DemandFocus
- DemandLaserPower
- MeltVIEWPlasma
- MeltVIEWMeltpool
- LaserVIEW
- LaserOutputPower
- LaserBackReflection
- LaserModulate
- Les tests d'égalité sont les suivants :
- EqualTo
- NotEqualTo
- GreaterThan
- GreaterThanOrEqualTo
- LessThan
- LessThanOrEqualTo
- CustomTest
- IsOn (LaserModulate uniquement)

La dernière variante du filtre permet de configurer un test de comparaison personnalisable. Toute fonction qui prend deux entrées d'entiers et renvoie un booléen peut être utilisée, par exemple :

```
var customTest = EmitSampleIf.DemandX.CustomTest(v => v % 2 == 0);
```

Le test personnalisé ici passerait les échantillons dont la valeur DemandX est paire.

6.4.11.12.2 SplitWhen

Ce filtre compare la valeur de la propriété d'un jeton `Sample` d'entrée à la valeur de la propriété du jeton précédent et, si elle est différente, ajoute un jeton `Split` avant d'émettre le jeton d'entrée, ajoutant ainsi un `Split` entre les deux jetons. Tous les types de jetons autres que `Sample` sont émis vers le flux de sortie.

Les variantes suivantes sont disponibles :

- DemandXChanges
- DemandYChanges
- DemandFocusChanges
- LaserModulateChanges
- DemandLaserPowerChanges

REMARQUE : seules les propriétés de la demande sont fournies ; les autres propriétés de surveillance de procédé présentent une variabilité élevée, d'un échantillon à l'autre, et sont donc jugées inadaptées pour fractionner le flux de jetons.

6.4.11.12.3 CollateInto

Ce filtre regroupe tous les jetons `Sample` délimités par des jetons `Split` en un seul jeton `PacketOfSamples`. Les jetons `Split` et les jetons `Sample` sont ignorés (« avalés »). Tous les autres types de jetons sont émis vers le flux de sortie.

```
public static Func<Token, IEnumerable<Token>> PacketOfSamples()
```

La portée de ce mappage mérite d'être soulignée. Si un flux de jetons a été précédemment fractionné par des jetons `Split` (par exemple, lorsque la modulation du laser, les positions, etc. ont été modifiées), le flux de sortie résultant ne contient pas de jetons `Sample` ou `Split` , ceux-ci ayant été remplacés par des jetons `PacketOfSamples` .

6.4.11.12.4 SummarisePacketOfSamples

Ce filtre traite les jetons `PacketOfSamples` produits par `CollateInto` et exécute une ou plusieurs fonctions *summary* sur les valeurs des données d'échantillon regroupées. Le constructeur de ce filtre permet de configurer une ou plusieurs fonctions *summary* nommées, et les résultats de chacune sont stockés dans le jeton de sortie.

```
public static Func<Token, IEnumerable<Token>> Summarise(params SummaryMethod[] summaryMethods)
```

Une instance de SummaryMethod est construite avec un nom et une fonction *summary* :

```
public SummaryMethod(string description, Func<IReadOnlyList<int>, double> func)
```

6.4.11.12.5 Pipeline

Ces deux utilitaires sont nécessaires pour construire des filtres dans des pipelines :

```
public static Func<Token, IEnumerable<Token>> First(<Token, IEnumerable<Token>> f1);
public static Func<Token, Ienumerable<Token>> Then(Func<Token, IEnumerable<Token>> f0,
                                                   Func<Token, IEnumerable<Token>> f1);
```

6.4.11.13 Utilitaires (*Helpers*)

L'API fournit quelques classes qui définissent les opérations courantes utilisées lors de l'écriture de plug-ins.

6.4.11.13.1 PositionHelpers

Cette classe fournit des fonctions permettant d'arrondir les valeurs doubles de manière cohérente :

```
public static double RoundToMicrons(double value, double microns);
public static int Round(double value);
```

6.4.11.13.2 Utilitaires pour les chaînes de retour

L'API fournit des méthodes utilitaires pour construire les chaînes de retour nécessaires à DataHUB à chaque étape du plug-in*(Initialise*, *Process* et *Shutdown*).

