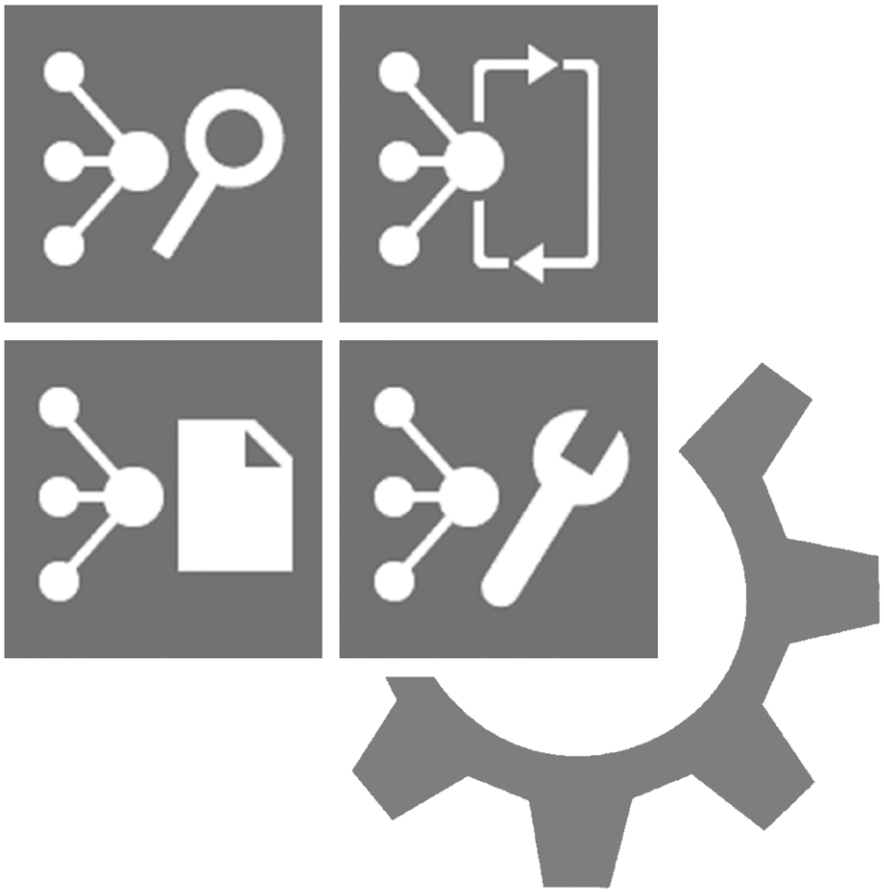
RENISHAW
apply innovation™

Manuale per sviluppatori di DataHUB



www.renishaw.it/additivemanufacturing

#renishaw

Pagina lasciata intenzionalmente vuota.

RENISHAW
apply innovation™

Sommario

1 Prima di iniziare

1.1 Termini, condizioni e garanzie

A meno che non sia stato separatamente concordato e firmato un contratto scritto fra Renishaw e l'utente, le apparecchiature e/o i software venduti sono soggetti ai Termini e alle condizioni standard di Renishaw, forniti insieme all'apparecchiatura e/o al software o disponibili su richiesta presso la sede Renishaw di zona.

Renishaw fornisce una garanzia per le proprie apparecchiature e/o software (secondo quanto riportato nei termini e nelle condizioni standard), purché questi vengano installati e utilizzati con le precise modalità indicate nella documentazione Renishaw associata alle apparecchiature in questione. Per informazioni dettagliate sulla garanzia, leggere i Termini e le condizioni standard.

Le apparecchiature e/o i software acquistati presso fornitori terze parti sono soggetti a termini e condizioni separati, che devono essere forniti insieme all'apparecchiatura o al software. Per maggiori informazioni, contattare il fornitore di terze parti.

1.2 Modifiche all'apparecchiatura

Renishaw si riserva il diritto di apportare modifiche alle specifiche delle apparecchiature senza preavviso.

1.3 Brevetti

Le caratteristiche dei sistemi Renishaw per lavorazioni additive e di altri sistemi simili sono oggetto di uno o più dei seguenti brevetti e/o domande di brevetto

1.3.1 Serie RenAM 500 (modelli Q, S e 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 Introduzione

2.1 Dotazione

Questo documento costituisce una guida per creare, testare e implementare componenti per plugin software da usare con DataHUB. Per una panoramica e una spiegazione delle funzioni del software DataHUB, vedere la guida all'uso di *DataHUB* (codice Renishaw H-5800-6853). Il ciclo di sviluppo di un plugin inizia con l'installazione di DataHUB nel PC di sviluppo. Successivamente, utilizzare Visual Studio per creare un assemblaggio .NET contenente il codice del plugin. Utilizzare una configurazione appropriata di Visual Studio per testare ed eseguire il debug del plugin e verificare che sia corretto prima di installarlo nel PC di raccolta dati (DCPC) di un sistema di produzione AM.

2.2 Abbreviazioni

Termine	Definizione
AM	(Additive Manufacturing) Produzione additiva
AMPM	(Additive Manufacturing Process Monitoring) Monitoraggio del processo di lavorazione additiva
DCPC	(Data Collection Personal Computer) PC di raccolta dati
HMI	(Human Machine Interface) Interfaccia uomo/macchina (touch screen)
OEM	(Original Equipment Manufacturer) Produttore di apparecchiature originali
PC	Personal Computer
PLC	(Programmable Logic Controller) Controllo logico programmabile
RAEE	Rifiuti di apparecchiature elettriche ed elettroniche
REACH	Registrazione, valutazione, autorizzazione e restrizione delle sostanze chimiche

2.3 Informazioni di sicurezza contenute in questa guida

Nella presente Guida all'uso, le informazioni importanti da leggere e comprendere verranno presentate con titoli quali Avviso, Avvertenza o Nota. Di seguito viene fornita una descrizione di tali informazioni, accompagnata da alcuni esempi.

2.3.1 Avviso

Di seguito viene fornito un esempio di avviso:

AVVISO: un avviso informa che il mancato rispetto della procedura descritta potrebbe causare lesioni personali all'utente o ad altre persone nelle vicinanze.

2.3.2 Avvertenza

Di seguito viene fornito un esempio di avvertenza:

AVVERTENZA: le avvertenze indicano all'utente finale che, se non viene seguita la procedura descritta, sussiste il rischio di danni al dispositivo.

2.3.3 Nota

Di seguito viene fornito un esempio di nota:

NOTA: le note forniscono all'utente finale suggerimenti o informazioni importanti correlati all'attività in corso di svolgimento.

2.4 Programma di formazione

Renishaw organizza corsi di formazione di base per imparare a utilizzare DataHUB in sicurezza. Renishaw offre inoltre corsi di formazione avanzati per operatori e tecnici. Vedere la Guida all'uso di *DataHub* (codice Renishaw H-5800-6853) e la guida fornita come parte integrante del corso di formazione che tutti gli utenti devono completare prima di utilizzare il DataHUB.

2.5 Documentazione di riferimento

Oltre alla presente Guida all'uso, vedere anche i seguenti documenti che contengono informazioni aggiuntive su vari aspetti del pacchetto InfiniAM®e sui sistemi AM di Renishaw.

- Guida all'installazione del *sistema RenAM 500Q/S per lavorazioni additive* (codice Renishaw H-5800-4636)
- Guida all'uso del *sistema RenAM 500Q/S per lavorazioni additive* (codice Renishaw H-5800-4637)
- Guida all'installazione del software *InfiniAM® e DataHUB* (codice Renishaw H-5800-6845)
- Guida all'uso di *DataHUB* (codice Renishaw H-5800-6853)
- Guida all'uso di *InfiniAM® Spectral* (codice Renishaw H-5800-6841)
- Guida all'uso di *InfiniAM® Camera* (codice Renishaw H-5800-6849)
- Guida all'uso di Renishaw Licence Manager v2.x (scaricabile tramite il link incluso nella email contenente la licenza InfiniAM)
- Plugin Export Packets (codice Renishaw 5800-6788)

3 Parti di ricambio

All'interno di DataHUB non vi sono componenti che possano essere riparati dell'utente. In caso di guasti a DataHUB, sarà necessario reinstallare e riconfigurare il software.

Per informazioni su come contattare l'ufficio Renishaw di zona e prenotare una visita da parte del tecnico, vedere la sezione 4, "Dettagli per i contatti".

I software InfiniAM e DataHUB vengono aggiornati periodicamente. Tutti gli utenti registrati possono scaricare le release più recenti dal proprio account MyRenishaw.com.

Pagina lasciata intenzionalmente vuota.

4 Dettagli per i contatti

Numero di telefono	+39 011 966 67 00	
Orario:	Dal lunedì al giovedì	08:00 – 17:00 (UTC e DST)
	Venerdì	08:00 – 16:00 (UTC e DST)
Email	am.support@renishaw.com	
Indirizzo dell'assistenza	Renishaw S.p.A. Via dei Prati 5, 10044 Pianezza Torino Italia	
Tipo di sistema AM		
Numero di serie del sistema AM		
Numeri della versione del software	Revisione HMI	
	Revisione PLC	
	Revisione PC	
Numero di serie dell'hardware InfiniAM Spectral (modulo MeltVIEW)		
Numero della versione del software InfiniAM		
Numero della versione del software DataHUB		

Indicare i dettagli forniti sopra. I dati del sistema AM Renishaw sono riportati sulla piastrina posta sul retro del sistema stesso. I dati del modulo MeltVIEW sono riportati sulla piastrina visibile quando InfiniAM viene installato nel sistema AM Renishaw. I dati dell'hardware CameraVIEW sono riportati sull'adesivo applicato sul retro della fotocamera. L'hardware CameraVIEW si trova dietro una protezione, al di sopra della camera.

È possibile ottenere ulteriore assistenza contattando l'ufficio Renishaw di zona. Vedere:

www.renishaw.it/contatti

Pagina lasciata intenzionalmente vuota.

5 Sicurezza

5.1 Introduzione

AVVERTENZA: tutte le informazioni di sicurezza sono in conformità alla guida all'uso applicabile per il sistema Renishaw AM, a meno che non venga indicato diversamente in questo documento.

AVVERTENZA: prima di accendere il laser del sistema AM, controllare che sull'apertura sia stata posizionata una piastra di protezione o il modulo hardware MeltVIEW.

AVVERTENZA: il modulo hardware MeltVIEW non è collegato al circuito di sicurezza laser. Se il laser dovesse essere attivato dopo che il modulo hardware MeltVIEW è stato rimosso, la luce emessa dall'apertura del sistema AM potrebbe essere

5.2 Etichette di avviso specifiche per il laser DataHUB Generator

Non vi sono etichette di sicurezza o di avviso aggiuntive da applicare al sistema AM dopo l'installazione di DataHUB Generator.

Pagina lasciata intenzionalmente vuota.

6 Funzionamento

6.1 Introduzione

Grazie alla sua architettura di tipo plugin, il pacchetto software InfiniAM è in grado di supportare l'analisi in tempo reale dei dati di monitoraggio del processo raccolti dalle piattaforme hardware LaserVIEW e MeltVIEW. Un plugin è un software autonomo da installare in un PC di raccolta dati (DCPC) in cui viene eseguito DataHUB. Quando DataHUB elabora i dati raccolti di una costruzione AM, passa i valori del monitoraggio del processo ai plugin che si occupano di svolgere analisi mirate. DataHUB supporta l'esecuzione simultanea di più plugin e consente l'applicazione di diversi algoritmi e tecniche di analisi ai dati di monitoraggio del processo. Inoltre è integrato con il sistema Renishaw Central e consente ai plugin di presentare i risultati delle analisi insieme alle letture ricavate da altri sensori della macchina durante la costruzione.

DataHUB supporta plugin con due versioni API: la versione 1.0 (V1.0) supporta i plugin per elaborare i dati di costruzione riuniti in pacchetti (vedere "Quantizzazione dei campioni in pacchetti" nella pagina 6-8); la versione 2 (V2.0) supporta i plugin che elaborano i dati di costruzione come flussi di token (vedere "Inclusione dei campioni in un flusso di token" nella pagina 6-8). È importante capire che l'API V2.0 non sostituisce la versione 1.0, ma presenta gli stessi dati di monitoraggio del processo con un formato diverso. È importante scegliere con attenzione l'API da utilizzare in base al tipo di analisi da svolgere: V1.0 fornisce un numero di pacchetti decisamente inferiore rispetto ai campioni prodotti da V2.0 potrebbe risultare più indicata quando non sono necessari dati molto dettagliati.

In questo documento vengono spiegati l'architettura a plugin del sistema, il ciclo vita di un plugin e altri aspetti del sistema applicabili a entrambe le versioni API.

Il plugin Export Packets scrive un file con tabulazioni che contiene un riepilogo dei dati di monitoraggio del processo raccolti da LaserVIEW e MeltVIEW durante la costruzione. Questo plugin viene distribuito insieme a DataHUB e non richiede licenze aggiuntive.

6.2 Architettura

Le relazioni di alto livello fra gli elementi hardware e software del sistema vengono descritti in Figura 1:

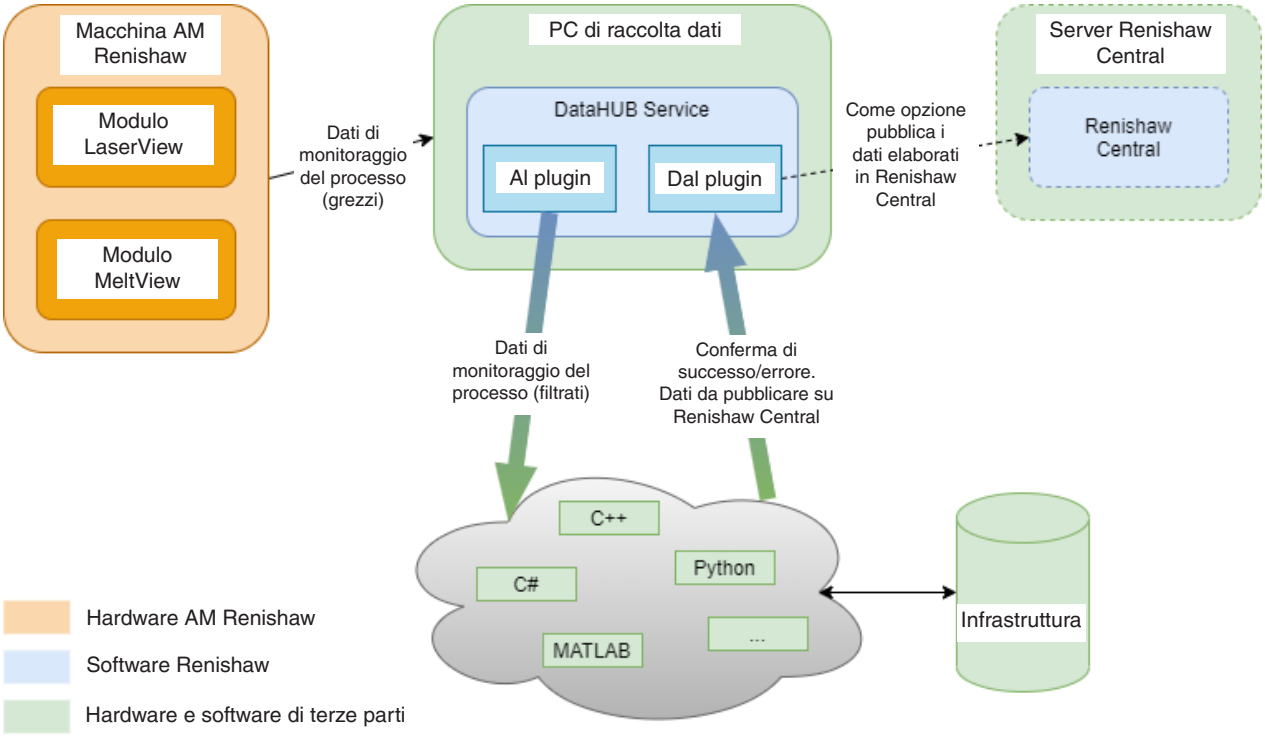


Figura 1 Architettura a plugin di DataHUB

6.2.1 Flusso di dati

Quando una macchina AM Renishaw con hardware LaserVIEW e MeltVIEW esegue una costruzione, i dati di monitoraggio del processo vengono trasferiti a un DCPC. Il servizio DataHUB in esecuzione nel DCPC indirizza i dati a tutti i plugin installati. Per ogni costruzione, DataHUB esegue un'istanza di tutti i plugin installati. A ciascuna istanza viene assegnata una cartella di output con un nome univoco e ognuna viene eseguita in modo indipendente, in modo che un errore in un'istanza non causi un errore generale di tutto il sistema.

Figura 2 mostra i dati di un flusso di dati tipico durante l'elaborazione di una costruzione:

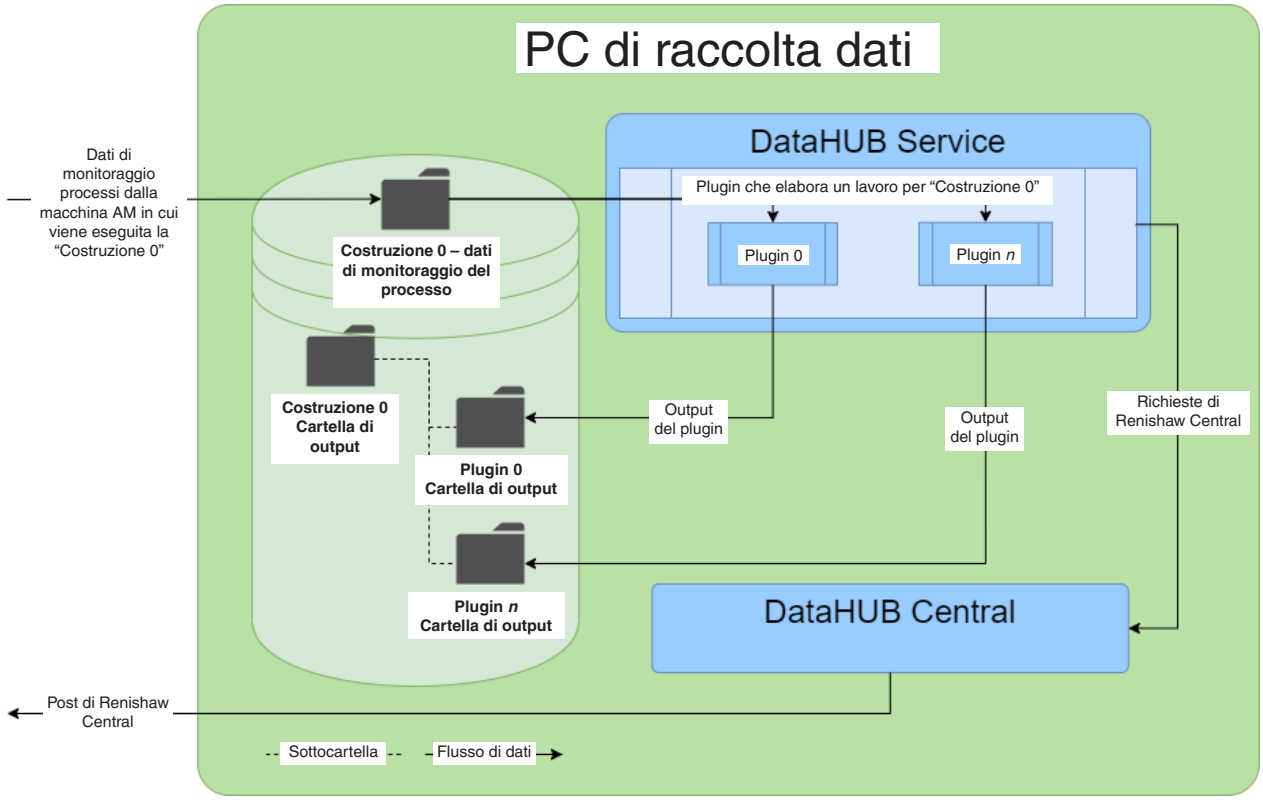


Figura 2 Flussi di dati della costruzione

I dati di monitoraggio del processo acquisiti da una macchina AM che esegue una costruzione vengono inviati a una cartella del DCPC. Il servizio DataHUB monitora la cartella e, man mano che arrivano nuovi dati, li indirizza alle istanze dei plugin associate alla costruzione. I plugin possono segnalare al servizio DataHUB i risultati (dati di serie temporali, avvisi e risultati delle analisi) da pubblicare in Renishaw Central. Il servizio DataHUB Central gestisce le richieste di upload.

6.2.2 Tecnologia di analisi dei plugin

DataHUB è in larga parte agnostico per quanto riguarda la tecnologia usata per implementare i plugin e le infrastrutture di supporto. Gli unici due requisiti sono:

1. Un plugin è un assemblaggio .NET che implementa la serie di metodi definiti come API del plugin.

2. Tutte le dipendenze di runtime vengono installate nel DCPC.

DataHUB richiama i metodi API dei plugin popolando i valori con i dati di monitoraggio del processo e l'implementazione dei plugin è in grado di trasferire i dati a qualsiasi tecnologia per svolgere le analisi necessarie. Una spiegazione dettagliata su come integrare .NET con altre tecnologie e linguaggi di programmazione non rientra negli scopi di questo documento, ma è comunque pubblicamente disponibile una vasta quantità di informazioni.