REMARQUE : DataHUB s'attend à recevoir une chaîne de retour JSON valide, conforme aux spécifications de l'API de la chaîne de retour. Il est important que les caractères spéciaux soient échappés par une barre oblique inverse supplémentaire dans les chaînes. Dans le cas contraire, une chaîne JSON non valide peut être envoyée à DataHUB et rejetée par celui-ci. Par exemple :

```
ProcessReturns.Success("this is a badly \"escaped\" message");
ProcessReturns.Success("this is a correctly \\\"escaped\\\" message");
```

Second exemple :

```
ProcessReturns.Success("this is a badly \nescaped message");
ProcessReturns.Success("this is a correctly \\nescaped message");
```

6.4.11.13.3 Retours *Initialise*

Les utilitaires qui permettent de créer des chaînes de retour de plug-in simples à partir de la fonction *Initialise* vers DataHUB sont les suivants :

```
public static string Success();
public static string Success(string message);
public static string Success(IReadOnlyCollection<string> messages);
public static string Failure();
public static string Failure(string message);
public static string FailureDueToThisException(Exception e);
```

Pour créer une ou plusieurs séries temporelles dans Renishaw Central, vous pouvez utiliser l'utilitaire suivant :

```
public static string SuccessAndCreateTimeSeries(IEnumerable<TimeSeries> timeSeries)
public static string SuccessAndCreateTimeSeries(string message, IEnumerable<TimeSeries> timeSeries)
```

Un objet `TimeSeries` doit être construit avec un nom, un type d'unité et une unité. La fabrique d'objets `TimeSeries` fournit une interface fluide avec des types d'unités et des unités bien connus afin de faciliter le processus de création d'objets et de limiter le risque d'erreurs. C'est ce que montre l'exemple ci-dessous :

```
var titaniumTimeSeries = TimeSeries.Factory().
                                    Name("Thermal expansion").
                                    Temperature().
                                    Celsius().
                                    Grouping("Titanium").
                                    Create();
```

6.4.11.13.4 Retours *Process*

Les utilitaires qui permettent de créer des chaînes de retour de plug-in simples à partir de la fonction *Process* vers DataHUB sont les suivants :

```
public static string Success();
public static string SuccessWithoutInformation();
public static string Success(string message);
public static string Success(IReadOnlyCollection<string> messages);
public static string Failure();
public static string Failure(string message);
public static string FailureDueToThisException(Exception e);
```

Grâce aux utilitaires suivants, les chaînes de retour du plug-in de la fonction *Process* peuvent être utilisées pour indiquer la réussite ou l'échec et envoyer des données telles que des mises à jour de séries temporelles, des alertes et des résultats d'analyse à Renishaw Central.

```
public static string SuccessAndSendData(string message, IEnumerable<UpdateTimeSeries> timeSeries)
public static string SuccessAndSendData(string message, IEnumerable<Alert> alerts)
public static string SuccessAndSendData(string message,
                                        IEnumerable<AnalysisResult>
                                        analysisResults)
public static string SuccessAndSendData(string message,
                                        IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts)
public static string SuccessAndSendData(string message,
                                        IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<AnalysisResult> analysisResult)
public static string SuccessAndSendData(string message,
                                        IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string SuccessAndSendData(string message,
```

```
                                        IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string SuccessAndSendData(IEnumerable<UpdateTimeSeries> timeSeries)
public static string SuccessAndSendData(IEnumerable<Alert> alerts)
public static string SuccessAndSendData(IEnumerable<AnalysisResult> analysisResults)
public static string SuccessAndSendData(IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts)
public static string SuccessAndSendData(IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<AnalysisResult> analysisResult)
public static string SuccessAndSendData(IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string SuccessAndSendData(IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(IEnumerable<UpdateTimeSeries> timeSeries)
public static string FailureAndSendData(IEnumerable<Alert> alerts)
public static string FailureAndSendData(IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts)
public static string FailureAndSendData(IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(string message, IEnumerable<UpdateTimeSeries> timeSeries)
public static string FailureAndSendData(string message, IEnumerable<Alert> alerts)
public static string FailureAndSendData(string message, IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(string message,
                                        IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts)
public static string FailureAndSendData(string message,
                                        IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(string message,
                                        IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
public static string FailureAndSendData(string message,
```

```
                                        IEnumerable<UpdateTimeSeries> timeSeries,
                                        IEnumerable<Alert> alerts,
                                        IEnumerable<AnalysisResult> analysisResults)
```

Les `TimeSeries` définies au cours de l'étape *Initialise* peuvent être mises à jour avec des valeurs, des limites ou les deux. L'objet `TimeSeries` fournit une méthode Update qui renvoie un objet `UpdateTimeSeries`, comme le montre l'exemple ci-dessous :

```
var update = titaniumTimeSeries.
      Update().
      WithValues((0, 32), (100, 67)).
      WithLimits(TimeSeriesLimit.Factory().
                                Time(0).
                                LowerDisplayLimit(0).
                                NoUpperDisplayLimit().
                                LowerWarningLimit(30).
                                NoUpperWarningLimit().
                                LowerOperatingLimit(10).
                                NoUpperOperatingLimit());
```

Les alertes peuvent être déclenchées sur Renishaw Central en construisant un objet `Alert` et en le passant à la méthode utilitaire `SendData` appropriée. La fabrique d'objets `Alert` dispose d'une interface de constructeur simple, comme le montre l'exemple ci-dessous :

```
var overheatingAlert = Alert.Factory().
                            Description("Part temperature has exceeded the threshold").
                            Warning().
                            Name("Overheating").
                            Time(350);
```

Enfin, les résultats d'analyse peuvent également être transmis à Renishaw Central en construisant un objet `AnalysisResult` et en le transmettant à la méthode utilitaire `Send Data` appropriée. La fabrique d'objets `AnalysisResult` dispose d'une interface de constructeur simple, comme le montre l'exemple ci-dessous :

```
var analysisResult = AnalysisResult.Factory().
                                    Name("Thermal expansion rate").
                                    Time(250).
                                    AddAdditionalProperty("Rate", 0.05).
                                    Create();
```

6.4.11.13.5 Retour *Shutdown*

Les utilitaires qui permettent de créer des chaînes de retour de plug-in simples à partir de la fonction *Shutdown* vers DataHUB sont les suivants :

```
public static string Success();
public static string Success(string message);
public static string Success(IReadOnlyCollection<string> messages);
public static string Failure();
public static string Failure(string message);
public static string FailureDueToThisException(Exception e);
```

6.5 Plug-in Export Packets

Les modules matériels LaserVIEW et MeltVIEW collectent des données échantillonnées à une fréquence de 100 kHz, chaque échantillon discret contenant plusieurs valeurs. Alors que les données à cette résolution sont utiles pour certaines tâches d'analyse, une représentation à plus faible résolution des mêmes échantillons peut s'avérer adéquate pour de nombreuses analyses, c'est pourquoi le plug-in quantifie les données en « paquets », où un paquet est défini par les éléments suivants :

- le laser se déclenche, et
- les positions X et Y de la demande sont constantes.

Cette définition regroupe effectivement l'ensemble des échantillons prélevés au cours d'une seule exposition du laser sur le lit de poudre. À partir de chaque ensemble, les valeurs résumées *mean* et *median* (moyenne et médiane) des capteurs de surveillance de procédé Laser/MeltVIEW sont produites. Il convient de noter qu'en raison des caractéristiques de fonctionnement du laser, les échantillons au début d'une exposition peuvent contenir des valeurs Laser/MeltVIEW légèrement faibles. La valeur médiane peut donc être un meilleur représentant synthétique des données de l'échantillon que la valeur moyenne.

6.5.1 Exécution du plug-in

Pour utiliser ce plug-in, lancez DataHUB Generator et sélectionnez le dossier *Build Data* contenant les fichiers de surveillance de procédé de fabrication. Sélectionnez un dossier pour la sortie du plug-in. Si nécessaire (par exemple, le fichier d'information sur la fabrication *BuildStarted aaaa.mm.jj.HH.mm.ss.dhxml* n'a pas été trouvé dans le dossier d'entrée), assurez-vous que le *nombre correct de lasers* est défini en fonction de la fabrication en question : le plug-in ne traitera pas les données s'il existe un décalage entre le nombre de lasers sélectionné et le nombre réel de lasers équipés des modules matériels LaserVIEW/MeltVIEW sur la machine de fabrication additive qui a produit les données. Les deux autres valeurs, **Layer Spacing** (Espacement des couches) et **Build Height** (Hauteur de la fabrication), peuvent être laissées telles quelles, car elles ne sont pas requises par le plug-in.