6.2.3 Flusso di dati "Push"

Il servizio DataHUB "spinge" i dati nei plugin: le API dei plugin non servono per interrogare serie di dati e estrarre valori dai dati. Questo flusso di dati è ottimo per supportare le analisi in tempo reale mentre la costruzione è in corso. Nel caso di serie di dati archiviate, è possibile ricorrere a DataHUB Generator per fornire al servizio DataHUB l'istruzione di rielaborare i dati e rendere possibile l'ulteriore analisi dei dati.

6.2.4 Ciclo vita del plugin

Per essere utilizzato da DataHUB, il plugin deve essere prima convalidato rispetto a tutti i requisiti di implementazione dell'API. Solo se il plugin risulta conforme le istanze verranno create e utilizzate per l'analisi di dati della costruzione. Dopo la convalida, verrà creata un'istanza del plugin per ciascuna nuova attività di elaborazione dei dati della costruzione. Le fasi del ciclo vita di un plugin sono riportate in Figura 3:

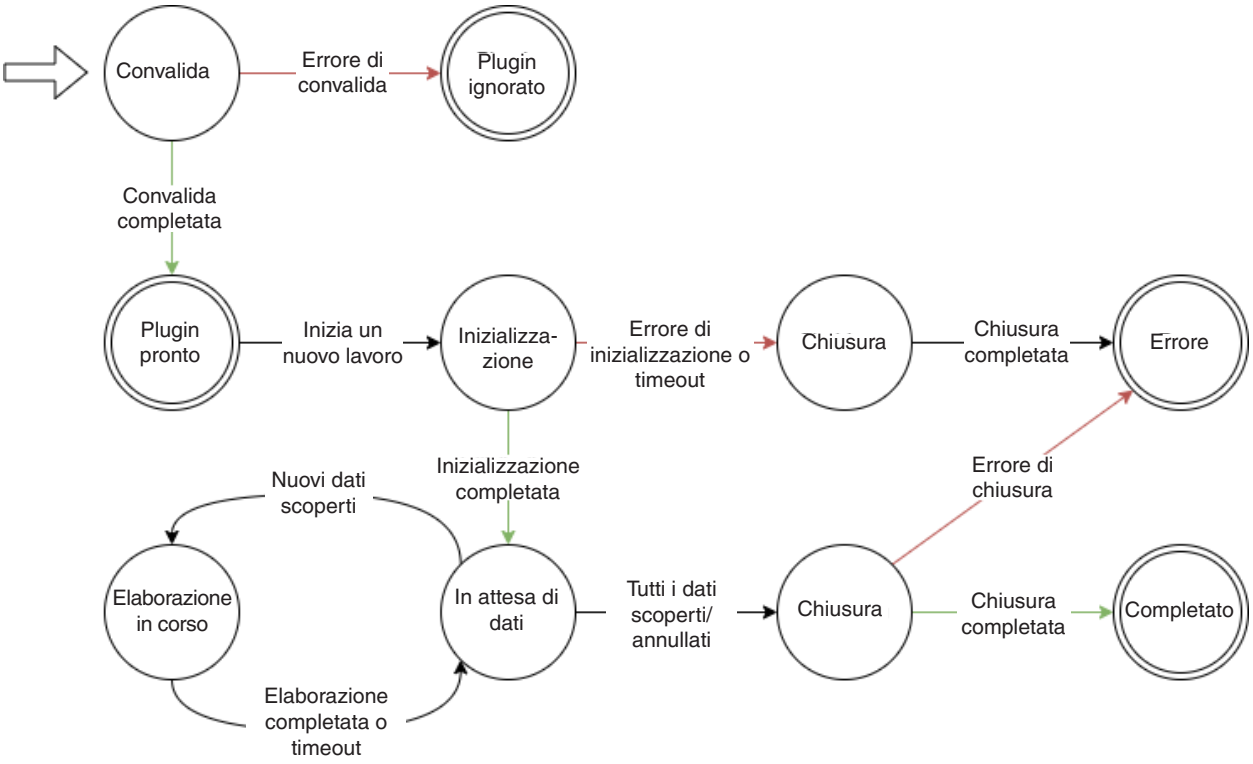


Figura 3 I vari stati in cui può trovarsi un plugin durante il suo ciclo vita

6.2.4.1 Convalida

All'avvio, DataHUB cerca gli assemblaggi .NET che implementano una o più classi pubbliche conformi alle API dei plugin di DataHUB. Tutti gli elementi conformi vengono convalidati e registrati per essere istanziati ogni volta che si avvia un'operazione di elaborazione di LaserVIEW e MeltVIEW tramite i plugin.

Se lo si desidera, un assemblaggio può contenere implementazioni di più plugin che utilizzano versioni diverse dell'API.

6.2.4.2 Inizializzazione

Quando DataHUB avvia un'operazione tramite plugin, viene creata una nuova istanza del plugin, richiamando il suo metodo *Initialise*. I valori passati a questo metodo sono "costanti di costruzione", come ad esempio il numero di laser tramite cui il plugin può calcolare i requisiti per l'allocazione delle risorse e altro ancora. Il metodo indica se l'inizializzazione è andata a buon fine o se si è verificato un errore. In questo secondo caso, viene immediatamente richiamato il metodo *Shutdown*. Come opzione, il plugin può restituire definizioni per gli assi delle serie temporali che saranno popolati con punti dati generati durante le elaborazioni successive (vedere "Pubblicazione dei dati in Renishaw Central" nella pagina 6-6).

6.2.4.3 In attesa di dati/Elaborazione in corso

Il metodo di elaborazione dati viene richiamato tutte le volte necessarie per trasferire tutti i dati di monitoraggio del processo di una costruzione all'istanza del plugin. Ad ogni chiamata, il plugin indica se l'elaborazione dati è andata a buon fine o se si è verificato un errore. Nel secondo caso, il plugin non entra in uno stato di errore e continua a trasferire dati (si presume che i dati degli altri strati siano comunque interessanti per il plugin). Come opzione, il plugin può restituire punti dati di serie temporali, avvisi e risultati delle analisi da pubblicare in Renishaw Central (vedere "Pubblicazione dei dati in Renishaw Central").

6.2.4.4 Shutdown

Il metodo *Shutdown* viene richiamato quando non vi sono più dati da elaborare (ad esempio, quando la costruzione è completa o se viene annullata). In questo metodo il plugin deve eseguire le elaborazioni finali dei dati di costruzione, rilasciare tutte le risorse acquisite e così via.

6.2.5 Tempo

Nell'API il tempo viene comunicato in microsecondi, in relazione all'inizio di uno strato. Un tempo 0 corrisponde all'inizio di uno strato, un tempo 1000 indica 1000 microsecondi dopo l'inizio dello strato e così via. Quando necessario (ad esempio, quando si pubblica su Renishaw Central), DataHUB converte i tempi relativi restituiti da un plugin in intervalli di tempo assoluti (ad esempio UTC).

6.2.6 Pubblicazione dei dati in Renishaw Central

Se DataHUB è connesso a Renishaw Central, un plugin può pubblicare i risultati dell'analisi tramite DataHUB. Se la stringa restituita dai metodi *Initialise* e Process è conforme allo schema JSON registrato nelle definizioni API (vedere la sezione 6.4, "API per plugin V2"), DataHUB pubblica i dati contenuti nell'endpoint di Renishaw Central.

DataHUB riconosce tre tipi di upload su Renishaw Central:

- Avvisi
- Risultati dell'analisi
- Serie temporali

6.2.6.1 Avvisi

Un avviso indica che un evento importante si è verificato in un dato momento. Può essere generato da un plugin quando rileva un'anomalia nei dati di costruzione (ad esempio, se i dati suggeriscono che un errore si è verificato o sta per verificarsi nella costruzione). Sono previsti tre livelli di avvisi – *Informazioni*, *Avvertenze* ed *Errori* – che descrivono all'applicazione di visualizzazione (ad esempio il sito Web di Renishaw Central) il livello di gravità dell'avviso.

Gli avvisi possono essere generati solo durante la fase *Process* del ciclo vita.

6.2.6.2 Risultati dell'analisi

Come gli avvisi, anche i risultati dell'analisi possono essere restituiti in relazione all'analisi svolta sui dati di costruzione. Hanno una definizione flessibile e il loro contenuto può essere specializzato in base al tipo di risultato. Questo significa che, mentre (ad esempio) il sito Web di Renishaw Central può non sapere in che modo rappresentare ogni aspetto del risultato dell'analisi, è possibile scrivere un'applicazione su misura da posizionare a valle.

I risultati dell'analisi possono essere generati solo durante la fase *Process* del ciclo vita.

6.2.6.3 Serie temporali

Una serie temporale rappresenta una serie di valori nel tempo. Una serie temporale può contenere tutti i tipi di dati, purché siano tracciabili nel tempo. Durante la fase *Process*, un plugin può aggiungere due tipi di dati a una serie temporale: valori e limiti. I valori sono serie di coppie tempo/valore che aggiungono un punto dati alla serie. I limiti definiscono gli intervalli previsti per agevolare la visualizzazione e mostrare le estremità dei dati.

Le serie temporali si differenziano da avvisi e risultati dell'analisi nel fatto che devono essere definite prima di aggiungere i punti dati. Un plugin può restituire le definizioni dal proprio metodo *Initialise*. Se il plugin dovesse successivamente tentare di aggiungere dati a una serie temporale che non si trovava in questo elenco di definizione iniziale (o se le definizioni stesse dovessero in qualche modo risultare non valide), i dati saranno rifiutati e andranno persi.

6.2.7 Registrazione delle verifiche

DataHUB conserva una serie di registri contenenti le azioni svolte, incluse alcune che possono essere usate per diagnosticare eventuali problemi con un plugin. I registri si trovano in questa posizione:

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

Per il nome dei registri si utilizza la data di creazione, con il formato "aaaMMgg".

Le voci che possono interessare uno sviluppatore di plugin sono:

- All'avvio, DataHUB registra:
  - numeri di versione
  - dati delle licenze
  - convalida degli assemblaggi di plugin

- All'avvio di una costruzione, DataHUB registra:
  - creazione dell'istanza del plugin
  - i valori usati per inizializzare ciascuna istanza del plugin
  - il risultato restituito per il metodo *Initialise*

- Durante una costruzione, DataHUB registra:
  - risultati utili dalle chiamate *Processing* del plugin (un plugin può produrre risultati superflui che non vengono registrati per ridurre lo spam)

- Al termine di una costruzione, DataHUB registra:
  - il risultato di tutte le chiamate per il metodo *Shutdown* del plugin

6.2.8 Pre-elaborazione dei dati di monitoraggio del processo

I moduli LaserVIEW e MeltVIEW raccolgono i dati come un flusso di campioni, ognuno dei quali è associato a uno specifico laser, a una posizione sul letto di polvere e a diverse letture dei sensori, in un determinato momento. DataHUB esegue una pre-elaborazione dei dati di monitoraggio del processo (grezzi). Per i plugin V1.0, DataHUB riepiloga i campioni in pacchetti, mentre per V2.0 i dati campione vengono assemblati in un flusso di token.

6.2.8.1 Quantizzazione dei campioni in pacchetti

Il servizio DataHUB quantizza i campioni consecutivi che condividono la stessa posizione, con il laser acceso, in un pacchetto. Il tempo di inizio di un pacchetto viene definito partendo dal primo campione incluso, mentre l'ultimo campione inserito serve a calcolarne la durata. Si calcola quindi una media dei valori delle letture dei sensori LaserVIEW e MeltVIEW per i campioni inclusi.

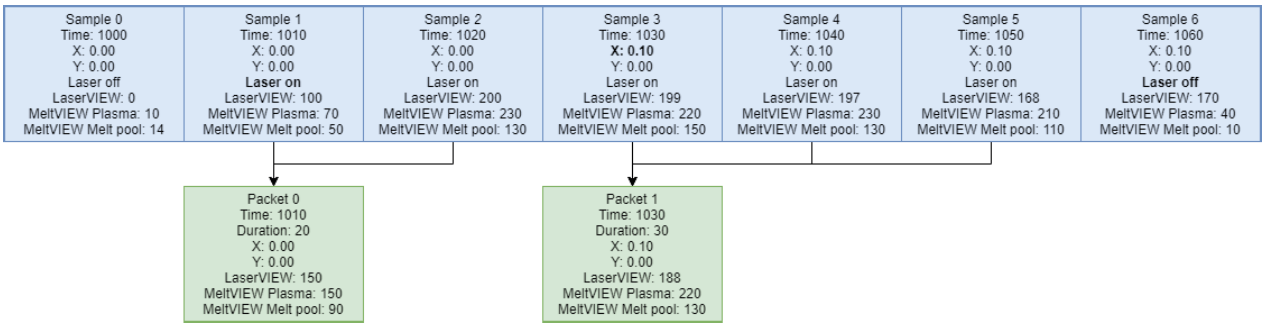


Figura 4 Quantizzazione dei campioni in pacchetti

In Figura 4, sette campioni vengono quantizzati in due pacchetti. Il primo campione viene ignorato perché il laser non è acceso. Dato che condividono la stessa posizione e che il laser è acceso, i campioni 1 e 2 son quantizzati in un pacchetto. La posizione del campione 3 cambia rispetto al campione precedente. Per questo motivo viene iniziato un nuovo pacchetto in cui vengono inseriti i campioni fino a quando il laser non si spegne. Il campione 6 condivide la stessa posizione del campione 5, ma non viene aggiunto, perché il laser è spento.

6.2.8.2 Inclusione dei campioni in un flusso di token

Quando si convertono i campioni di monitoraggio del processo in un flusso di token, DataHUB crea una serie di token di *campioni* circondati da altri token di informazioni che descrivono la struttura e il contenuto della costruzione. A differenza di V1.0, anche i campioni acquisiti mentre il laser era spento vengono presi in considerazione e aggiunti al flusso di token.

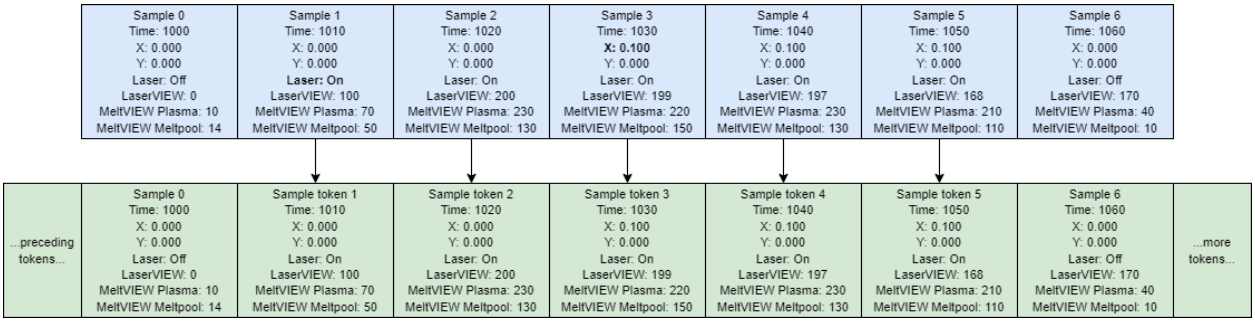


Figura 5 Inclusione dei campioni in un flusso di token

Figura 5 mostra una parte di un flusso di token, con gli stessi campioni usati precedentemente in *Quantizzazione dei campioni in pacchetti*.

6.3 API per plugin V1

Questo documento costituisce una guida per creare, testare e implementare un componente software (V1.0) per plugin da usare con DataHUB. Il ciclo di sviluppo di un plugin inizia con l'installazione di DataHUB nel PC di sviluppo. Successivamente, utilizzare Visual Studio per creare un assemblaggio .NET contenente il codice del plugin. Utilizzare una configurazione appropriata di Visual Studio per testare ed eseguire il debug del plugin e verificare che sia corretto prima di installarlo nel PC di raccolta dati (DCPC) di un sistema di produzione AM.

6.3.1 Attività necessarie

Oltre al presente documento, leggere quanto segue:

- Guida all'installazione del software *InfiniAM® e DataHUB* (codice Renishaw H-5800-6845)
- Guida all'uso di *DataHUB* (codice Renishaw H-5800-6853)
- Guida all'uso di Renishaw Licence Manager v2.x

Prima di continuare, svolgere le seguenti attività nel PC da utilizzare per lo sviluppo del plugin:

- Assicurarsi che il PC disponga di una GPU con le specifiche minime per DataHUB (vedere le specifiche hardware del PC di raccolta dati nella guida all'installazione dei *software InfiniAM® e DataHUB*).
- Verificare che siano stati installati i driver più aggiornati per la GPU.
- Accertarsi che il server delle licenze disponga di un numero sufficiente di licenze appropriate per DataHUB e Spectral API.
- Controllare che dalla rete sia possibile accedere al server delle licenze.
- Verificare che sia installata una versione di Microsoft Visual Studio in grado di supportare .NET Framework 4.7.2.

6.3.2 Installazione di DataHUB per lo sviluppo dei plugin

Il servizio DataHUB deve essere installato nel PC di sviluppo. Durante l'installazione, viene offerta la possibilità di scegliere le licenze da richiedere. Assicurarsi che l'opzione "LaserVIEW/MeltVIEW Pluggability" (Connettibilità a LaserVIEW/MeltVIEW) sia selezionata:

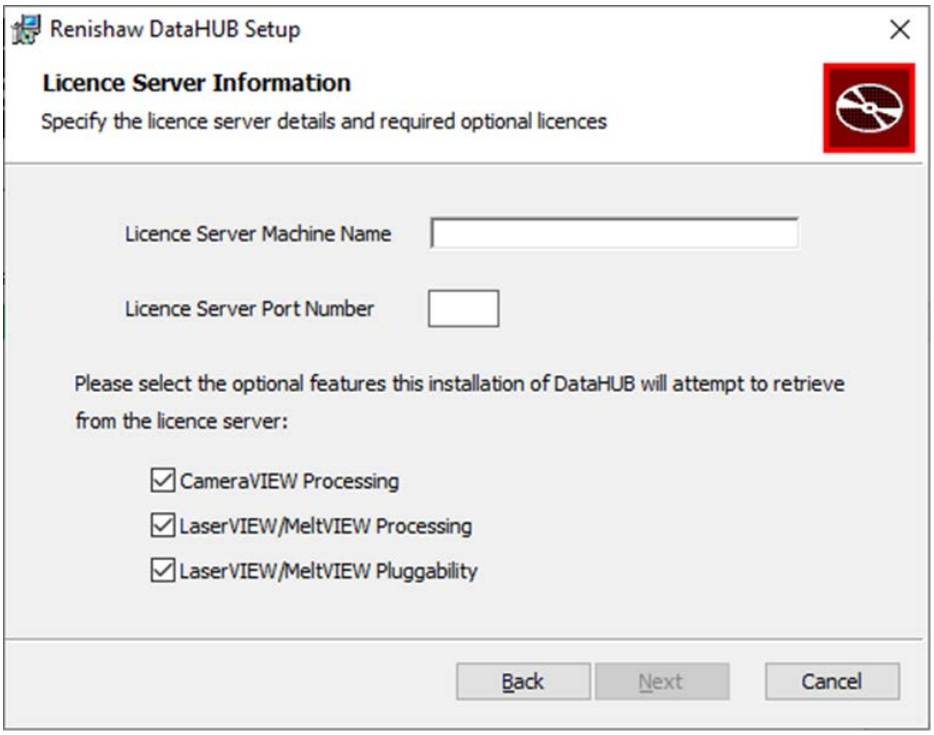


Figura 6 Licenze DataHUB

6.3.3 Implementazione del plugin

Le successive sezioni di questo documento descrivono il flusso di lavoro end-to-end per la creazione, compilazione, debugging e messa in funzione di un plugin con Visual Studio.

I plugin sono scritti in C# e definiscono una classe pubblica per implementare la specifica serie di metodi pubblici usati da DataHUB per identificare, convalidare e utilizzare le istanze del plugin durante l'elaborazione dei dati di monitoraggio del processo di una costruzione.

NOTA: le immagini riprodotte di seguito sono state prese da Microsoft Visual Studio 2019, ma il flusso di lavoro è molto simile anche con le altre versioni.

6.3.3.1 Creazione della soluzione Visual Studio

Avviare Visual Studio e selezionare *Crea nuovo progetto*. Fra le opzioni disponibili, selezionare *Libreria classi (.NET Framework)*:

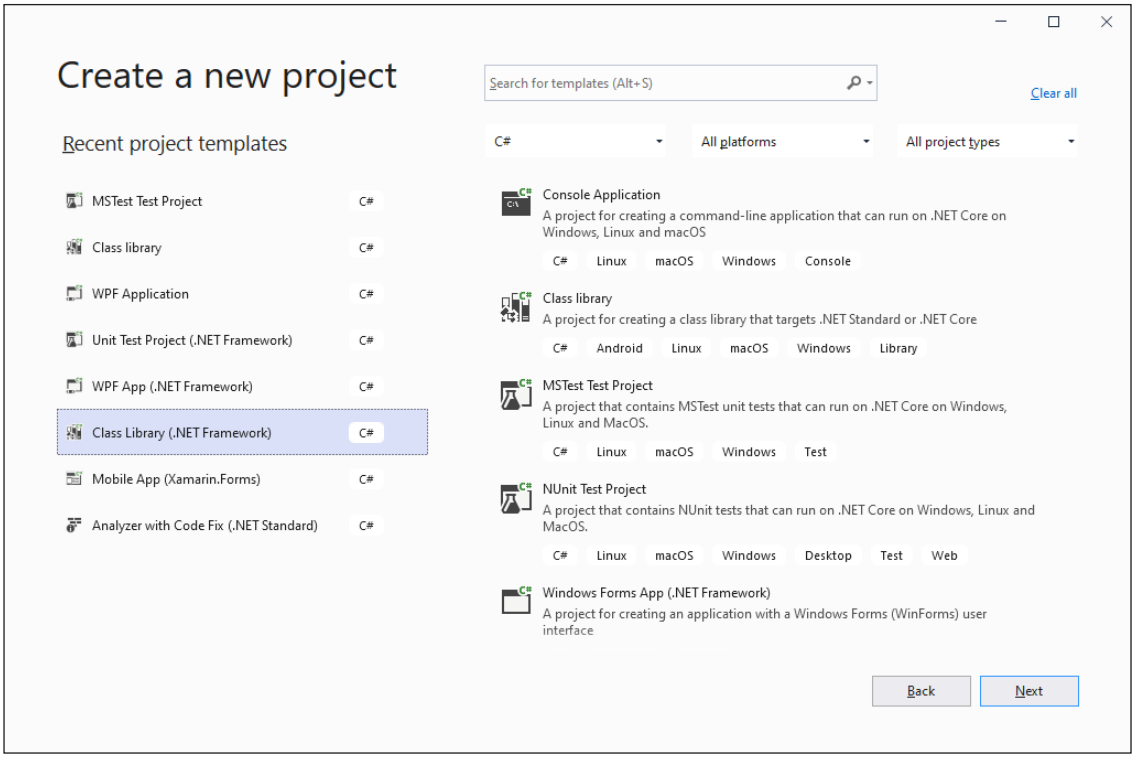


Figura 7 Creazione della soluzione

Configurare il progetto:

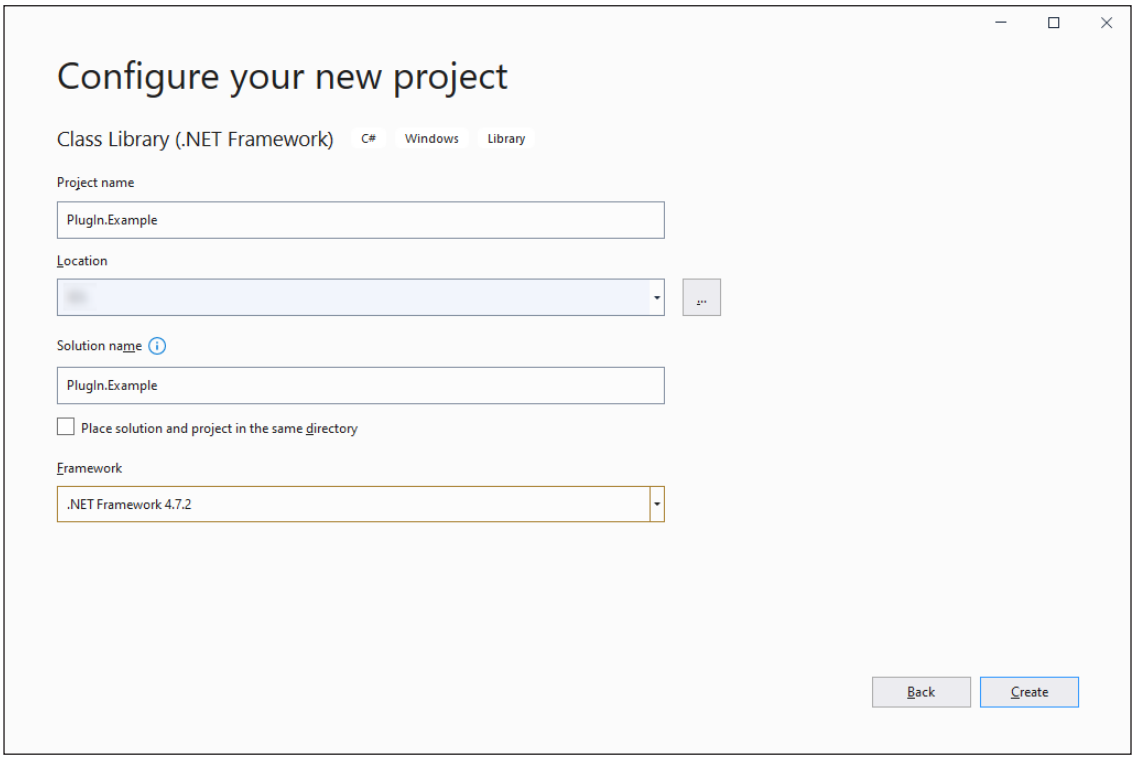


Figura 8 Configurazione del progetto

DataHUB richiede un assemblaggio plugin che inizi con **PlugIn.** (La parola “PlugIn” seguita da un punto). Selezionare una cartella a scelta come *posizione* e creare la soluzione.

NOTA: DataHUB supporta i plugin per x86/x64/Any CPU. Si consiglia di indicare Framework 4.7.2 o superiore.

6.3.3.2 Assegnare un nome al tipo di plugin

Il singolo file sorgente Class1.cs contiene la definizione di `Class1` che verrà usata come punto iniziale.

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

Quando DataHUB rileva un potenziale assemblaggio plugin, cerca un `PlugIn` di tipo pubblico. Rinominare `Class1` in `PlugIn`.

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

6.3.3.3 Aggiunta dell'API

Con un plugin di tipo potenziale, DataHUB richiede l'implementazione dei seguenti metodi pubblici:

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

Aggiungere le seguenti implementazioni di metodo alla 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() => "";
}
```

Si noti che tutti i metodi API restituiscono una `string` (che per il momento è stata lasciata vuota). Il contenuto della stringa restituita viene utilizzato per segnalare a DataHUB il risultato del metodo. Il ruolo di questo valore restituito verrà spiegato man mano che ciascun metodo viene definito.

6.3.3.4 Metodo APIVersion

Questo metodo indica a DataHUB la versione dell'API usata dal plugin. Per usare l'API versione 1.0, deve restituire il valore "1.0":

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

6.3.3.5 Metodo DisplayName

Il contenuto della stringa restituita da questo metodo viene utilizzata come nome del plugin visualizzato in DataHUB Monitor e durante la creazione della cartella di output del plugin. Anche se non è necessario, l'uso di un nome univoco aiuta a identificare il plugin quando, ad esempio, vi sono più versioni installate in un DCPC.

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

6.3.3.6 Metodo Initialise

Questo metodo viene richiamato da DataHUB all'inizio di una costruzione, prima che i dati sugli strati siano inviati al plugin e passa dati *invariabili sulla costruzione*. Nel plugin di esempio il metodo *Initialise* assegna il percorso della cartella di output dell'istanza al campo **_outputFolderPath**:

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

6.3.3.7 Metodo ProcessPackets

Questo metodo viene richiamato da DataHUB quando tutti i dati di uno strato sono stati ricevuti. Nel plugin di esempio il metodo *ProcessPackets* crea prima una riga di intestazioni di colonna separate con delimitatori, quindi aggiunge valori riga per riga ai dati del pacchetto di monitoraggio del processo:

```
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 Metodo Shutdown

Questo metodo viene richiamato da DataHUB dopo che tutti i dati di una costruzione sono stati inviati al plugin o in caso di errori con tutte le precedenti chiamate al plugin. Il metodo non ha parametri.

In questo metodo il plugin deve eseguire le elaborazioni finali dei dati di costruzione, rilasciare tutte le risorse che potrebbe avere acquisito e così via. Nel plugin di esempio, il metodo *Shutdown* è:

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

6.3.4 Distribuzione del plugin per il debug

Creare la soluzione in *Debug* e aprire un browser per trovare la cartella di output che conterrà un file *.dll* con il nome assegnato al progetto (ad esempio, *PlugIn.Example.dll*). Potrebbero essere presenti anche altri file, come ad esempio simboli di debug o altro.

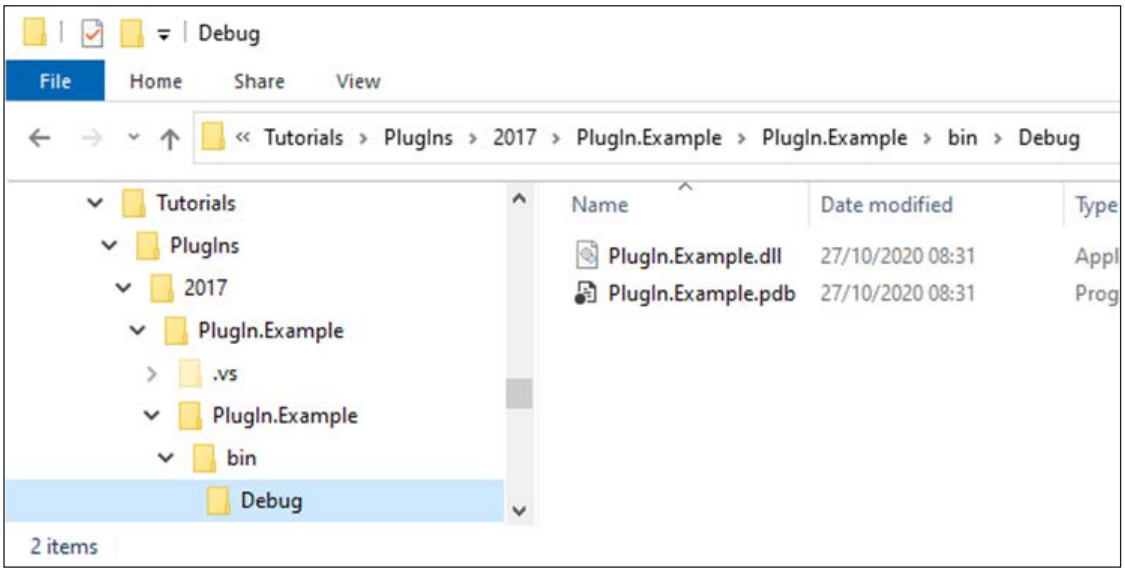


Figura 9 File binari del plugin

Copiare questa cartella e incollarla nella sottocartella *PlugIns* della cartella dati di DataHUB:

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

NOTA: in genere, $PROGRAMDATA$ si trova in <C:\ProgramData>, ma potrebbe essere nascosta. In questo caso, selezionare l'opzione di Esplora risorse di Windows "Visualizza - Mostra/Nascondi - Elementi nascosti".

Per modificare il contenuto di questa cartella è necessario disporre dei diritti di amministratore.

Rinominare la cartella *Debug* con un nome adeguato al plugin. Nell'immagine di seguito, alla cartella è stato assegnato il nome *PlugIn.Example*.

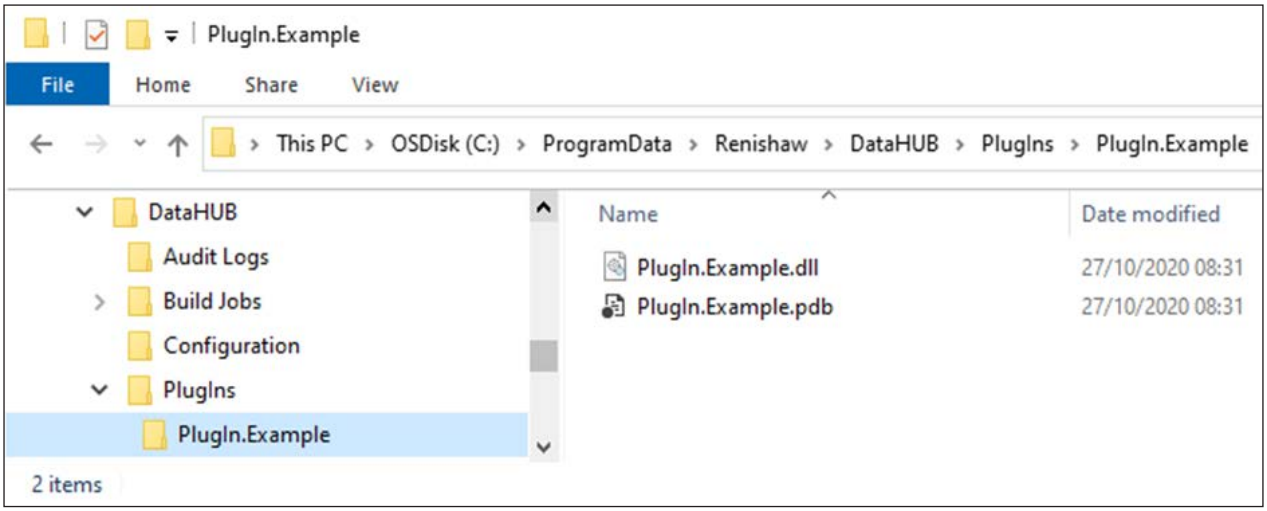


Figura 10 Il plugin dopo la distribuzione

6.3.4.1 Registrazione del plugin

È necessario riavviare il servizio DataHUB per registrare il nuovo plugin. A tale scopo, utilizzare l'applicazione *Servizi*. Fare clic con il pulsante destro del mouse su *DataHUB Service* e selezionare *Riavvia*:

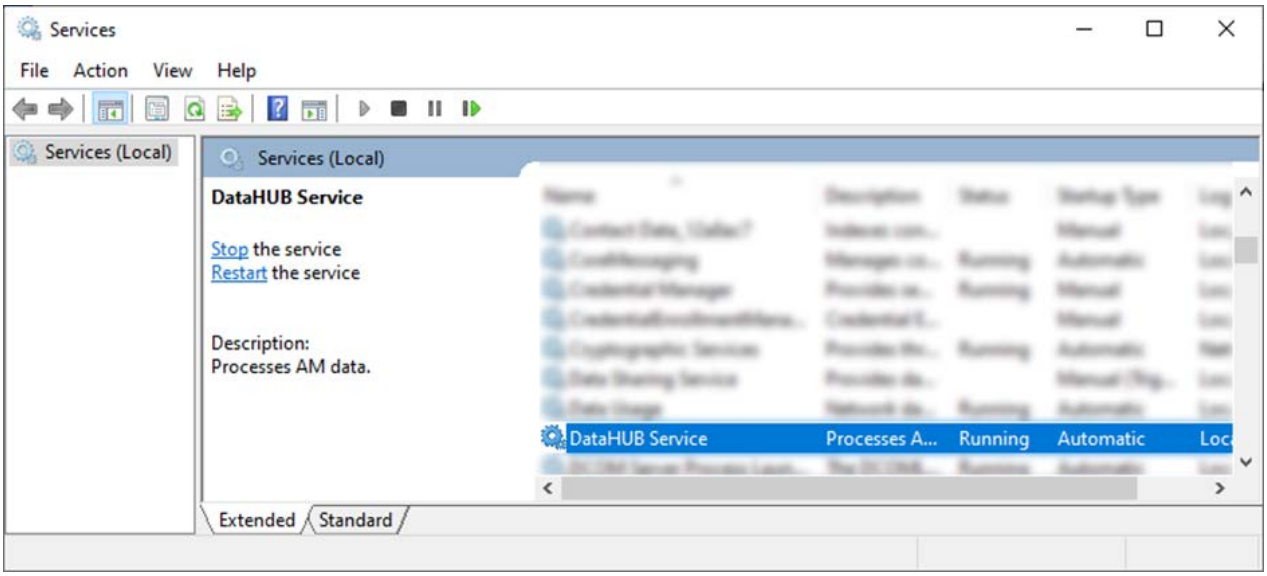


Figura 11 L'applicazione Servizi di Windows

Dopo alcuni secondi, lo stato del servizio DataHUB diventerà *In esecuzione*.

6.3.4.2 Verifica della registrazione

Durante l'esecuzione, DataHUB genera un file di registro in cui inserisce gli eventi significativi, come l'inizio della costruzione, gli errori e, ovviamente, la verifica dei plugin. Con Esplora risorse, trovare la cartella <$PROGRAMDATA$\Renishaw\DataHUB\Audit Logs>.

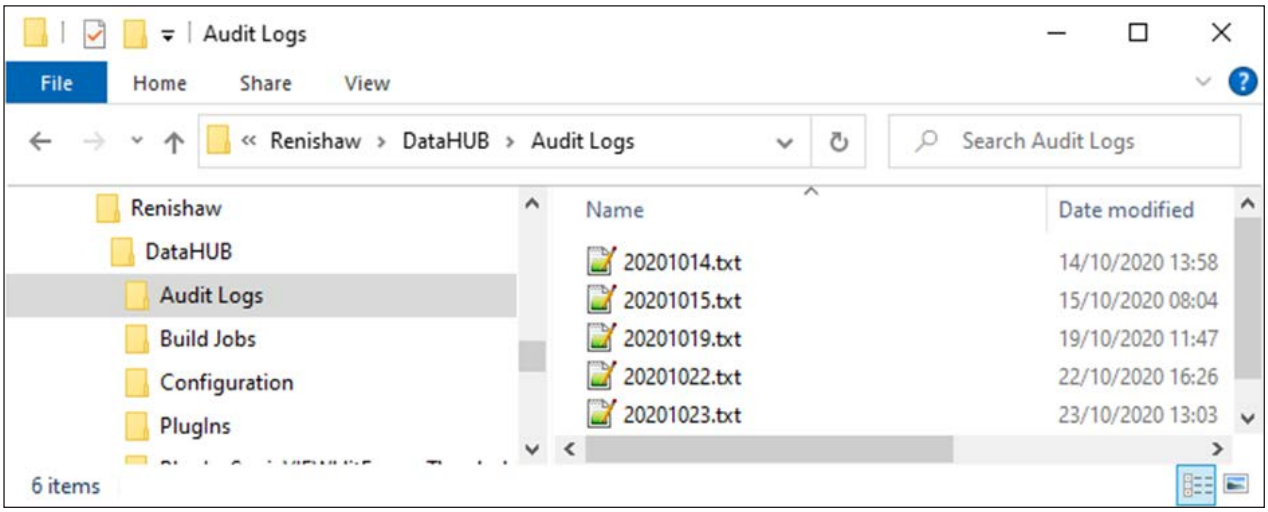


Figura 12 Cartella DataHUB con i file di registro

Aprire il file di registro più recente (dovrebbe riportare la data odierna), scorrere fino alla fine e cercare le righe che riportano il rilevamento di plugin validi:

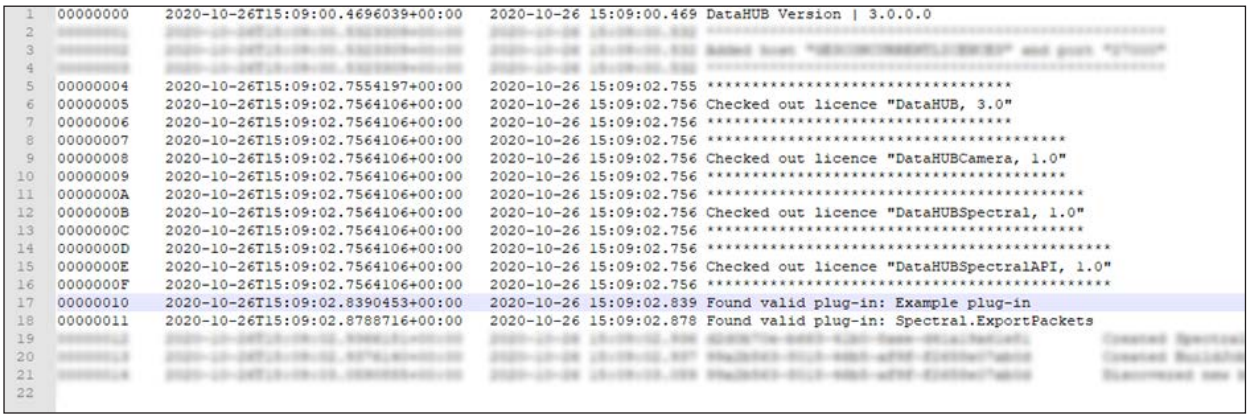


Figura 13 Plugin valido rilevato da DataHUB

6.3.5 Timeout

DataHUB non può garantire che tutte le chiamate a un'istanza del plugin vengano restituite in modo puntuale, per cui adotta un criterio di timeout. Se la chiamata a un plugin non viene completata entro un certo lasso di tempo, il sistema presume la presenza di un errore e la chiude. I timeout predefiniti sono:

- Per *Initialise*, 20 secondi
- Per *ProcessPackets*, 200 secondi
- Per *Shutdown*, 20 secondi

Capiterà spesso, durante il debug, che il timeout venga superato e che DataHUB chiuda il plugin prematuramente. Per allungare il tempo di debug, è possibile incrementare la durata del timeout, modificando il file <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns\PlugInsConfiguration.xml> con un editor per XML o testi.

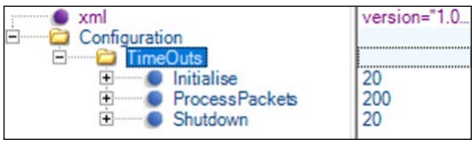


Figura 14 Timeout predefiniti per i plugin

NOTA: il servizio DataHUB deve essere riavviato per applicare le modifiche al timeout.

6.3.6 Debug del plugin

Dopo che un plugin valido è stato distribuito, è possibile eseguire il debug dell'esecuzione. Il plugin non è non è eseguibile direttamente, ma se il sistema di debug di Visual Studio viene collegato al servizio DataHUB e si avvia una costruzione, l'implementazione del plugin verrà invocata dal servizio e i punti di interruzione impostati nel codice saranno raggiunti.

NOTA: per effettuare il debug tramite il servizio, Visual Studio potrebbe richiedere diritti da amministratore. Per ottenere tali diritti, fare clic con il pulsante destro del mouse sul comando di scelta rapida di Visual Studio e selezionare "Esegui come amministratore".