Saisissez une description et appuyez sur **Next** (Suivant).

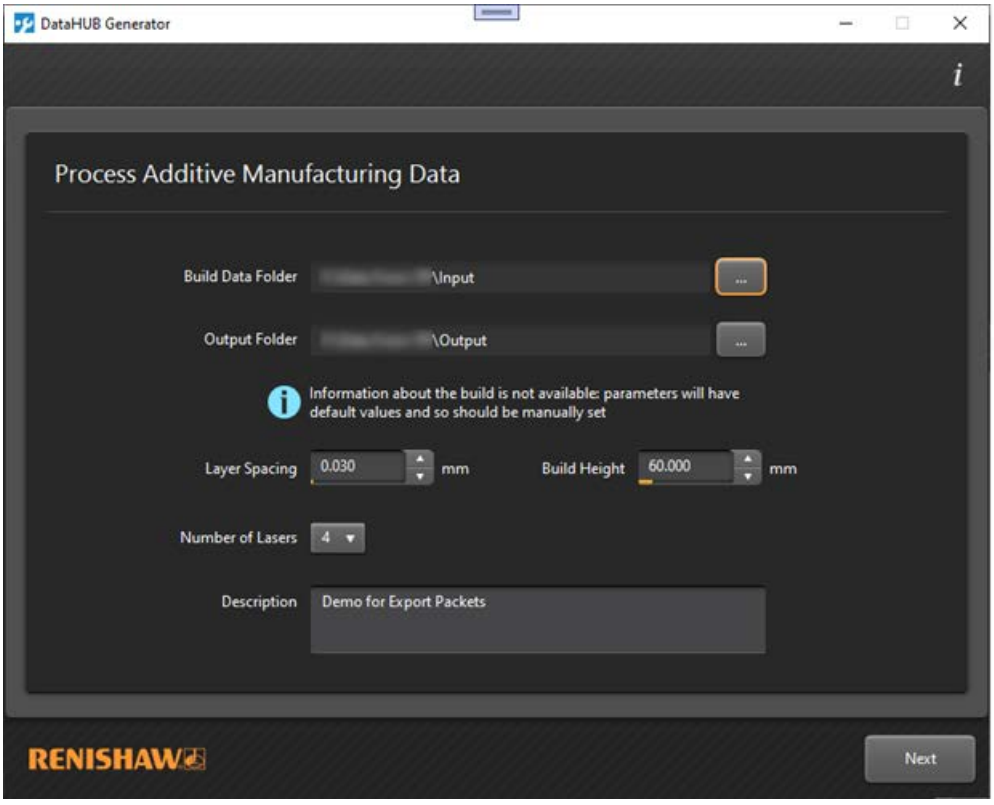


Figure 44 Étape 1 du flux de production : configurer les données

Dans la partie suivante du flux de production, sélectionnez **Process LaserVIEW and/or MeltVIEW data via plug-ins** (Traiter les données LaserVIEW et/ou MeltVIEW via des plug-ins), puis cliquez sur **Next** (Suivant).