In Visual Studio impostare i punti di interruzione all'inizio dei metodi *Initialise* e *ProcessPackets*:

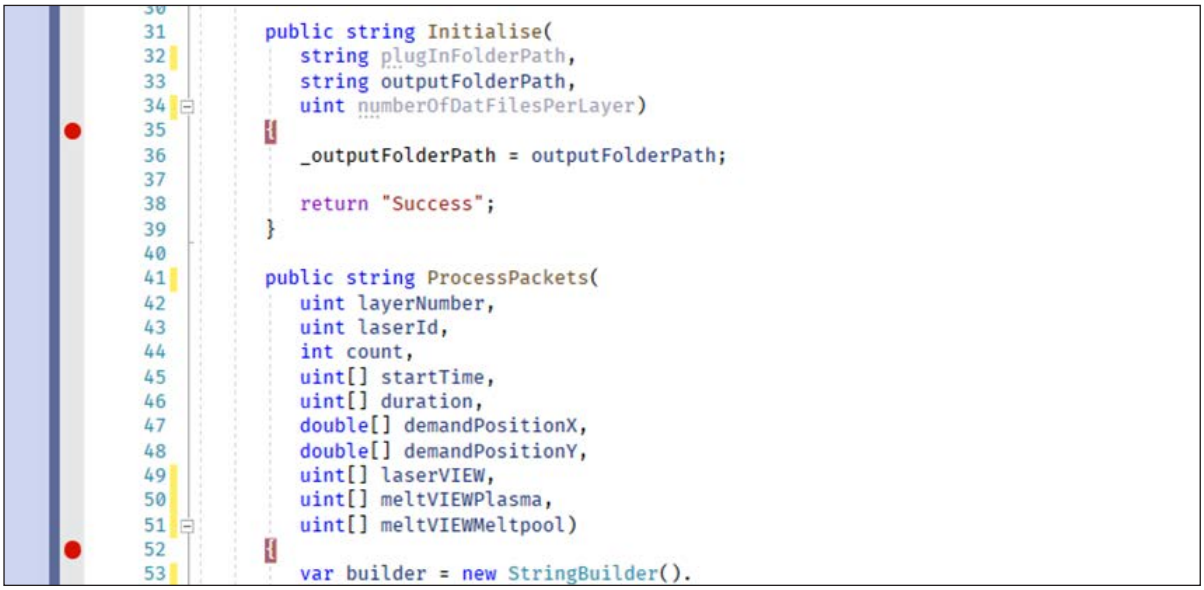


Figura 15 Punti di interruzione impostati in *Initialise* e *ProcessPackets*

Aprire il menu di debug e selezionare “Connetti a processo”:

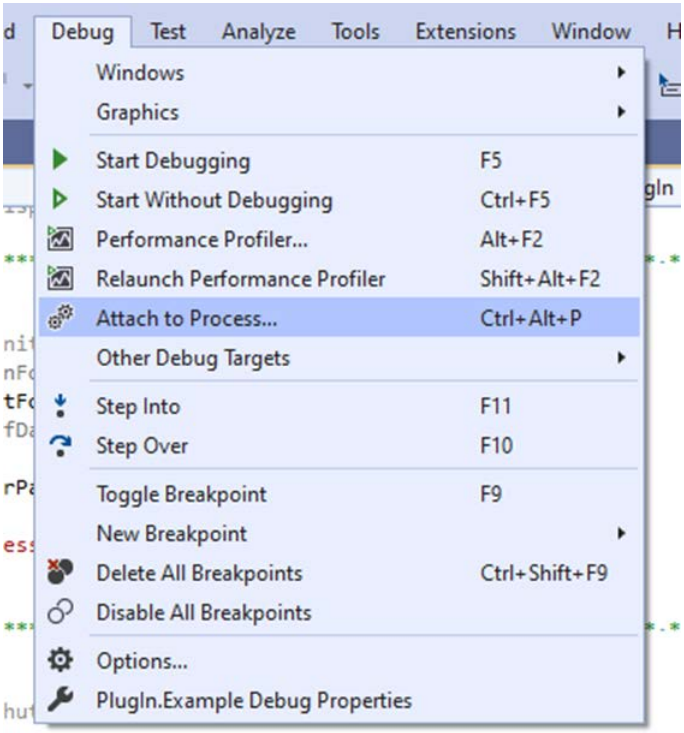


Figura 16 Connessione a un processo per il debug

Nell’elenco dei processi in corso trovare *DataHUB Service.exe*. Potrebbe essere necessario selezionare “Mostra processi da tutti gli utenti”:

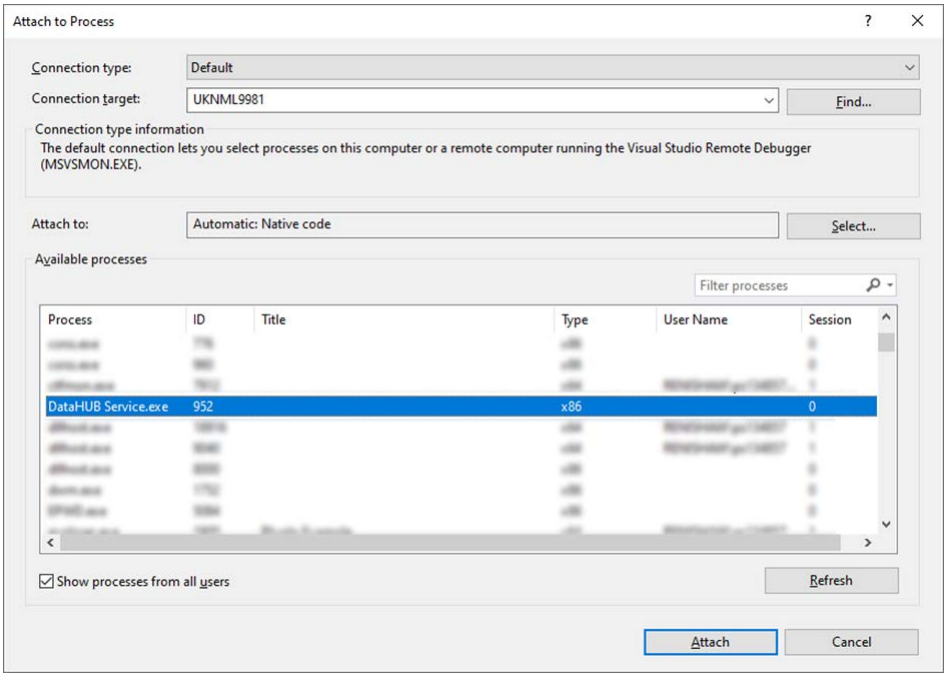


Figura 17 Processo DataHUB Service

Premere il pulsante “Connetti” e Visual Studio inizierà una sessione di debug sul servizio DataHUB.

Per fare in modo che DataHUB richiami il plugin e che i punti di interruzione vengano raggiunti, è necessario avviare una costruzione. Questo viene fatto tramite DataHUB Generator oppure, se alcune costruzioni sono già state generate, duplicando la cartella di una costruzione in <$PROGRAMDATA$\Renishaw\DataHUB\Build Jobs>.

DataHUB rileva automaticamente la nuova costruzione e inizia a elaborare i dati. Una delle attività preliminari del servizio è di creare (invocando il costruttore senza parametri) un'istanza di ciascun plugin registrato. Quando poi DataHUB richiama l'istanza del metodo *Initialise*, il primo punto di interruzione viene raggiunto. I valori dei parametri del metodo sono quelli della costruzione in questione, ad esempio:

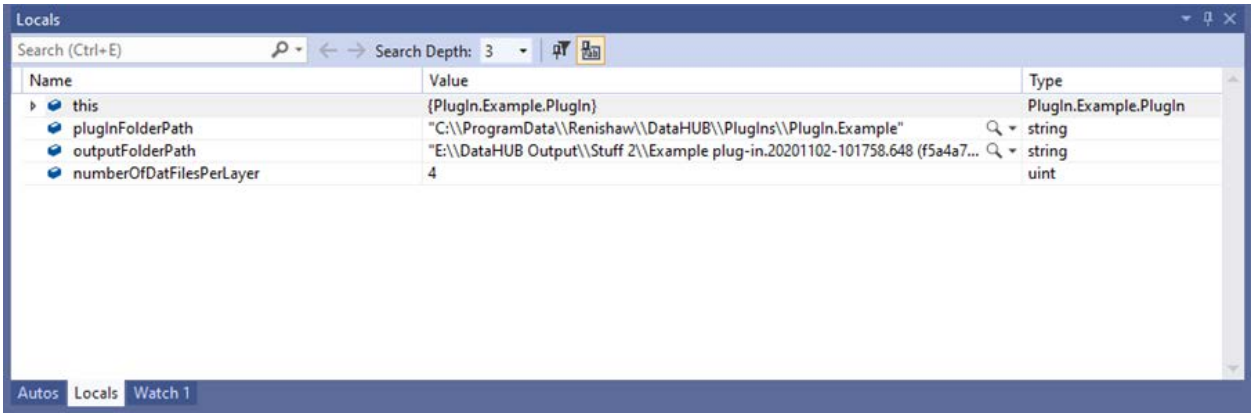


Figura 18 Valori dei parametri del metodo *Initialise*

Da questo momento in poi, i dati esistenti (e, nel caso della costruzione in corso, i nuovi dati che arrivano) vengono elaborato e il metodo *ProcessPackets* dell'istanza viene richiamato con i valori dei parametri di LaserVIEW e MeltVIEW associati a uno strato:

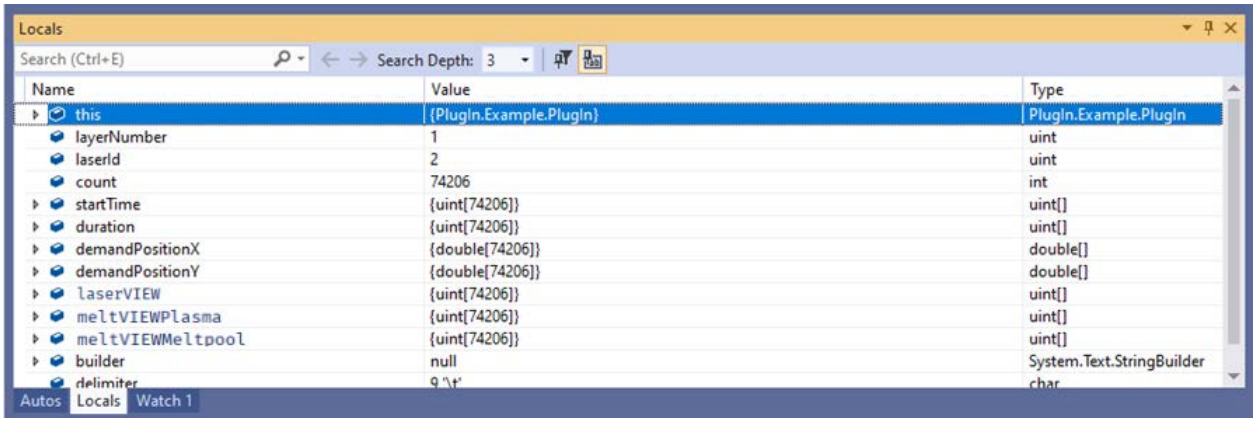


Figura 19 Valori dei parametri del metodo *ProcessPacket*

Il compito di aggiungere e verificare i punti di interruzione nel metodo *Shutdown* è lasciato al lettore.

6.3.7 Distribuzione del plugin per la produzione

Durante la preparazione del plugin in un DCPC di produzione, è fondamentale compilare, distribuire localmente e testare una costruzione di rilascio. Dopo che la funzionalità del plugin è stata convalidata e verificata, può essere distribuita nei DCPC di produzione.

Il processo di distribuzione di un DCPC di produzione è molto simile a quello usato per il PC di sviluppo. La cartella *Release* deve essere copiata nella sottocartella *PlugIns* della cartella dati di DataHUB (<$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>) nel DCPC e rinominata in modo adeguato al plugin. Il servizio DataHUB deve essere quindi riavviato per registrare il nuovo plugin: a tale scopo, usare l'applicazione *Servizi*. Il successo o meno della registrazione del nuovo plugin viene riportata nel registro giornaliero delle verifiche.

NOTA: DataHUB è stato sviluppato per recuperare tutte le attività di elaborazione dati in corso dopo il riavvio. Per questo motivo, non è necessario attendere il completamento di tutte le attività prima di riavviare. Tuttavia, solo le attività di elaborazione dei nuovi dati utilizzeranno il plugin appena registrato.

6.3.8 Diagnosi degli errori dei plugin durante la produzione

In un ambiente di produzione, DataHUB Monitor mostra (insieme ad altre attività di elaborazione della costruzione) lo stato di elaborazione del plugin:

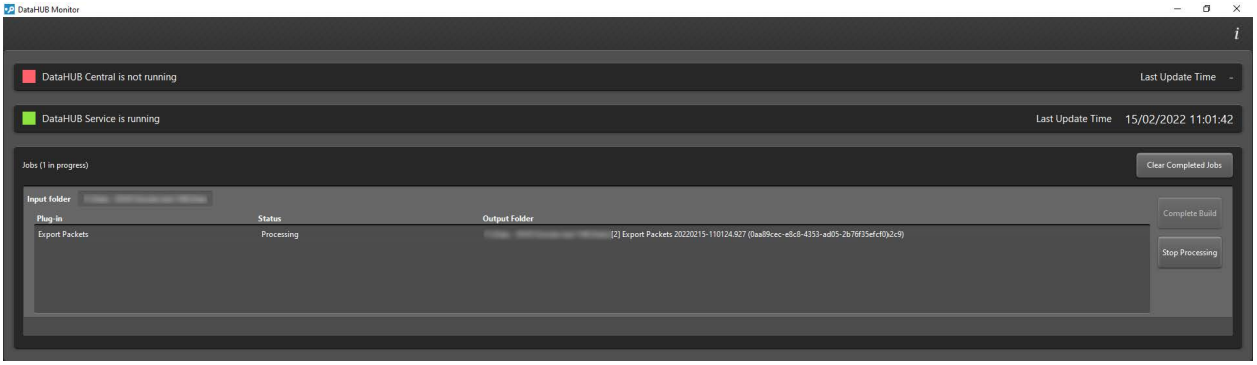


Figura 20 Esecuzione di una costruzione con plugin

Se lo stato di un plugin è *Errore*, il registro delle verifiche conterrà i record dell'errore, come voci aggiunte da DataHUB (ad esempio, un errore di caricamento dell'assemblaggio plugin a causa di dipendenze mancanti) oppure inserite dal plugin tramite i contenuti del valore di ritorno dai metodi *Initialise*, *ProcessPackets* o *Shutdown* del plugin.

6.3.9 Risoluzione dei problemi

Nella tabella di seguito sono riportati gli errori più comuni, con la loro voce nel registro delle verifiche e le possibili soluzioni:

Voce nel registro delle verifiche	Errore	Possibile soluzione
The assembly does not contain a class named PlugIn	Impossibile trovare classe PlugIn	Cambiare nome alla classe plugin, ricompilare e ridistribuire
The PlugIn class does not contain a method named xyz	La classe PlugIn non implementa uno dei metodi API	Controllare la classe per verificare che non vi siano metodi mancanti o con errori ortografici
The xyz method does not have the expected signature …	La classe PlugIn non implementa correttamente uno dei metodi API	Controllare la firma del metodo e verificare che tutti i parametri corrispondano alla definizione dell'API
The PlugIn class is using an unsupported version of the plugin API '{version}'. This version of DataHUB only supports version '1.0' of the plugin API.	La classe PlugIn non restituisce una stringa valida per la versione API	Verificare che il metodo APIVersion restituisca "1.0"
Failed to create plug-in output folder <path>	Non è stata creata la cartella per un'istanza specifica di un plugin	Assicurarsi di avere diritti di lettura/scrittura/modifica/ creazione per <percorso>
Initialisation result: timed out Process result: timed out Shutdown result: timed out	La risposta a una chiamata a uno di questi metodi API ha impiegato troppo tempo: si presume che il plugin sia in errore	Valutare la possibilità di ridurre la quantità di lavoro svolta nel metodo oppure incrementare la durata del timeout in PlugInsConfiguration.xml
Exception:	Si è verificato un errore imprevisto	Esaminare i messaggi di eccezione registrati per determinare l'origine dell'errore

6.3.10 Integrazione di Renishaw Central

I risultati dell'analisi (punti dati serie temporale, avvisi e altro) di un plugin possono essere inoltrati a un server Renishaw Central. *API per plugin V2.0 e tutorial* (codice Renishaw H-5800-6784) fornisce una descrizione dettagliata sulle modalità di strutturazione dei contenuti della stringa restituita dai metodi *Initialise* e *ProcessPackets* del plugin per ottenere questa integrazione. Include inoltre dettagli sulle funzioni dell'helper di *PlugIns.API.v2.dll* che risultano compatibili con l'API V1.0.

6.3.11 API per plugin V1.0

6.3.11.1 Assegnazione di un nome all'assemblaggio

Il nome dell'assemblaggio .NET deve iniziare con "PlugIn" ("PlugIn" seguito da un punto) e deve avere l'estensione "dll".

6.3.11.2 Assegnazione di un nome alla classe del plugin

L'assemblaggio plugin deve definire una classe pubblica denominata "`PlugIn`".

6.3.11.3 Metodi della classe del plugin

La classe del plugin deve implementare i seguenti metodi pubblici:

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

6.3.11.4 Costruttore senza parametri

Il tipo deve avere un costruttore senza parametri. C# lo fornisce automaticamente se non è stato definito nessun costruttore con parametri e non è necessario definirlo in modo esplicito.

NOTA: Un plugin non può acquisire risorse all'interno del suo costruttore.

Un'istanza del plugin costruita potrebbe non richiamare il metodo *Shutdown* e queste risorse non potrebbero essere rimosse tempestivamente. *Initialise* è il luogo appropriato per acquisire risorse.

6.3.11.5 Metodo `APIVersion`

Identifica la versione dell'API da usare per la convalida del plugin e il formato dei dati di monitoraggio del processo che verranno elaborati.

Parametri:

Nessuno.

Restituisce: `string`

Un plugin che implementa API V1.0 deve restituire il valore letterale "1.0".

6.3.11.6 Metodo `DisplayName`

Il contenuto della stringa restituita da questo metodo viene utilizzata come nome del plugin visualizzato in DataHUB Monitor, durante la creazione della cartella di output del plugin e durante la verifica degli eventi del plugin. Anche se non è necessario, l'uso di un nome univoco aiuta a identificare il plugin quando, ad esempio, vi sono più versioni installate in un DCPC.

Parametri:

Nessuno.

Restituisce: `string`

Il valore letterale da usare in varie parti di DataHUB UX, per la registrazione e altre attività.

6.3.11.7 Metodo `Initialise`

Questo metodo viene richiamato da DataHUB all'inizio di una costruzione, prima che i dati sullo strato vengano inviati al plugin, per questo motivo riceve valori di parametro *invariabili sulla costruzione*. Il plugin può utilizzare questo metodo per allocare le risorse necessarie durante la durata dell'istanza, calcolare le costanti della costruzione e così via.

Parametro 1: `string`

Il percorso della cartella dell'assemblaggio plugin, ad esempio, <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns\PlugIn.Example>

Parametro 2: `string`

Il percorso di una cartella in cui il plugin può scrivere in file di output. Nota: questa cartella deve essere univoca per ciascun plugin e per ciascuna costruzione e deve avere il nome di visualizzazione del plugin (come definito dal valore restituito dal metodo `DisplayName`), data e ora e un GUID. Ogni volta che si esegue il plugin per una costruzione, il nome della cartella cambia, ma mantiene sempre la stessa struttura:

<Cartella di output costruzione>\[1.0] <nome di visualizzazione del plugin>.aaaaMMgg-HHmmss.fff (GUID)

Parametro 3: `uint`

Un valore da 1 a 4, secondo quanto definito dalla configurazione dell'hardware della macchina che crea i dati.

Restituisce: `string`

Il plugin deve restituire una stinga contenente:

- la parola "Success", se il metodo è stato completato correttamente,
- i dettagli desiderati sulla causa di un eventuale fallimento oppure
- una stringa JSON conforme all'API di ritorno di Renishaw Central.

Se DataHUB riceve una stringa non conforme, il plugin verrà chiuso e il contenuto della stringa restituita sarà aggiunto al registro delle verifiche per analisi successive.

6.3.11.8 Metodo `ProcessPackets`

Questo metodo viene richiamato da DataHUB quando tutti i dati di uno strato sono stati ricevuti dal DCPC. I parametri di questo metodo sono:

Parametro 1: `uint`

Il numero dello strato della costruzione a cui i dati fanno riferimento. Il primo strato è lo strato 1.

Parametro 2: `uint`

L'identificativo "1", "2", "3" o "4" del laser per cui sono stati raccolti i dati.

Parametro 3: `int`

Il conteggio del numero di elementi negli insiemi successivi.

Parametro 4: `uint[ ]`

Tempo di inizio di ciascun pacchetto, in µs, relativo all'inizio dello strato.

Parametro 5: `uint[ ]`

Durata di ciascun pacchetto, in µs.

Parametro 6: `double[ ]`

Posizione della richiesta X di ciascun pacchetto sul letto di polvere, in mm.

Parametro 7: `double[ ]`

Posizione della richiesta Y di ciascun pacchetto sul letto di polvere, in mm.

Parametro 8: `uint[ ]`

Il valore medio di tutti i campioni raccolti dal sensore di LaserVIEW nel corso della durata di ciascun pacchetto.

Parametro 9: `uint[ ]`

Il valore medio di tutti i campioni raccolti dal sensore di MeltVIEW Plasma nel corso della durata di ciascun pacchetto.

Parametro 10: `uint[ ]`

Il valore medio di tutti i campioni raccolti dal sensore di Meltview Meltpool nel corso della durata di ciascun pacchetto.

Restituisce: `string`

Il plugin deve restituire:

- una stringa contenente la parola “Success”, se il metodo è stato completato correttamente,
- una stringa con quanti più dettagli possibile sulla causa di un eventuale fallimento oppure
- Una stringa JSON conforme all’API di ritorno di Renishaw Central.

6.3.11.9 Metodo `Shutdown`

Questo metodo viene richiamato da DataHUB dopo che tutti i dati di una costruzione sono stati inviati al plugin o in caso di errori con tutte le chiamate *Initialise* al plugin. In questo metodo il plugin deve eseguire le elaborazioni finali dei dati di costruzione tutte le risorse che potrebbe avere acquisito e così via. Il metodo non ha parametri.

Parametri:

Nessuno.

Restituisce: `string`

Il plugin deve restituire una stinga contenente:

- la parola “Success”, se il metodo è stato completato correttamente,
- quanti più dettagli possibile sulla causa di un eventuale fallimento oppure

Se DataHUB riceve una stringa non conforme, il contenuto della stringa restituita sarà aggiunto al registro delle verifiche per analisi successive.

6.4 API per plugin V2

Questa sezione costituisce una guida per creare, testare e implementare un componente software (V2.0) per plugin da usare con DataHUB. Il ciclo di sviluppo di un plugin inizia con l'installazione di DataHUB nel PC di sviluppo. Successivamente, utilizzare Visual Studio per creare un assemblaggio .NET contenente il codice del plugin. Utilizzare una configurazione appropriata di Visual Studio per testare ed eseguire il debug del plugin e verificare che sia corretto prima di installarlo nel PC di raccolta dati (DCPC) di un sistema di produzione AM.

In questa sezione, viene definita la struttura del *flusso di token* di una costruzione. Un esempio di codice spiega quindi come scrivere un flusso di token elaborando il plugin tramite V2.0 API. Il secondo esempio descrive l'uso dei *filtri* per l'elaborazione del flusso di token. Successivamente, viene ricreata l'implementazione del plugin *ExportPackets* (così come viene distribuito dal software DataHUB). Gli ultimi esempi di codice descrivono come inviare i risultati di un plugin a Renishaw Central, per visualizzarli a fianco degli altri dati raccolti dai sistema AM di Renishaw.

La sezione si conclude con una definizione dell'API V2.0.

6.4.1 Attività necessarie

Oltre al presente documento, leggere quanto segue:

- Guida all'installazione del software *InfiniAM® e DataHUB* (codice Renishaw H-5800-6845)
- Guida all'uso di *DataHUB* (codice Renishaw H-5800-6853)
- Guida all'uso di Renishaw Licence Manager v2.x

Prima di continuare, svolgere le seguenti attività nel PC da utilizzare per lo sviluppo del plugin:

- Assicurarsi che il PC disponga di una GPU con le specifiche minime per DataHUB (vedere le specifiche hardware del PC di raccolta dati nella guida all'installazione dei *software InfiniAM e DataHUB*).
- Verificare che siano stati installati i driver più aggiornati per la GPU.
- Accertarsi che il server delle licenze disponga di un numero sufficiente di licenze appropriate per DataHUB e Spectral API.
- Controllare che dalla rete sia possibile accedere al server delle licenze.
- Verificare che sia installata una versione di Microsoft Visual Studio in grado di supportare .NET Framework 4.7.2.

6.4.2 Flussi di token

6.4.2.1 Struttura dei flussi di token

Il servizio DataHUB trasforma i dati di monitoraggio del processo di una costruzione in un flusso di token che verrà consumato dai plugin V2.0. L'ordine e il tipo di token presenti in un flusso creano una descrizione pseudogerarchica dei dati di monitoraggio del processo.

I token di un flusso ricadono in due categorie: token accoppiati di inizio e fine e token vincolati da token accoppiati di inizio e fine.

6.4.2.2 Il flusso per un laser e uno strato

I dati di monitoraggio del processo per un laser e uno strato della costruzione sono rappresentati nel seguente flusso di token:

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

Nota: i token accoppiati `StartOfLayerLaserData` e `EndOfLayerLaserData` segnano un limite concettuale intorno ai token specifici del laser.

Il token `DataSourceInformation` contiene valori presi dalle origini dei dati di monitoraggio del processo (ad esempio, file DAT) contenenti i dati campione.

Ciascun `Sample` token contiene i valori dei dati di monitoraggio del processo raccolti dagli hardware LaserVIEW e MeltVIEW per un unico campione da 100 KHz.

6.4.2.3 Il flusso per uno strato

La serie di tutti i dati di monitoraggio del processo di uno strato viene combinata nel flusso di token, ripetendo la struttura del laser (una volta per ciascun laser presente in macchina):

```
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)
```

Anche in questo caso, i token specifici per lo strato sono vincolati a dati `StartOfLayerData` e `EndOfLayerData` accoppiati.

6.4.2.4 Flusso di una costruzione

Tutti i dati di una costruzione creano un flusso di token che ripete la struttura dello strato (per ciascuno strato presente in macchina):

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

6.4.2.5 Separazioni

Un dettaglio secondario è stato omesso dalla struttura del flusso di token appena descritta: i token `Split`. I token Split vengono aggiunti al flusso di token per segnare i punti in cui i dati cambiano carattere. Vengono aggiunti come segnale generico, in aggiunta ai token `StartOf/EndOf…`, più specifici. Durante l'elaborazione di un flusso di token, risulta spesso più semplice agire su questo token generico piuttosto che sui marcatori specifici.

6.4.2.6 Definizione completa del flusso di token di una costruzione

La definizione completa del flusso di token di una costruzione è:

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

6.4.2.7 Aggiunta di un tipo di token

L'API definisce una classe base `Token` da cui derivano tutti i token. Per aggiungere un nuovo tipo di token nel flusso, è necessario farlo derivare da questa classe:

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

Il catalogo completo dei token delle API è reperibile nella sezione 6.4, "API per plugin V2".

6.4.3 Installazione di DataHUB per lo sviluppo dei plugin

Il servizio DataHUB deve essere installato nel PC di sviluppo. Durante l'installazione, viene offerta la possibilità di scegliere le licenze da richiedere. Assicurarsi che l'opzione "LaserVIEW/MeltVIEW Pluggability" (Connettibilità a LaserVIEW/MeltVIEW) sia selezionata:

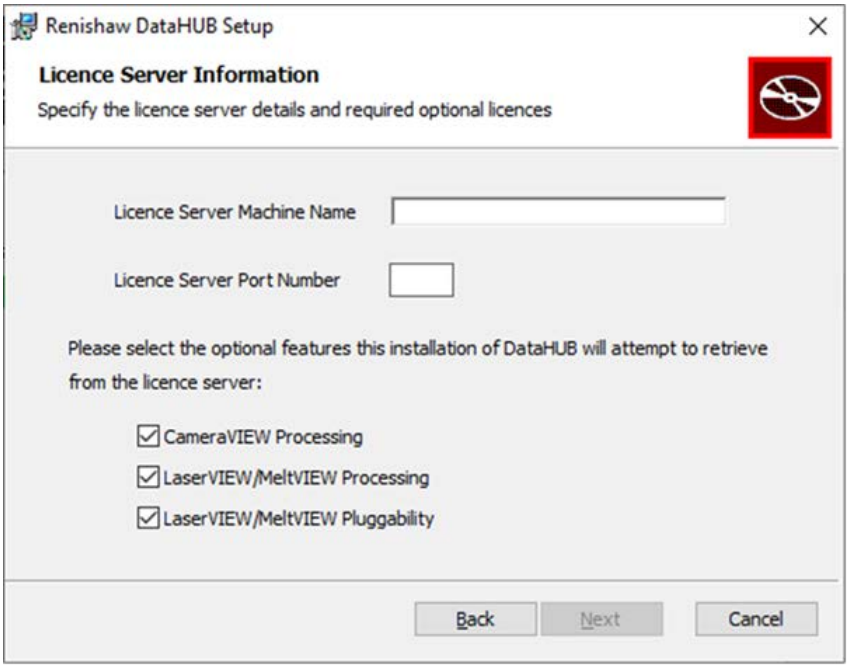


Figura 21 Licenze DataHUB

6.4.4 Creazione di un plugin in Visual Studio

Nelle sezioni successive di questo documento viene spiegato come creare una soluzione plugin con Visual Studio.

I plugin sono scritti in C# e definiscono una classe pubblica per implementare la specifica serie di metodi pubblici usati da DataHUB per identificare, convalidare e utilizzare le istanze del plugin durante l'elaborazione dei dati di monitoraggio del processo di una costruzione.

NOTA: il flusso di lavoro riportato di seguito usa Microsoft Visual Studio 2019, ma risulta molto simile anche in altre versioni di Microsoft Visual Studio.

6.4.4.1 Creazione della soluzione

Avviare Visual Studio e selezionare *Crea nuovo progetto*. Fra le opzioni disponibili, selezionare *Libreria classi (.NET Framework)*:

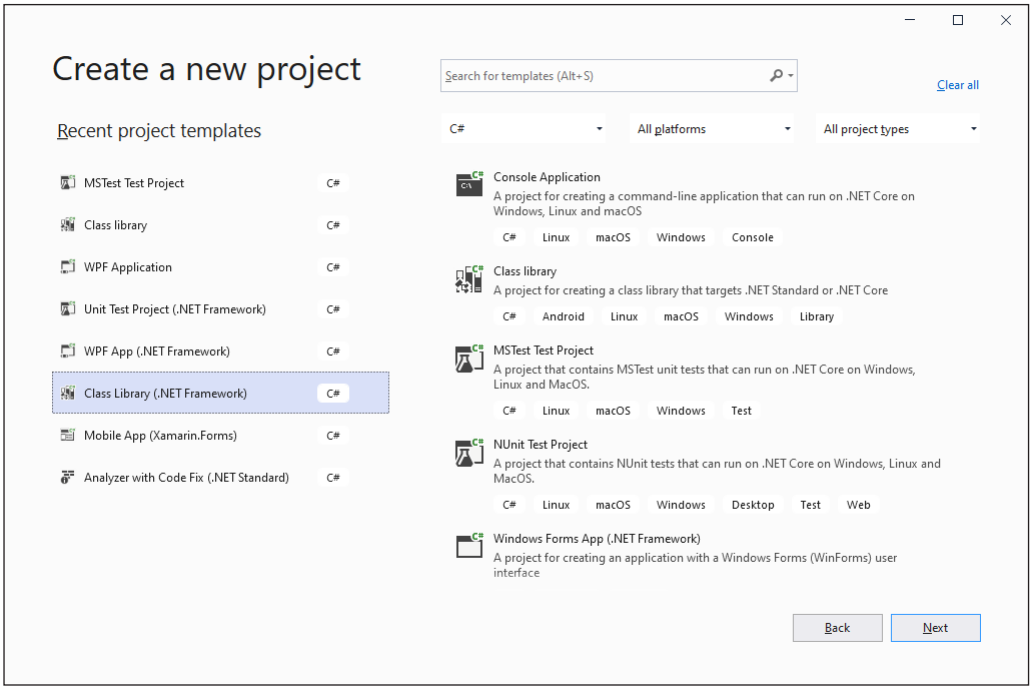


Figura 22 Creazione della soluzione

Configurare il progetto:

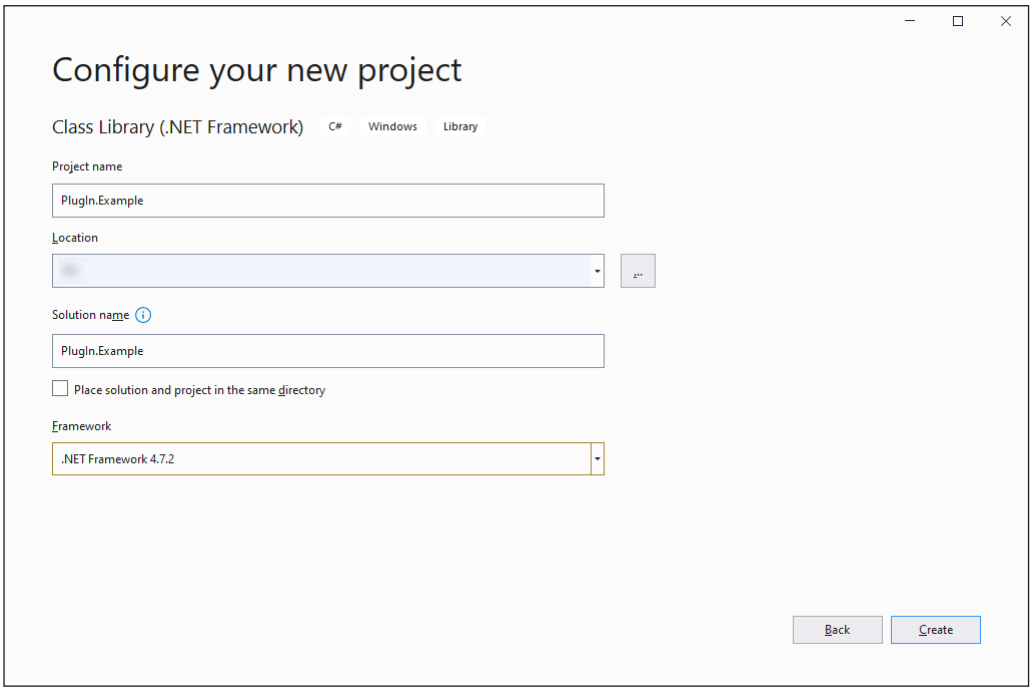


Figura 23 Configurazione del progetto

DataHUB richiede un assemblaggio plugin che inizi con **PlugIn.** (La parola “PlugIn” seguita da un punto). Selezionare una cartella a scelta come *posizione* e creare la soluzione.

NOTA: DataHUB supporta i plugin per x86/x64/Any CPU. Si consiglia di indicare Framework 4.7.2 o superiore.

6.4.4.2 Aggiunta di un riferimento all'assemblaggio API del plugin

Per sviluppare un plugin per l'API V2.0 API, è necessario un riferimento all'assemblaggio API del plugin. Fare clic con il pulsante destro del mouse sulla voce di soluzione "References" (Riferimenti) e selezionare "Add Reference…" (Aggiungi riferimento)

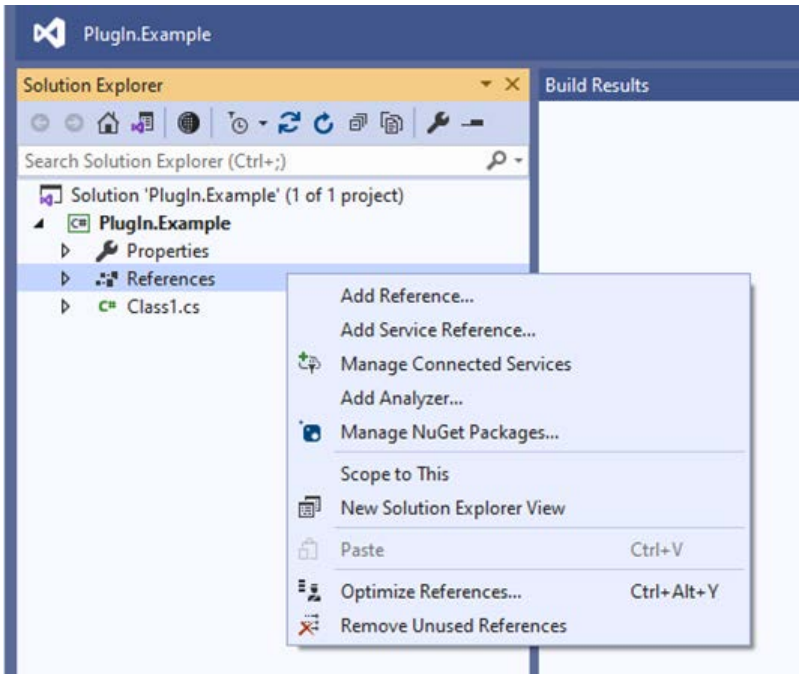


Figura 24 Aggiunta di un riferimento

In Reference Manager, scegliere "Browse…" (Sfoglia) e posizionare "PlugIns.API.v2.dll" nella cartella di installazione di DataHUB (ad esempio, <$PROGRAMFILES$\Renishaw\DataHUB>):

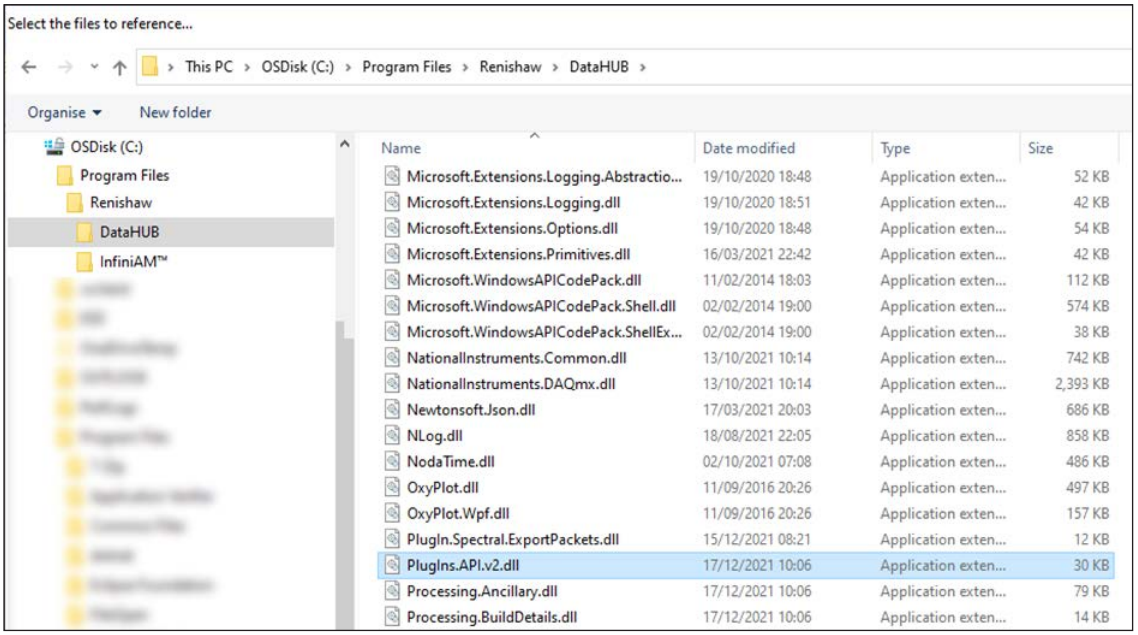


Figura 25 Assemblaggio API

L'assemblaggio verrà elencato nelle dipendenze del progetto del plugin:

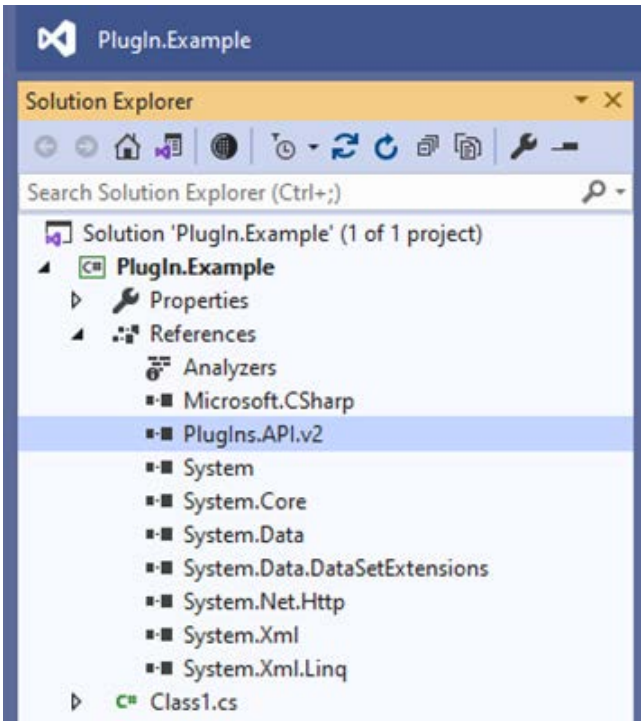


Figura 26 L'assemblaggio API del plugin indicato come dipendenza

6.4.4.3 Assegnare un nome al tipo di plugin

Il singolo file sorgente Class1.cs contiene la definizione di Class1 che verrà usata come punto iniziale.