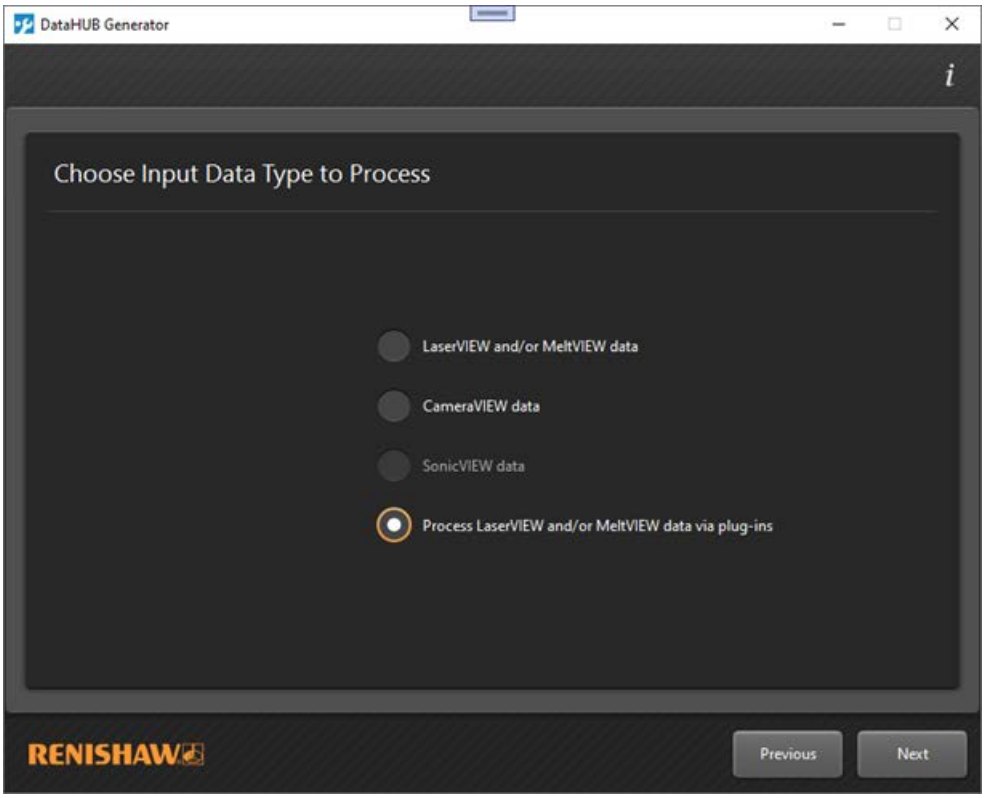


Figure 45 Étape 2 du flux de production : sélection du traitement de plug-in

Comme ce plug-in ne nécessite pas de données d'initialisation, la zone de texte peut rester vierge. Appuyez sur **Next** (Suivant).