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

Quando DataHUB rileva un potenziale assemblaggio plugin, cerca un `PlugIn` di tipo pubblico. Rinominare `Class1` in:

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

6.4.4.4 Aggiunta dell'API

Con un plugin di tipo potenziale, DataHUB richiede l'implementazione dei seguenti metodi pubblici:

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

Aggiungere le seguenti istruzioni d'uso e implementazioni di metodo alla 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();
}
```

Si noti che tutti i metodi API restituiscono una `string`. Il contenuto della stringa restituita viene utilizzato per segnalare a DataHUB il risultato del metodo. Il ruolo di questo valore restituito verrà spiegato man mano che ciascun metodo viene definito.

6.4.4.5 Metodo `APIVersion` statico

Questo metodo indica a DataHUB la versione dell'API usata dal plugin. Per usare l'API V2.0, deve restituire il valore "1.0":

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

6.4.4.6 Metodo DisplayName statico

Il contenuto della stringa restituita da questo metodo viene usato come nome del plugin visualizzato in DataHUB Monitor, al momento di creare la cartella di output del plugin e così via. Anche se non è necessario, l'uso di un nome univoco aiuta a identificare il plugin quando, ad esempio, vi sono più versioni installate in un DCPC.

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

6.4.4.7 Metodo `Initialise`

Questo metodo viene richiamato da DataHUB all'inizio di una costruzione, prima che i dati sugli strati siano inviati al plugin. Per questa ragione, passa al plugin dati *invariabili sulla costruzione*. L'implementazione minima prevede la restituzione di quanto inizializzato dal plugin, senza errori:

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

6.4.4.8 Metodo `Process`

Questo metodo viene richiamato da DataHUB dopo che tutti i dati di uno strato sono stati ricevuti. L'implementazione minima restituisce che il plugin ha elaborato i dati senza errori:

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

6.4.4.9 Metodo `Shutdown`

Questo metodo viene richiamato da DataHUB dopo che tutti i dati di una costruzione sono stati inviati al plugin o in caso di errori con tutte le precedenti chiamate al plugin. Anche in questo caso, l'implementazione minima è la restituzione senza errori:

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

6.4.4.10 Distribuzione del plugin per il debug

Creare la soluzione in *Debug* e aprire un browser per trovare la cartella di output che conterrà un file *.dll* con il nome assegnato al progetto (ad esempio, *PlugIn.Example.dll*). Potrebbero essere presenti anche altri file, come ad esempio simboli di debug o altro.

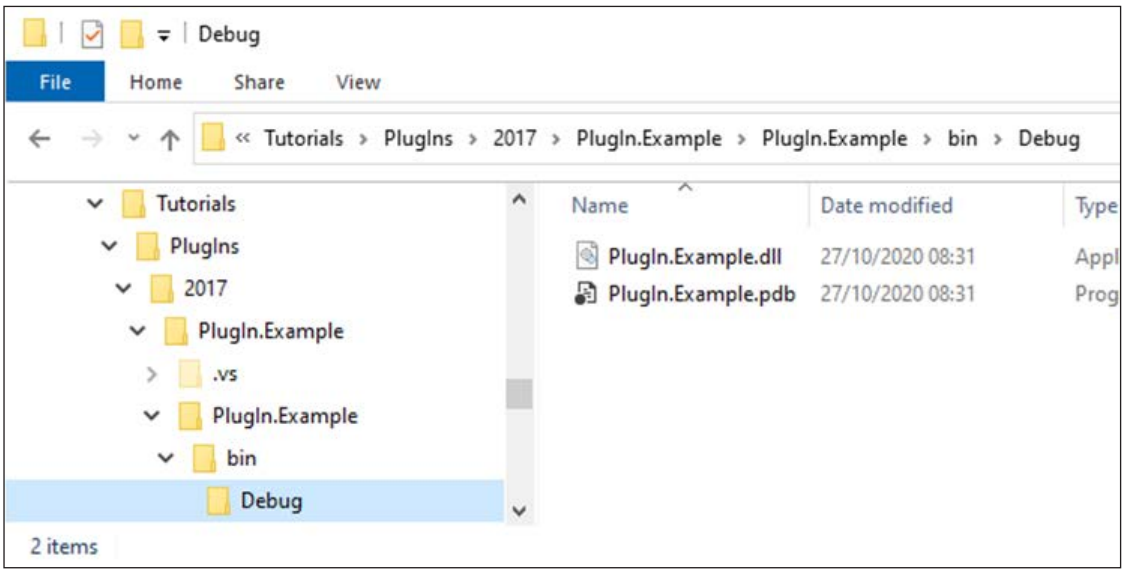


Figura 27 File binari del plugin

Copiare questa cartela e copiarla nella sottocartella "PlugIns" della cartella dati di DataHUB plugin (<$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>).

NOTA: in genere, $PROGRAMDATA$ si trova in <C:\ProgramData>, ma potrebbe essere nascosta. In questo caso, selezionare l'opzione di Esplora risorse di Windows "Visualizza – Mostra/Nascondi – Elementi nascosti".

Per modificare il contenuto di questa cartella è necessario disporre dei diritti di amministratore.

Rinominare la cartella *Debug* con un nome adeguato al plugin. Nell'immagine di seguito, alla cartella è stato assegnato il nome *PlugIn.Example*.

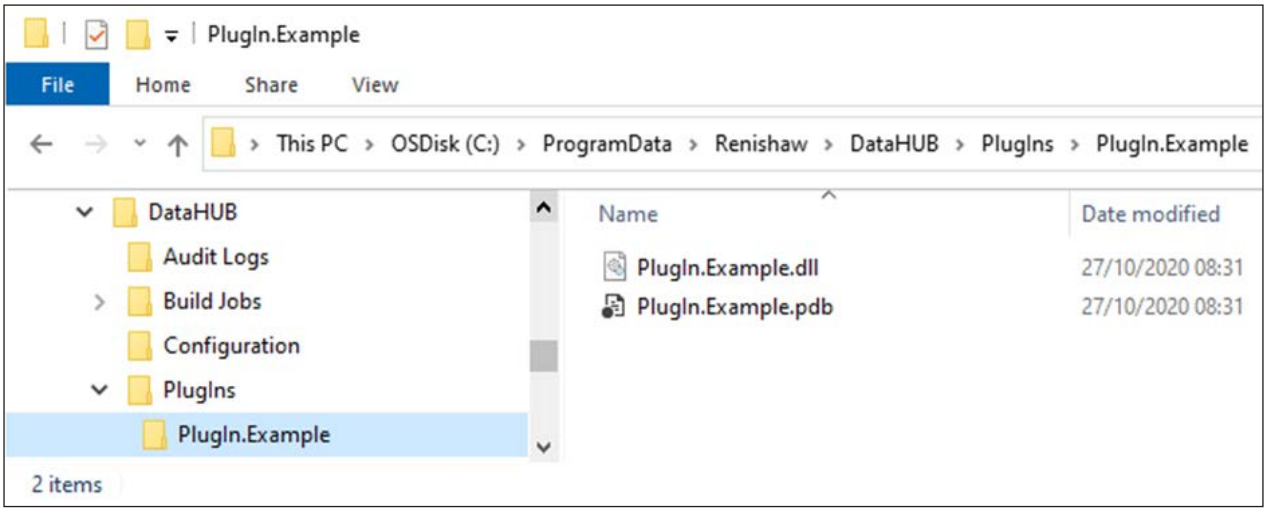


Figura 28 Il plugin dopo la distribuzione

6.4.4.11 Registrazione del plugin

È necessario riavviare il servizio DataHUB per registrare il nuovo plugin. A tale scopo, utilizzare l'applicazione *Servizi*. Fare clic con il pulsante destro del mouse su *DataHUB Service* e selezionare *Riavvia*:

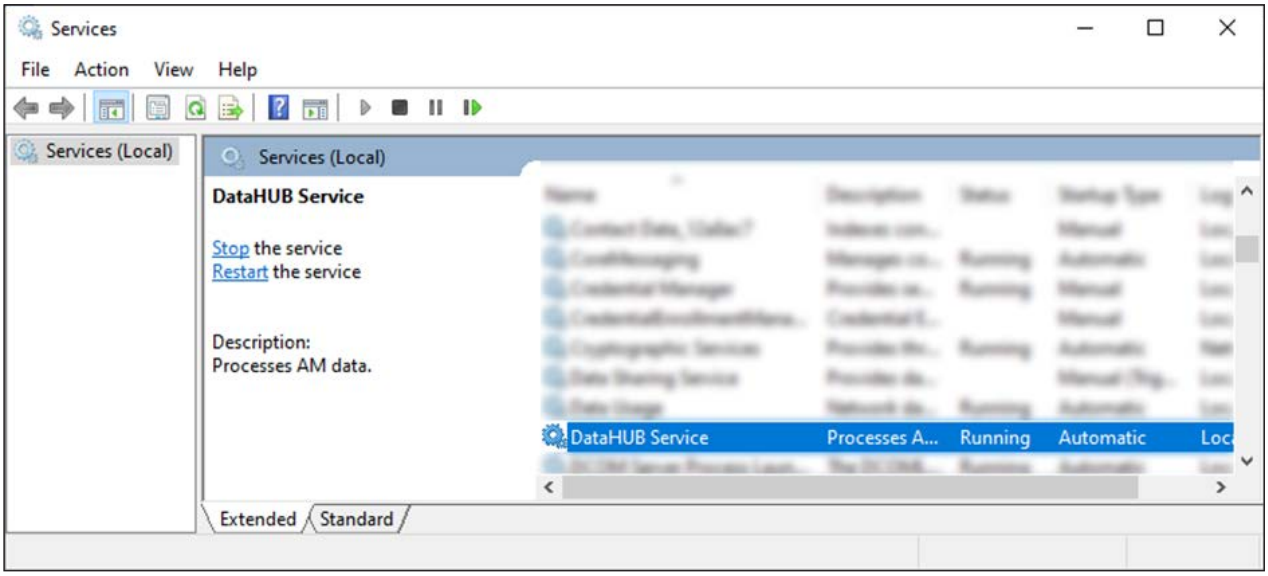


Figura 29 L'applicazione Servizi di Windows

Dopo alcuni secondi, lo stato del servizio DataHUB diventerà *In esecuzione*.

6.4.4.12 Verifica della registrazione

Durante l'esecuzione, DataHUB genera un file di registro in cui inserisce gli eventi significativi, come l'inizio della costruzione, gli errori e, ovviamente, la verifica dei plugin. Usare Esplora risorse per trovare la cartella <$PROGRAMDATA$\Renishaw\DataHUB\Audit Logs>:

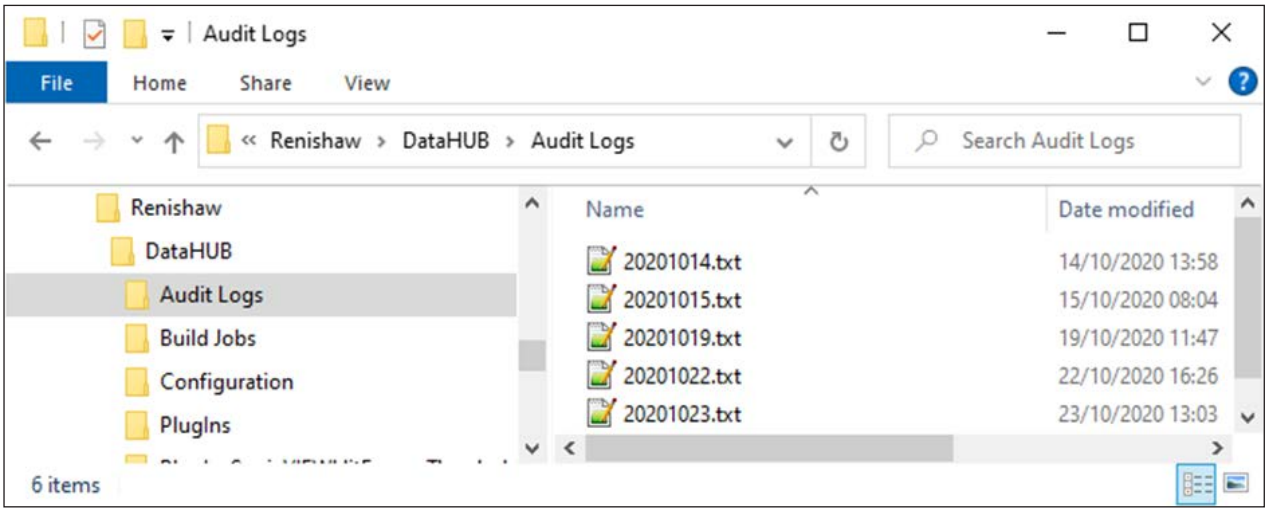


Figura 30 Cartella DataHUB con i file di registro

Aprire il file di registro più recente (dovrebbe riportare la data odierna), scorrere fino alla fine e cercare le righe che riportano il rilevamento di plugin validi:

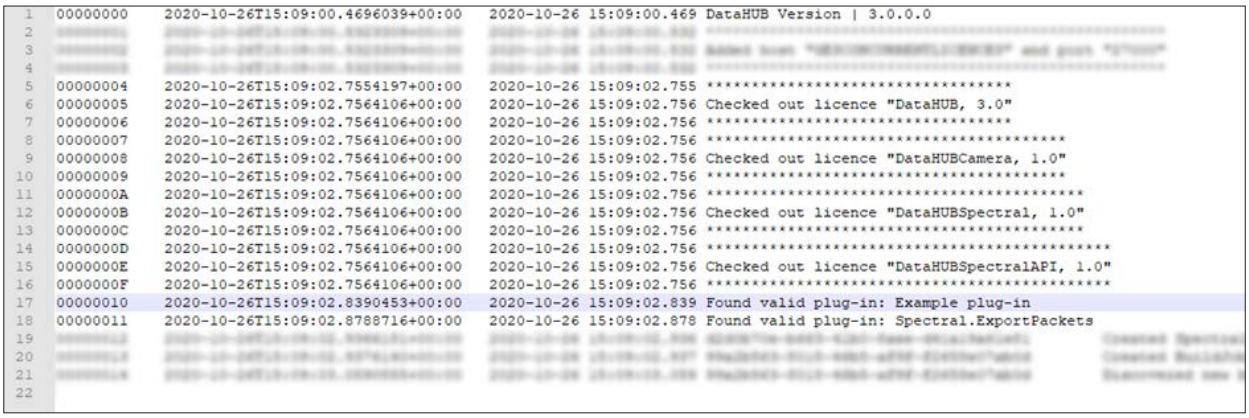


Figura 31 Plugin valido rilevato da DataHUB

6.4.4.13 Timeout

DataHUB non può garantire che tutte le chiamate a un'istanza del plugin vengano restituite in modo puntuale, per cui adotta un criterio di timeout. Se un plugin non viene completato entro un certo lasso di tempo, il sistema presume la presenza di un errore e lo chiude. I timeout predefiniti sono:

- Per *Initialise*, 20 secondi
- Per *Process* (condiviso con le API V1.0 *ProcessPackets*), 200 secondi
- Per *Shutdown*, 20 secondi

Capiterà spesso, durante il debug, che il timeout venga superato e che DataHUB chiuda il plugin prematuramente. Per allungare il tempo di debug, è possibile incrementare la durata del timeout, modificando il file <$PROGRAMDATA$\Renishaw\DataHUB\PlugIns\PlugInsConfiguration.xml> con un editor per XML o testi:

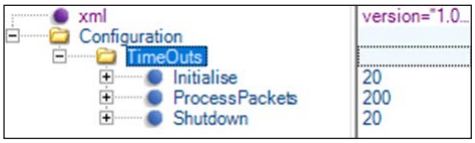


Figura 32 Timeout predefiniti per i plugin

NOTA: il servizio DataHUB deve essere riavviato per applicare le modifiche al timeout.

6.4.4.14 Debug del plugin

Dopo che un plugin valido è stato distribuito, è possibile eseguire il debug dell'esecuzione. Il plugin non è non è eseguibile direttamente, ma se il sistema di debug di Visual Studio viene collegato al servizio DataHUB e si avvia una costruzione, l'implementazione del plugin verrà invocata dal servizio e i punti di interruzione impostati nel codice saranno raggiunti.

NOTA: per effettuare il debug tramite il servizio, Visual Studio potrebbe richiedere diritti da amministratore. Per ottenere tali diritti, fare clic con il pulsante destro del mouse sul comando di scelta rapida di Visual Studio e selezionare "Esegui come amministratore".

In Visual Studio impostare i punti di interruzione all'inizio dei metodi *Initialise*, *Process* e *Shutdown*:

Figura 33 Punti di interruzione impostati in *Initialise* e *ProcessPackets*

Aprire il menu di debug e selezionare “Connetti a processo”:

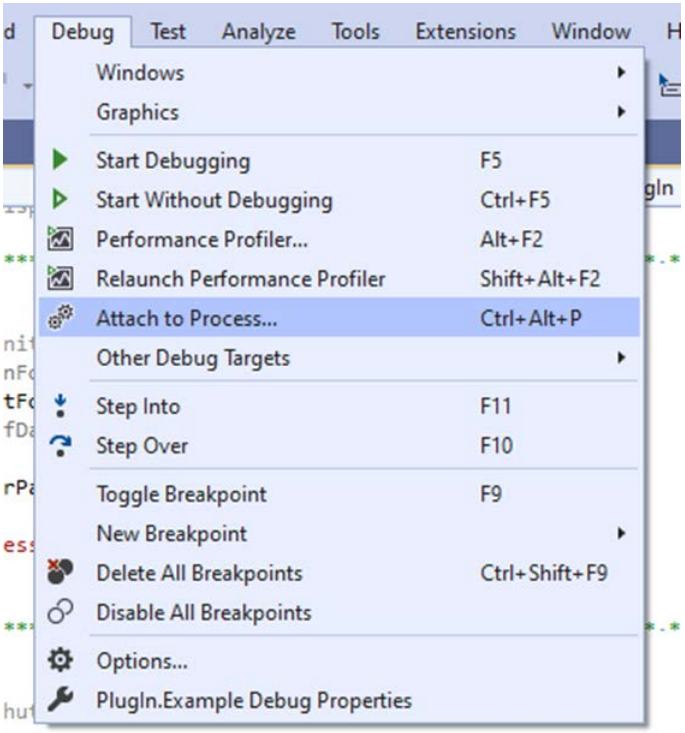


Figura 34 Connessione a un processo per il debug

Nell’elenco dei processi in corso trovare *DataHUB Service.exe*. Potrebbe essere necessario selezionare “Mostra processi da tutti gli utenti”:

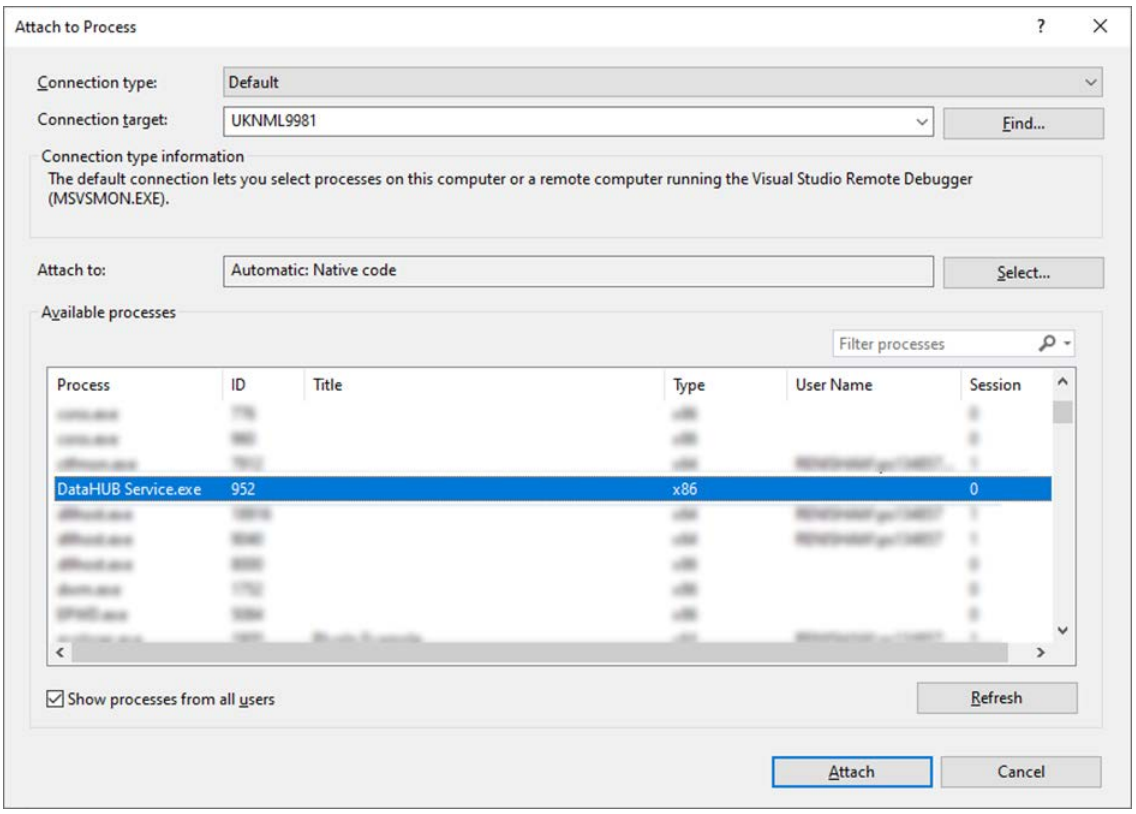


Figura 35 Processo DataHUB Service

Premere il pulsante “Connetti” e Visual Studio inizierà una sessione di debug sul servizio DataHUB.

Per fare in modo che DataHUB richiami il plugin e che i punti di interruzione vengano raggiunti, è necessario avviare una costruzione. Questo viene fatto tramite DataHUB Generator oppure, se alcune costruzioni sono già state generate, duplicando la cartella di una costruzione in <$PROGRAMDATA$\Renishaw\DataHUB\Build Jobs>.

DataHUB rileva automaticamente la nuova costruzione e inizia a elaborare i dati. Una delle attività preliminari del servizio è di creare (invocando il costruttore senza parametri) un’istanza di ciascun plugin registrato. Quando poi DataHUB richiama l’istanza del metodo *Initialise*, il primo punto di interruzione viene raggiunto. I valori dei parametri del metodo sono quelli della costruzione in questione, ad esempio:

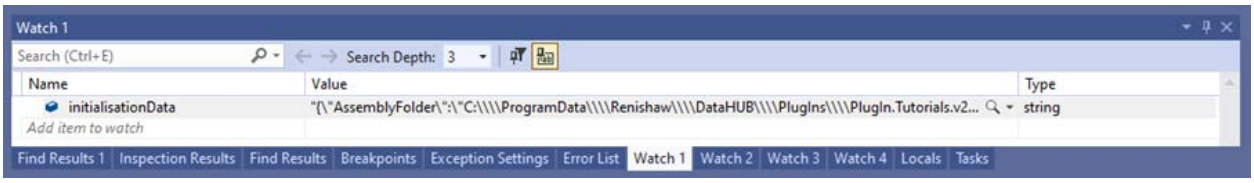


Figura 36 Valori dei parametri del metodo Initialise

Da questo momento in poi, i dati esistenti (e, nel caso della costruzione in corso, i nuovi dati che arrivano) vengono elaborato e il metodo *Process* dell’istanza viene richiamato con i valori dei parametri di LaserVIEW e MeltVIEW associati a uno strato:

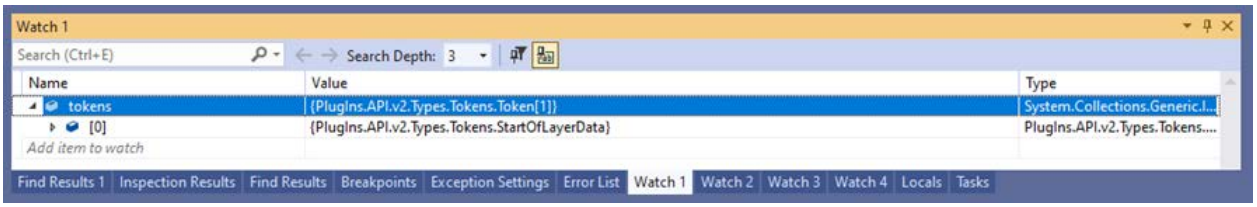


Figura 37 Valori dei parametri del metodo Process

6.4.4.15 Distribuzione del plugin per la produzione

Durante la preparazione del plugin in un DCPC di produzione, è fondamentale compilare, distribuire localmente e testare una costruzione di *rilascio*. Dopo che la funzionalità del plugin è stata convalidata e verificata, può essere distribuita nei DCPC di produzione.

Il processo di distribuzione di un DCPC di produzione è molto simile a quello usato per il PC di sviluppo. La cartella *Release* deve essere copiata nella sottocartella *PlugIns* della cartella dati di DataHUB (<$PROGRAMDATA$\Renishaw\DataHUB\PlugIns>) nel DCPC e rinominata in modo adeguato al plugin. Il servizio DataHUB deve essere quindi riavviato per registrare il nuovo plugin: a tale scopo, usare l’applicazione *Servizi*. Il successo o meno della registrazione del nuovo plugin viene riportata nel registro delle verifiche.

NOTA: DataHUB è stato sviluppato per recuperare tutte le attività di elaborazione dati in corso dopo il riavvio. Per questo motivo, non è necessario attendere il completamento di tutte le attività prima di riavviare. Tuttavia, solo le attività di elaborazione dei nuovi dati utilizzeranno il plugin appena registrato.

6.4.4.16 Diagnosi degli errori dei plugin durante la produzione

In un ambiente di produzione, DataHUB Monitor mostra (insieme ad altre attività di elaborazione della costruzione) lo stato di elaborazione del plugin:

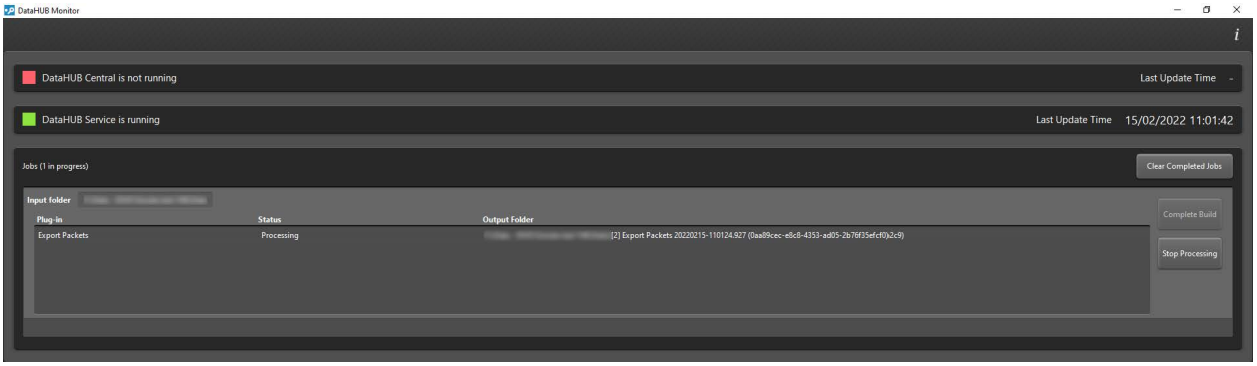


Figura 38 Esecuzione di una costruzione con plugin

Se lo stato di un plugin è *Errore*, il registro delle verifiche conterrà i record dell'errore, come voci aggiunte da DataHUB (ad esempio, un errore di caricamento dell'assemblaggio plugin a causa di dipendenze mancanti) oppure inserite dal plugin tramite i contenuti del valore di ritorno dai metodi *Initialise*, Processo *Shutdown* del plugin.

6.4.5 Esempio 1 – Elaborazione di un flusso di token

Questo primo esempio di codice mostra come elaborare un flusso di token per estrarre i valori minimi e massimi di ciascun canale dei dati di monitoraggio del processo, per ciascun laser e per ciascuno strato. Verranno presi in esame diversi concetti:

- Decodifica dei dati di inizializzazione
- Elaborazione dei token
- Cartella di output dell'istanza

6.4.5.1 Creazione

L'implementazione minima del plugin è la seguente:

```
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();
   }
}
```

Le classi `InitialiseReturns`, `ProcessReturns` e `ShutdownReturns` dell'helper dell'API rappresentano un metodo comodo per generare valori di ritorno dell'API. In questo caso, i metodi Success restituiscono una stringa "Success" standard, riconosciuta da DataHUB.

6.4.5.2 Inizializzazione

Quando DataHUB inizializza questo plugin, la prima attività consiste nel salvare la posizione della cartella di output univoca dell'istanza del plugin per un utilizzo successivo, durante la scrittura del file. Il metodo decodifica (tramite la libreria Newtonsoft.Json) la stringa `initialisationData` ed estrae il percorso della cartella di output:

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

NOTA: i dati di inizializzazione sono codificati come stringa in formato JSON.

6.4.5.3 Elaborazione del flusso di token

Al termine dell'inizializzazione, l'istanza del plugin riceve il flusso di token della costruzione. Alcune serie di dati di monitoraggio del processo potrebbero avere grandi dimensioni. Per questo motivo, il flusso di token viene consegnato in batch. Ogni volta che viene richiamato `Process`, si riceve un nuovo batch del flusso.

Ora è possibile aggiungere la gestione dei token `Sample` al metodo `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);
…
```

Alcuni campi sono inizializzati e salvano i valori minimi e massimi correnti di tutte le proprietà dei dati di monitoraggio del processo di un token `Sample`.

Il valore restituito da `Process` è diventato `SuccessWithoutInformation()` che indica a DataHUB che il plugin ha terminato l'elaborazione del batch di token, ma non intende aggiungere un messaggio ai registri di DataHUB. Questo garantisce che i registri di DataHUB non vengano riempiti di inutili messaggi "Success".

NOTA: i flussi di token avvengono in batch. `Process` viene richiamato più volte.

NOTA: non tutte le chiamate a `Process` devono restituire un messaggio da registrare nel servizio DataHUB.

Il valore di ciascuna proprietà dei dati di monitoraggio del processo di un token `Sample` viene utilizzato per aggiornare i valori minimi e massimi correnti. Nota: alcune proprietà usano il tipo “double” (doppio), altre il tipo “integer” (numero intero).

NOTA: un token `Sample` ha le seguenti proprietà dei dati di monitoraggio del processo:

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

6.4.5.4 Scrittura nel file

L’attività finale del plugin è di scrivere i risultati nel file quando il plugin elabora un token `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}";
}
…
```

Il percorso di output del file dei risultati viene creato combinando la cartella di output salvata nella cache durante l'inizializzazione e i numero dello strato e del laser per i token appena elaborati. I valori massimo e minimo di ciascuna proprietà `Sample` vengono scritti nel file e sono poi resettati per prepararsi all'elaborazione dei token dello strato successivo.

Il plugin restituisce un risultato "Success" con un messaggio che sarà verificato da DataHUB.

NOTA: DataHUB assegna a ciascuna istanza di un plugin un percorso univoco per la cartella di output, che dovrebbe essere usato in caso di output persistente.

6.4.5.5 Implementazione finale

Il plugin completo è:

```
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 Filtri

Un *filtro* dell'API è un'unità di codice che elabora un token di input e produce zero o più token. Dopo un'adeguata elaborazione, un filtro può produrre il token di input, produrre nuovi token o anche assorbire il token di input. Tramite questi metodi, un filtro trasforma un flusso di token di input in un nuovo flusso di token di output.

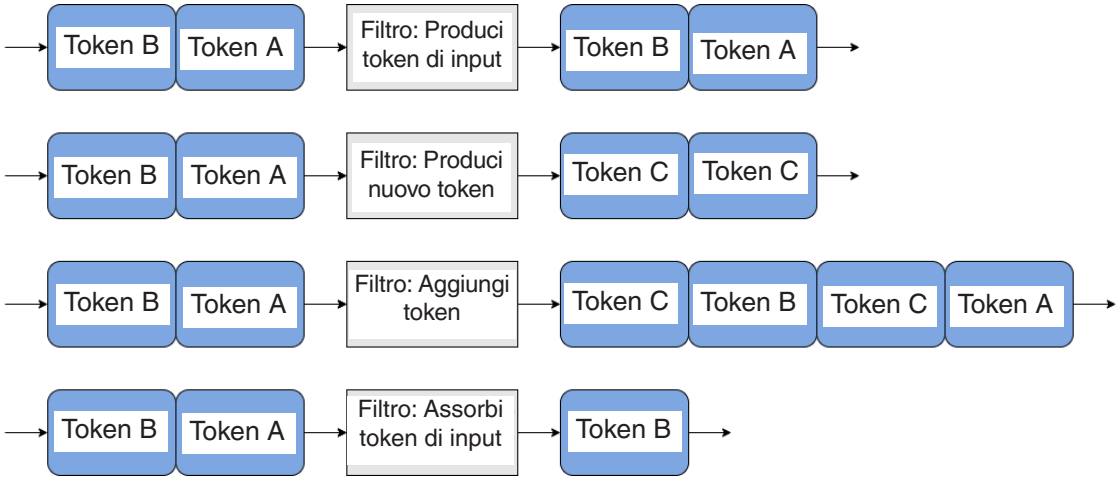


Figura 39 Metodi di output dei filtri

6.4.6.1 Pipeline

Se si hanno due filtri, l'API fornisce un metodo di combinarli in una *pipeline*. Il fatto che una pipeline abbia la stessa firma di un filtro singolo indica che pipeline e filtri possono essere combinati in pipeline più lunghe e complesse. Quando una pipeline elabora un flusso di token, l'output del primo filtro diventa l'input del secondo e così via fino a quando l'output dell'ultimo filtro non forma l'output finale della pipeline.

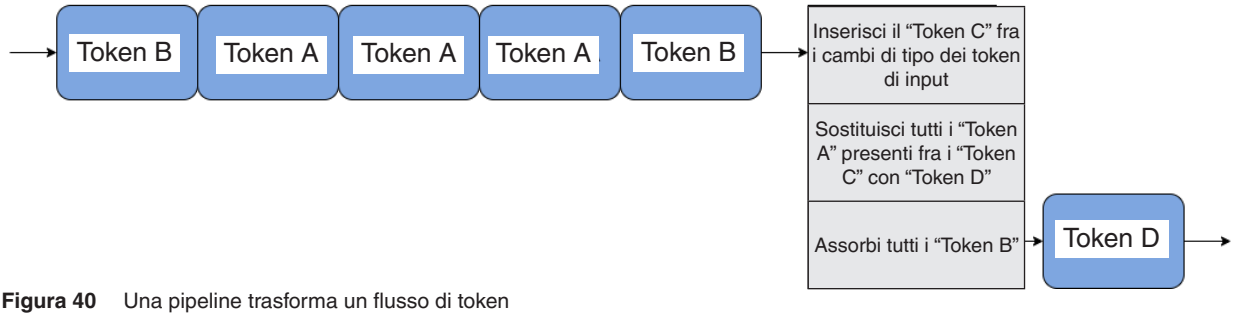


Figura 40 Una pipeline trasforma un flusso di token

6.4.6.2 Riepilogo dei campioni in pacchetti con una pipeline di filtri

In questa sezione viene descritta la trasformazione di un flusso di token di input di campioni in un flusso di token di output di dati riepilogativi, utilizzando varie fasi di filtraggio.

La pipeline è costituita da questa serie di filtri:

- Split when Laser Modulate changes (Separa quando LaserModulate cambia)
- Emit if laser is firing (Emetti se il laser è acceso)
- Split when DemandX changes (Separa quando DemandX cambia)
- Split when DemandY changes (Separa quando DemandY cambia)
- Collate into packets of samples (Riunisci in pacchetti di campioni)
- Summarise packets of samples (Riepiloga pacchetti di campioni)

6.4.6.2.1 Flusso di token di input

Il flusso di token di input è composto da sette campioni di token:

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

Il funzionamento del laser può essere tracciato attraverso i diversi valori di ciascun campione:

- Il laser inizia nella posizione (10, 10) all'ora 0 e non è acceso
- Il laser si accende all'ora 20
- Il laser si sposta nella posizione (20, 10) all'ora 30
- Il laser si sposta nella posizione (20, 20) all'ora 50
- Il laser si spegne all'ora 70

6.4.6.2.2 Filtro 1 – Separa quando LaserModulate cambia

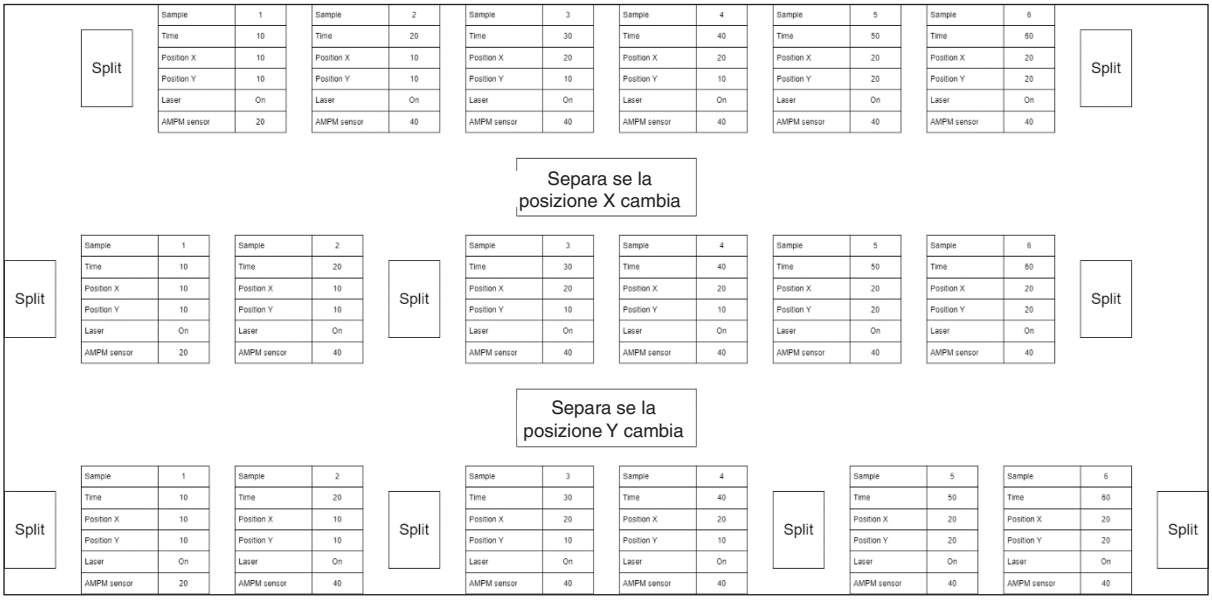
Il primo filtro inserisce token `Split` fra i cambi nel canale LaserModulate:

6.4.6.2.3 Filtro 2 – Emetti se il laser è acceso

Il filtro successivo elimina tutti i campioni nei punti in cui il laser è spento:

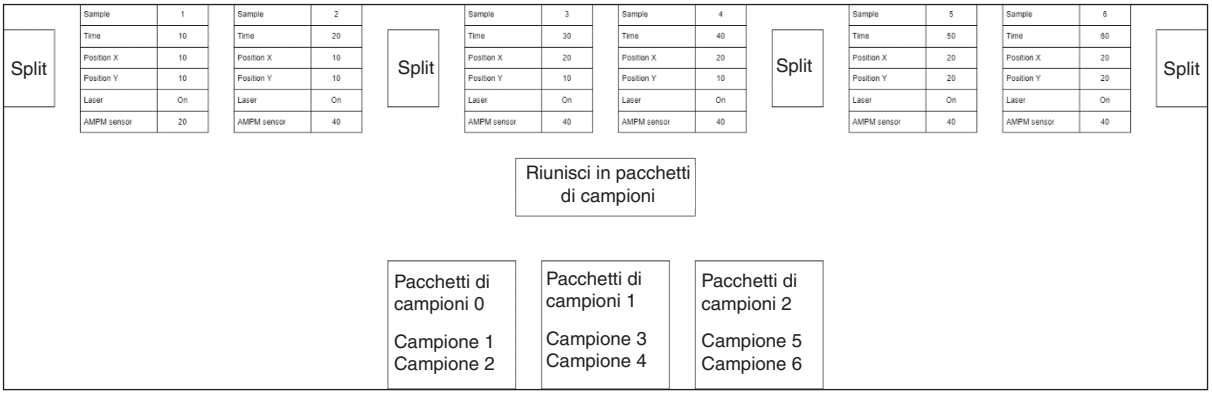
6.4.6.2.4 Filtri 3 e 4 – Separa quando la posizione cambia

I due filtri successivi inseriscono token `Split` fra i campioni nei punti in cui la posizione del laser cambia:



6.4.6.2.5 Filtro 5 – Riunisci in pacchetti di campioni

I campioni sono ora vincolati da token `Split`, fra i quali si trovano i campioni in cui il laser era fermo e acceso (ovvero, i campioni di una singola pozza di fusione). Questi gruppi di campioni vengono quindi riuniti insieme.



6.4.6.2.6 Filtro 6 – Riepilogo dei pacchetti di campioni

Infine, si opera un riepilogo dei campioni riuniti:

Riepiloga pacchetti di campioni

Summary Packet	0
Time	10
Duration	20
Position X	10
Position Y	10
AMPM summary	30

Summary Packet	1
Time	30
Duration	20
Position X	20
Position Y	10
AMPM summary	30

Summary Packet	2
Time	50
Duration	20
Position X	20
Position Y	20
AMPM summary	30

Ognuno dei token del pacchetto riepilogativo contiene valori che descrivono l'ora di inizio (l'ora del primo campione utile), la durata (il numero di campioni utili × la durata di un singolo campione), la posizione e il valori di monitoraggio del processo riepilogativo.

6.4.7 Esempio 2 – Filtri

In questo esempio si estendono i plugin minimo e massimo utilizzando i filtri forniti dall'API. Verranno presi in esame i seguenti concetti chiave:

- Filtri
- Elaborazione dei token tramite filtro

6.4.7.1 Filtri

I filtri elaborano e modificano un flusso di token tramite l'aggiunta, la rimozione, l'inoltro o la modifica dei token. Un filtro elabora un singolo token e produce zero o più eventi. La mappatura di tipo "da uno a molti" fornisce un meccanismo molto efficace per la modifica di un flusso di token durante l'elaborazione.

Molte attività di analisi del monitoraggio del processo prendono in considerazione solo i campioni registrati mentre il laser è acceso (perché contengono dati relativi alla pozza di fusione). L'eliminazione dei campioni registrati mentre il laser era spento riduce anche il numero di token da prendere in esame per l'analisi. Il filtro integrato `EmitSampleIf.LaserModulate.IsOn` è stato studiato proprio per questo motivo. Il plugin può far passare i token `Sample` attraverso questo filtro per rimuovere tutti i campioni acquisiti a laser spento, in modo da produrre valori minimi e massimi utili per la formazione della pozza di fusione.

In C#, un delegato è un tipo che rappresenta un metodo con una firma particolare. Il valore di un delegato viene assegnato con un'*istanza di metodo*. Quel metodo può essere invocato tramite il delegato.