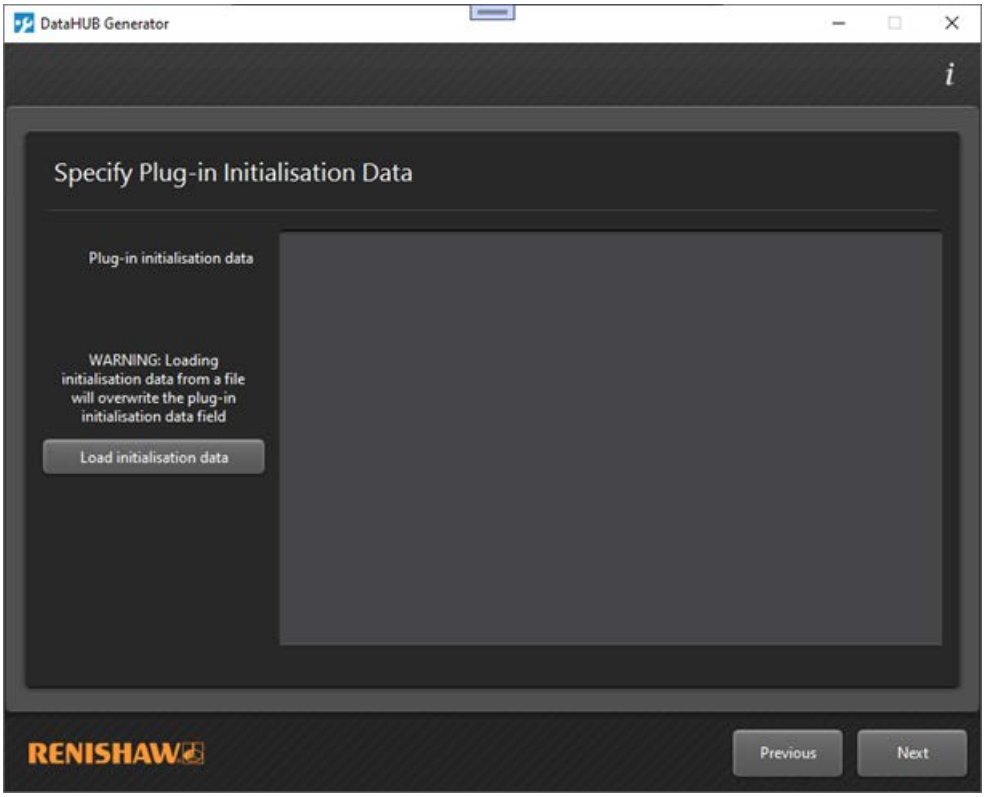


Figure 46 Étape 3 du flux de production : initialisation du plug-in

Dans la dernière partie du flux de production, cliquez sur **Trigger LaserVIEW/MeltVIEW Plug-in Processing** (Déclencher le traitement du plug-in LaserVIEW/MeltVIEW).

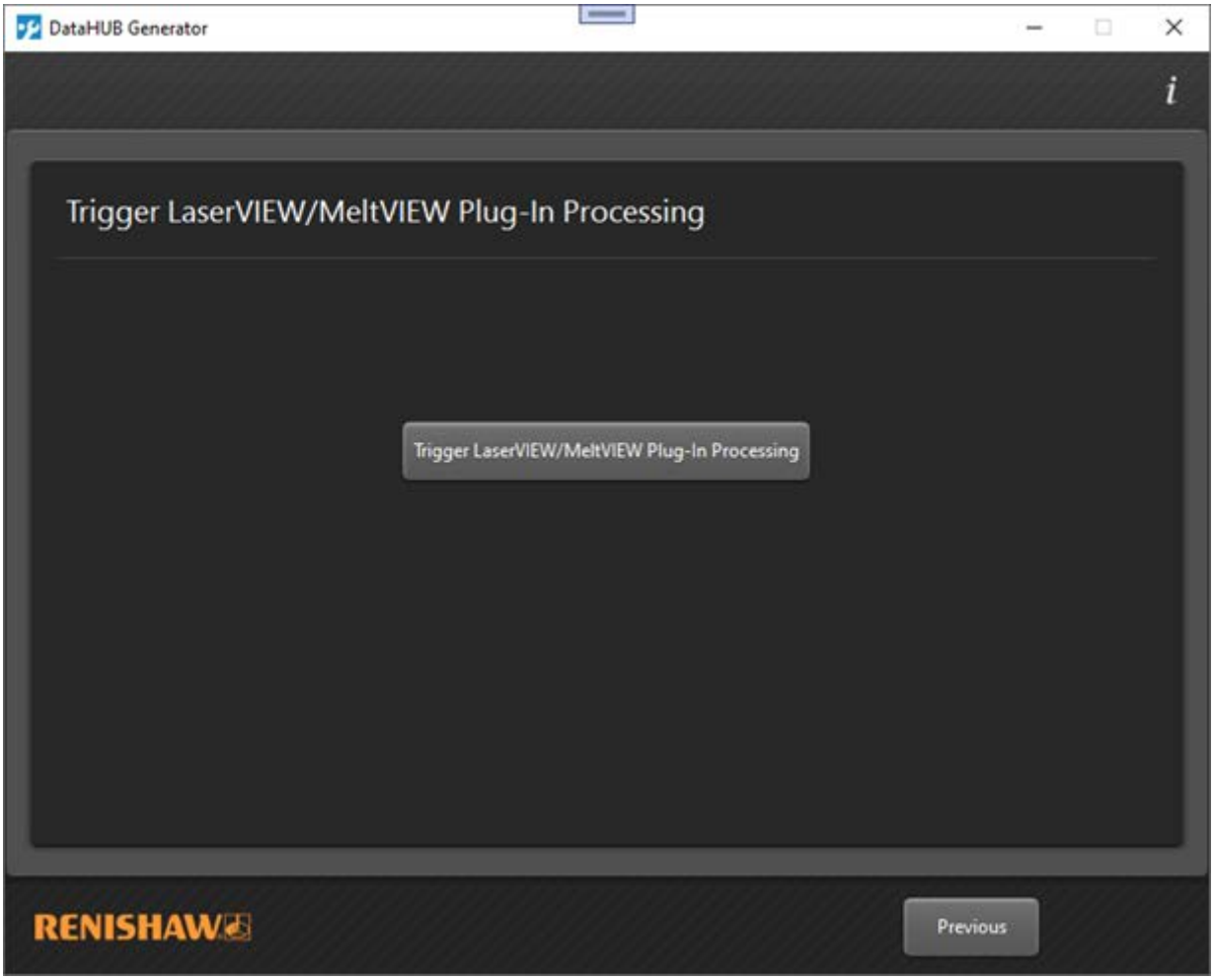


Figure 47 Étape 4 du flux de production : démarrage du traitement

Vous pouvez à présent quitter l'application DataHUB Generator.

6.5.2 Nom de fichier et emplacement

Le dossier *Output* spécifié dans le flux de production constitue le dossier racine utilisé par le plug-in dans lequel il écrit ses sorties. Le plug-in crée un sous-dossier nommé comme suit :

[2] Export Packets aaaaMMjj-HHmmss.ff (GUID)

Où « aaaaMMjj-HHmmss.ff » est la date et l'heure auxquelles le plug-in a commencé à traiter les données, et « GUID » est un code unique. Cette dénomination a été choisie pour s'assurer que la ré-exécution du plug-in sur les mêmes données n'écrase pas les résultats préexistants du plug-in.

Dans le sous-dossier, le plug-in écrira un fichier par laser, par couche, dénommé « Packet data for layer x, laser y.txt », où x est le numéro de la couche et y le numéro du laser. Notez que bien que le fichier utilise l'extension « txt », il se présente sous la forme d'un simple fichier délimité par des tabulations et peut donc être importé dans de nombreux logiciels sans aucun traitement préalable.

6.5.3 Contenu du fichier

Chaque fichier commence par une ligne d'en-têtes de colonnes :

- **Start time** (Heure de début) : correspond au début de la couche (µs).
- **Duration** (Durée) : découle des heures du premier et du dernier échantillon dans le paquet (µs)
- **Demand X** (X de la demande) : position du paquet sur le lit de poudre (de gauche –125,0 à droite +125,0 mm)
- **Demand Y** (Y de la demande) : position du paquet sur le lit de poudre (d'avant –125,0 en arrière +125,0 mm)
- **Demand focus** (Focalisation de la demande) : focalisation du faisceau laser sur le lit de poudre (±5,0 mm)
- **Demand laser power (mean)** (Puissance laser de la demande (moyenne)) : valeurs moyennes des échantillons regroupés dans le paquet
- **MeltVIEW Plasma (mean)** (MeltVIEW Plasma (moyenne)) : valeurs moyennes des échantillons regroupés dans le paquet
- **MeltVIEW Melt Pool (mean)** (MeltVIEW Melt Pool (moyenne)) : valeurs moyennes des échantillons regroupés dans le paquet
- **LaserVIEW (mean)** (LaserVIEW (moyenne)) : valeurs moyennes des échantillons regroupés dans le paquet
- **Laser back reflection (mean)** (Rétro-réflexion laser (moyenne)) : valeurs moyennes des échantillons regroupés dans le paquet
- **Laser output power (mean)** (Puissance de sortie laser) (moyenne) : valeurs moyennes des échantillons regroupés dans le paquet
- **Demand laser power (median)** (Puissance laser de la demande (médiane)) : valeurs médianes des échantillons regroupés dans le paquet
- **MeltVIEW Plasma (median)** (MeltVIEW Plasma (médiane)) : valeurs médianes des échantillons regroupés dans le paquet
- **MeltVIEW Melt Pool (median)** (MeltVIEW Melt Pool (médiane)) : valeurs médianes des échantillons regroupés dans le paquet
- **LaserVIEW (median)** (LaserVIEW (médiane)) : valeurs médianes des échantillons regroupés dans le paquet
- **Laser back reflection (median)** (Rétro-réflexion laser (médiane)) : valeurs médianes des échantillons regroupés dans le paquet
- **Laser output power (median)** (Puissance de sortie laser (médiane)) : valeurs médianes des échantillons regroupés dans le paquet

Suit une série de lignes contenant les valeurs synthétisées pour chaque paquet de données AMPM :

Start time	Duration	Demand X	Demand Y	Demand focus	Demand laser power (mean)	MeltVIEW plasma (mean)	MeltVIEW melt pool (mean)	LaserVIEW (mean)	Laser back reflection (mean)	Laser output power (mean)	Demand laser power (median)	MeltVIEW plasma (median)	MeltVIEW melt pool (median)	LaserVIEW (median)	Laser back reflection (median)	Laser output power (median)
29800	20	-110.365	-76.35	0	2418	59.5	59	167	12801	16216	2418	59.5	59	167	12801	16216
29830	20	-110.36	-76.325	0	2418	631.5	371	696.5	17184	19679.5	2418	631.5	371	696.5	17184	19679.5
29860	20	-110.365	-76.29	0	2418	1017	672	725.5	17709.5	20158.5	2418	1017	672	725.5	17709.5	20158.5
29890	20	-110.365	-76.265	0	2418	902	653.5	729.5	17847.5	20287.5	2418	902	653.5	729.5	17847.5	20287.5
29920	20	-110.37	-76.235	0	2418	962.5	746.5	735	9562	7149	2418	962.5	746.5	735	9562	7149
30970	20	-110.26	-76.14	0	2418	379.5	434	424	15568.5	18501	2418	379.5	434	424	15568.5	18501
31000	20	-110.265	-76.165	0	2418	717.5	432.5	725.5	17634.5	20103	2418	717.5	432.5	725.5	17634.5	20103
31030	20	-110.265	-76.19	0	2418	1022	684	730	17855	20304.5	2418	1022	684	730	17855	20304.5
31060	20	-110.27	-76.22	0	2418	964	725.5	737.5	17891.5	20339.5	2418	964	725.5	737.5	17891.5	20339.5
31090	20	-110.27	-76.245	0	2418	975	756.5	735	17934.5	20400.5	2418	975	756.5	735	17934.5	20400.5
31120	20	-110.27	-76.275	0	2418	1180.5	913.5	739	17960.5	20414	2418	1180.5	913.5	739	17960.5	20414
31150	20	-110.265	-76.3	0	2418	1196	922.5	736	17984	20421	2418	1196	922.5	736	17984	20421
31180	20	-110.265	-76.33	0	2418	1163	888	740.5	17965.5	20420	2418	1163	888	740.5	17965.5	20420
31210	20	-110.265	-76.36	0	2418	1077.5	853	734.5	17973	20416	2418	1077.5	853	734.5	17973	20416
31240	20	-110.26	-76.385	0	2418	1098.5	875.5	743.5	17963.5	20403.5	2418	1098.5	875.5	743.5	17963.5	20403.5
31270	20	-110.265	-76.415	0	2418	1165.5	947	737	17928	20363	2418	1165.5	947	737	17928	20363
31300	20	-110.265	-76.445	0	2418	1086	873.5	741.5	9582	7164	2418	1086	873.5	741.5	9582	7164
32350	20	-110.175	-76.535	0	2418	418	456	428	15556.5	18509	2418	418	456	428	15556.5	18509
32380	20	-110.175	-76.505	0	2418	834	490	728	17640	20111	2418	834	490	728	17640	20111
32410	20	-110.175	-76.475	0	2418	1060	711	742	17858	20319	2418	1060	711	742	17858	20319
32440	20	-110.175	-76.45	0	2418	1125	812	741.5	17918	20383	2418	1125	812	741.5	17918	20383
32470	20	-110.165	-76.42	0	2418	1210	894.5	744.5	17959.5	20409.5	2418	1210	894.5	744.5	17959.5	20409.5
32500	20	-110.17	-76.39	0	2418	1202	860	745	17952	20387	2418	1202	860	745	17952	20387

Cette page est laissée vierge intentionnellement.

RENISHAW
apply innovation™

www.renishaw.fr/contacter

#renishaw

+33 1 64 61 84 84 **france@renishaw.com**

Référence : H-5800-6855-01-A
Édition : 10.2023