Vengono aggiunti il nuovo namespace `PlugIn.Tutorials.v2.Example2` e la classe del plugin con un delegato `FilterOutLaserOffSamples` che viene assegnato al filtro `EmitSampleIf.LaserModulate.IsOn()` al momento di creare l'istanza del plugin:

```
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();
  }
}
```

Successivamente, viene implementato il metodo Process, utilizzando un'espressione LINQ per passare ciascun batch di token prima attraverso il filtro (tramite il delegato) e poi attraverso un metodo che elabora il flusso di dati modificato:

```
…
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}";
        }
}
```

NOTA: Per brevità, i vari campi (`_demandX` e così via) e i metodi dell'helper (`FormatMinAndMax` e così via) vengono omessi e sono implementati come nel primo esempio.

L'espressione LINQ si compone di due fasi. Innanzitutto, ciascun batch di token viene passato al filtro `FilterOutLaserOffTokens` tramite l'helper API `FilteredBy`. I token del flusso risultante (contenente solo i token `Sample` registrati con il laser acceso) vengono passati a `ProcessToken` che esegue l'elaborazione di ciascun tipo di token che risulti interessante (ovvero `Sample` e `EndOfLayerLaserData`). Le stringhe restituita da `ProcessTokens` vengono raccolte e restituite a DataHUB dall'helper `Success`.

NOTA: la sintassi LINQ di C# è studiata per l'elaborazione dei flussi di dati.

NOTA: i flussi di token possono essere modificati tramite filtro.

NOTA: un filtro restituisce zero o più token per ciascun token di input.

Il plugin popola i file dei risultati con i valori minimo e massimo del canale per i campioni registrati mentre il laser era acceso.

6.4.7.2 Implementazione finale

Il plugin completo è:

```
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 Esempio 3 – ExportPackets

Presi singolarmente, i filtri sono utili, ma il loro pieno potenziale si esprime combinandoli in pipeline (vedere “Pipeline” nella pagina 6-56). Una pipeline svolge un’operazione complessa su un su un flusso di token, tramite una serie di filtri semplici e riutilizzabili. Questo è il lavoro del plugin `ExportPackets`, distribuito con il servizio DataHUB e questo esempio ricrea quell’implementazione, trattando quanto segue:

- Combinazione di filtri in una pipeline.
- Elaborazione di un flusso di token attraverso una pipeline.

6.4.8.1 Combinazione di filtri

L’API del plugin fornisce i due helper `First` e `Then` per aiutare a combinare i filtri in pipeline. Una pipeline viene creata in questo modo:

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

Non c’è un limite al numero di filtri che possono essere concatenati e la pipeline creata si comporta anch’essa come un filtro, mappando un singolo token di input su zero o più token di output.

Il plugin *ExportPackets* distribuito con DataHUB elabora un flusso di token in pacchetti (ciascun pacchetto è un riepilogo dei dati di monitoraggio del processo raccolti per campioni consecutivi con un laser acceso e puntato su una specifica posizione DemandX e DemandY. Per creare pacchetti dai campioni, il flusso di token viene elaborato in questo modo:

1. Separazione del flusso ogni volta che cambia lo stato del laser (acceso/spento).
2. Rimozione di tutti i campioni acquisiti a laser spento.
3. Separazione del flusso ogni volta che cambia la Posizione X.
4. Separazione del flusso ogni volta che cambia la Posizione Y.
5. Raccolta di tutte le serie di campioni separate in pacchetti di campioni.
6. Produzione di valori riepilogativi medi e mediani per ciascun pacchetto di campioni.

NOTA: I campioni del flusso acquisiti a laser spento non vengono eliminati immediatamente, come potrebbe apparire logico, perché nel caso in cui il laser dovesse rimanere nella stessa posizione DemandX e DemandY e accendersi e spegnersi ripetutamente, si potrebbero creare pacchetti errati.

Il codice iniziale per il plugin è:

```
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;
    }
}
```

Il plugin utilizza `StringBuilder` per accumulare i riepiloghi di tutti i pacchetti di campioni. Innanzitutto aggiunge le intestazioni di ciascuna colonna nel file csv.

La pipeline presa in esame in precedenza viene creata e assegnata al delegato `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;
}
…
```

La pipeline consiste di diversi filtri integrati e concatenati in modo che l'output finale dalla pipeline sia una serie di token `SummaryPacket` contenenti valori medi e mediani. Per una panoramica della funzione della pipeline, vedere "Riepilogo dei campioni in pacchetti con una pipeline di filtri" nella pagina 6-57.

NOTA: la concatenazione di filtri semplici forma una pipeline in grado di effettuare la trasformazione complessa di un flusso di token di input.

Il filtro finale, `SummarisePacketOfSamples.Summarise`, viene inizializzato con `SummaryMethods` che assegna un nome al riepilogo – “Media” e “Mediana” – e indica il metodo con cui calcolare il valore. Si tratta di una soluzione flessibile per configurare il modo per riepilogare i pacchetti di campioni: si possono fornire molti metodi di riepilogo, ad esempio, per produrre medie, mediane, minimi e massimi:

```
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)))
```

Il passo finale consiste nell’implementare il metodo *Process* che utilizza il metodo helper `AddSummaryLine` per aggiungere a `StringBuilder` una riga per ciascun token `SummaryPacket`, e scrive il contenuto di `StringBuilder` nel file di output:

```
…
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);
}
…
```

I token `SummaryPacket` vengono aggiunti come righe di testo, con una formattazione adeguata a ciascun valore. Per questa operazione si utilizza `StringBuilder`. Quando si incontra un token `EndOfLayerLaserData`, il plugin scrive i dati del pacchetto riunito in un file della cartella di output.

NOTA: il flusso di token della costruzione può essere elaborato tramite una pipeline predefinita.

6.4.8.2 Implementazione finale

Il plugin completo è:

```
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 Esempio 4 – Restituzione dei dati di serie temporali a Renishaw Central

Nell'esempio di seguito viene spiegato il modo in cui un plugin crea e popola una serie temporale di Renishaw Central con punti dati. La UI Web di Renishaw Central mostra i dati della serie temporale in forma di grafico.

Il plugin di esempio definisce una serie temporale e, per ciascuno strato, aggiunge un punto dati per il valore massimo del canale LaserVIEW. La definizione della serie temporale e i punti dati vengono restituiti a DataHUB tramite i valori di ritorno dei metodi *Initialise* e Process.

Verranno presi in esame i seguenti concetti chiave:

- Creazione di una serie temporale di Renishaw Central
- Aggiunta di dati a una serie temporale di Renishaw Central

Partendo dal plugin vuoto standard:

```
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 Inizializzazione

Quando DataHUB inizializza questo plugin, l'attività prevede la restituzione di una definizione per la serie temporale "LaserVIEW Max". Per la definizione di una serie temporale occorrono un nome, un tipo di unità e un'unità. Si possono definire proprietà aggiuntive, come ad esempio la proprietà Grouping che categorizza la serie temporale in un raggruppamento usato dal front-end Web di Renishaw Central per riunire e gestire la visualizzazione delle serie temporali correlate.

La definizione della serie temporale viene restituita a DataHUB utilizzando il metodo `InitialiseReturns SuccessAndCreateTimeSeries()`. DataHUB converte il valore restituito in messaggi inviati a Renishaw Central con i dati necessari per creare la serie temporale, pronta per essere popolata con punti dati:

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

NOTA: Il metodo Initialise del plugin potrebbe restituire uno o più definizioni di serie temporali.

NOTA: DataHUB gestisce i dettagli dell'invio di dati a Renishaw Central e i plugin possono focalizzarsi sulle definizioni delle serie temporali.

6.4.9.2 Elaborazione

Durante l'elaborazione di ciascun token `Sample` token del flusso di token, il valore massimo corrente viene posto a confronto con il valore del canale LaserVIEW del campione e, se necessario, aggiornato:

```
…
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);
…
```

Al termine dei dati laser per lo strato in esame,quando il plugin riceve il token `EndOfLayerLaserData`, è possibile restituire a DataHUB un messaggio "Success" per indicare che i dati dello strato e il laser sono stati elaborati:

```
…
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();
}
…
```

Al termine dei dati dello strato, il plugin ricever un token `EndOfLayerData`. L'oggetto `TimeSeries` viene utilizzato per produrre un oggetto `UpdateTimeSeries` con un valore serie temporale contenente il valore massimo finale dello strato. Gli helper `ProcessReturns` forniscono un metodo `SuccessAndSendData()` che prende un insieme di oggetti `UpdateTimeSeries`.

Il valore massimo memorizzato nella cache per lo strato viene resettato, ed è pronto per il batch di token successivo. L'aggiornamento della serie temporale restituisce:

```
…
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 Implementazione finale

Il plugin completo è:

```
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 Esempio 5 – Creazione di avvisi in Renishaw Central

In questo esempio viene spiegato come creare avvisi in Renishaw Central quando alcuni valori superano le soglie predefinite.

Verranno presi in esame i seguenti concetti chiave:

- Decodifica dei dati di inizializzazione
- Creazione di avvisi

6.4.10.1 Configurazione del lavoro del plugin con dati di inizializzazione

Nell'esempio, le soglie oltre le quali vengono creati avvisi sono controllate tramite i dati di inizializzazione del plugin. I dati di inizializzazione che corrispondono alla struttura seguente vengono immessi nella schermata "Specify Plug-in Initialisation Data" (Specifica dati di inizializzazione plugin).

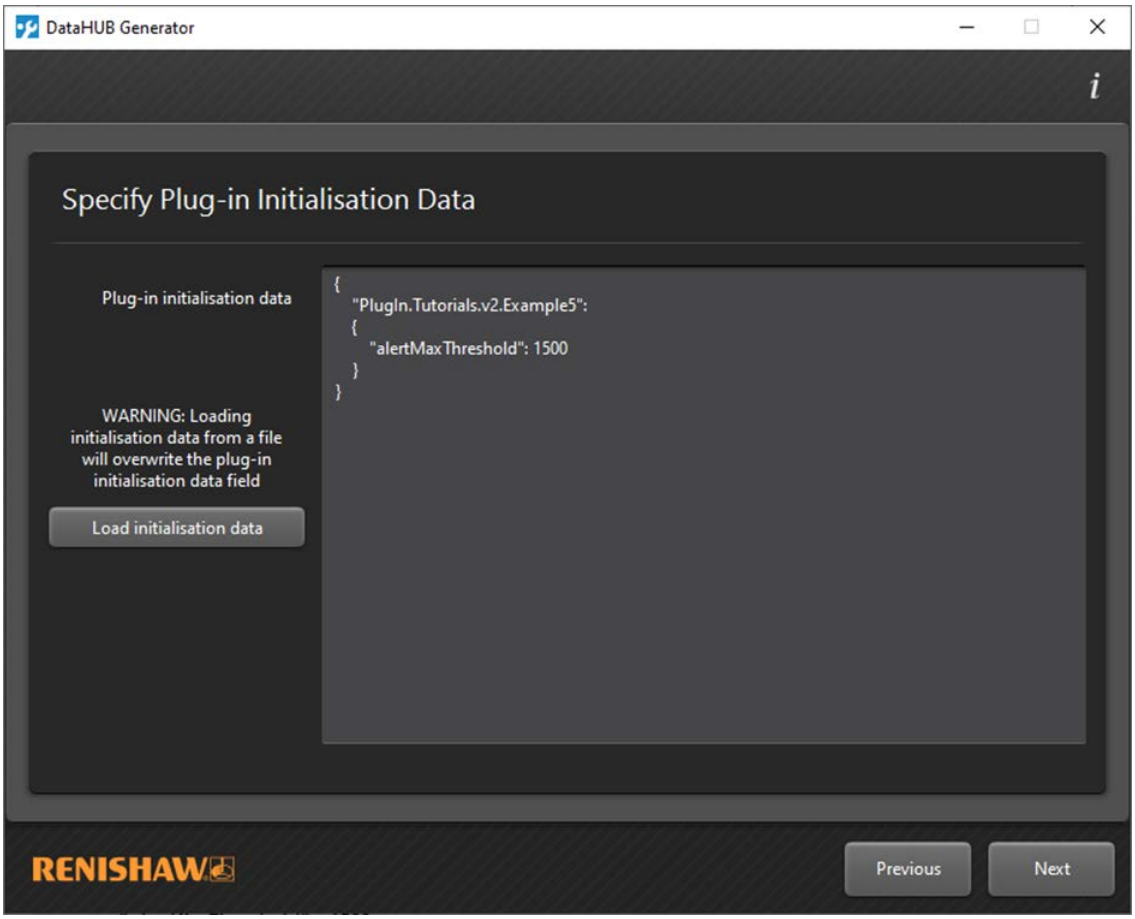


Figura 41 Voce dei dati di inizializzazione del plugin in DataHUB Generator

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

Partendo dal plugin vuoto standard:

```
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 Inizializzazione

Durante l'inizializzazione, è possibile accedere ai dati di inizializzazione specifici del plugin dall'interno dell'oggetto JSON "initialisationData" con il nome fornito al momento di configurare il lavoro. In questo esempio, "PlugIn.Tutorials.v2.Example5".

```
…
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;
…
```

NOTA: la stringa di inizializzazione del plugin specificata al momento di impostare l'attività di elaborazione in DataHUB Generator è disponibile per il plugin durante l'inizializzazione.

6.4.10.3 Elaborazione

Lo scopo di questo plugin è di generare un avviso in Renishaw Central nel caso in cui il valore massimo di LaserVIEW per lo strato dovesse superare la soglia definita. Per prima cosa, il valore massimo corrente viene salvato e memorizzato insieme all'ora del campione associato. Durante l'elaborazione dei token del campione, il canale "LaserVIEW" viene posto a confronto con il valore massimo corrente, sovrascrivendolo, se necessario. Anche l'ora del campione viene salvata insieme al valore massimo:

```
…
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);
…
```

Al termine dei dati laser per un determinato strato, il plugin riceverà il token `EndOfLayerLaserData`. È possibile restituire un messaggio "Success" a DataHUB per indicare che i dati dello strato/laser sono stati elaborati correttamente:

```
…
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();
}
…
```

Al termine di tutti i campioni dello strato, il plugin riceverà un token `EndOfLayerData`, che indica che non vi sono altri dati per questo strato. Quando si incontra questo token, l'elaborazione può essere finalizzata e i risultati inviati a Renishaw Central.

Se il valore massimo corrente supera la soglia definita, un avviso viene creato e aggiunto all'elenco. L'avviso viene creato con una descrizione, un livello di gravità, un nome e l'indicazione di data/ora.

Gli helper `ProcessReturns` forniscono un metodo `SuccessAndSendData` che prende un messaggio e un insieme di avvisi e codifica i dati per restituirli a 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);
        …
      }
…
```

NOTA: le descrizioni degli avvisi restituiti a DataHUB vengono inoltrate a Renishaw Central, che può attivare azioni quali l'invio di SMS.

6.4.10.4 Implementazione finale

Il plugin completo è:

```
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 per plugin V2.0

Perché un assemblaggio possa essere usato dal servizio DataHUB come plugin V2.0, è necessario implementare le seguenti API pubbliche.

6.4.11.1 Assegnazione di un nome all'assemblaggio

Il nome dell'assemblaggio .NET deve iniziare con "PlugIn" ("PlugIn" seguito da un punto) e deve avere l'estensione "dll".

6.4.11.2 Assegnazione di un nome alla classe del plugin

L'assemblaggio plugin deve definire una classe pubblica denominata "PlugIn".

6.4.11.3 Metodi della classe del plugin

La classe del plugin deve implementare i seguenti metodi pubblici:

```
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 Costruttore senza parametri

Il tipo deve avere un costruttore senza parametri. Se non è stato definito nessun costruttore con parametri, C# lo fornisce automaticamente e non è necessario definirlo in modo esplicito.

NOTA: un plugin non può acquisire risorse all'interno del suo costruttore.

Un'istanza del plugin costruita potrebbe non richiamare il metodo *Shutdown* e queste risorse non potrebbero essere rimosse tempestivamente. *Initialise* è il luogo appropriato per acquisire risorse.

6.4.11.5 Metodo `APIVersion` statico

Il metodo `APIVersion` statico identifica la versione dell'API da usare per la convalida del plugin e il formato dei dati di monitoraggio del processo che verranno elaborati.

Parametri:

Nessuno.

Restituice: `string`

Un plugin che implementa API V2.0 deve restituire il valore letterale "2".

6.4.11.6 Metodo DisplayName statico

Il contenuto della stringa restituita dal metodo `DisplayName` statico viene usato come nome del plugin visualizzato in DataHUB Monitor, al momento di creare la cartella di output del plugin, durante la verifica degli eventi del plugin e così via. Anche se non è necessario, l'uso di un nome univoco aiuta a identificare il plugin quando, ad esempio, vi sono più versioni installate in un DCPC.

Parametri:

Nessuno.

Restituisce: `string`

Il valore letterale da utilizzare per assegnare un nome al plugin in vari DataHUB UX e registri.

6.4.11.7 Metodo `Initialise`

Questo metodo viene richiamato da DataHUB all'inizio di una costruzione, prima che i dati sugli strati siano inviati al plugin. Per questa ragione, passa al plugin dati *invariabili sulla costruzione*. Il plug-in potrebbe utilizzare questo metodo per assegnare le risorse necessarie per la durata dell'istanza, calcolare le costanti della costruzione e altro ancora.

Parametro: `string`

Una stringa contenente un oggetto JSON conforme allo schema seguente:

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

`AssemblyFolder: string`

Il percorso della cartella del gruppo che contiene questo plugin:

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

`OutputFolder: string`

Il percorso della cartella in cui l'istanza del plugin dovrebbe scrivere i file di output. Ogni volta che l'istanza del plugin viene eseguita per un lavoro, viene generata una cartella univoca e specifica all'interno della cartella di output del lavoro. Il nome della cartella verrà costruito utilizzando il nome di visualizzazione del plugin, la versione API, data/ora e GUID, con la struttura seguente:

"\[2] <Nome visualizzazione>.<Data/ora nel formato yyyMMdd-hhmmss.fff> (<GUID>)"

`NumberOfLasers: unsigned integer`

Un valore da 1 a 4, secondo quanto definito dalla configurazione dell'hardware della macchina che crea i dati.

InitialisationData: JSON object

Il contenuto fornito durante la fase “Dati di inizializzazione plugin” del flusso di creazione del lavoro in DataHUB Generator. Per maggiori dettagli sulla struttura e sui modi di utilizzo della stringa, vedere “Stringa dei dati di inizializzazione” nella pagina 6-97.

Restituisce: string

Una stringa contenente un oggetto JSON conforme allo schema seguente:

```
{
   “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
         [Un numero qualsiasi di proprietà personalizzate]
       }
    ]
 }
```

result: string

Indica se il plugin è stato inizializzato correttamente o meno. Deve essere:

- Il valore letterale “Success” per indicare che il plugin è stato inizializzato correttamente e può iniziare l’elaborazione oppure
- Il valore letterale “Failure” per indicare che il plugin ha riscontrato un errore e verrà chiuso senza ricevere dati.

Questa proprietà è obbligatoria.

message: string

Un messaggio contenente dettagli aggiuntivi verrà allegato al risultato e riportato nel registro di verifica. Questa proprietà è facoltativa – se non è presente, nel registro di verifica verrà indicato solo il risultato.

timeSeries: `JSON array`

Un insieme contenente le definizione della serie temporale in cui il plugin pubblicherà i dati durante l'elaborazione. Questa proprietà è facoltativa – se non è presente o se l'insieme è vuoto, DataHUB non crea una serie temporale in Renishaw Central.

timeSeries[n].name: `string`

Il nome della serie temporale. Questa proprietà è obbligatoria.

timeSeries[n].unitType: `string`

La famiglia di unità rappresentata dai dati. Consente la conversione a valle per visualizzare i dati con il tipo di unità preferito. I tipi di unità predefiniti sono:

- Acceleration
- Angle
- Binary
- Distance
- Flowrate
- Humidity
- MicroDistance
- Percentage
- Pressure
- Speed
- Temperature
- Dimensionless

Questa proprietà è obbligatoria.

timeSeries[n].unit: `string`

Per convenzione, Renishaw Central preferisce unità SI, presume che le relazioni di tipo "per" siano indicate con il carattere "/" e formatta le potenze come numeri. Ad esempio, in genere l'accelerazione viene rappresentata come m/s^2. Per la temperatura e i gradi si utilizza il carattere "°" se possibile, ad esempio °C per i gradi Celsius. Quando i valori non hanno una dimensione, utilizzare il valore letterale `Dimensionless`. Questa proprietà è obbligatoria.

timeSeries[n].absoluteOrRelative: `string`

Un'indicazione che aiuta le applicazioni a capire se i valori sono assoluti o in relazione al valore precedente. Deve essere:

- il valore letterale "Absolute" che indica che si tratta di un valore assoluto oppure
- il valore letterale "Relative" che indica che si tratta di un valore relativo al valore precedente

Questa proprietà è facoltativa – se non è presente, verrà inserito il valore "Absolute".

timeSeries[n].component: `string`

Associa la serie temporale a un aspetto distinto di un dispositivo, normalmente un'entità fisica quale una stazione meteo, un utensile o un sistema software. Questa proprietà è facoltativa – se non è presente, verrà inserito il valore "Machine".

`timeSeries[n].description: string`

Una descrizione della serie temporale. Questa proprietà è facoltativa.

`timeSeries[n].displayHintPrecision: unsigned integer`

Un'indicazione che aiuta le applicazioni integrate con Renishaw Central a selezionare la precisione con cui visualizzare i valori della serie temporale. Questa proprietà è facoltativa.

`timeSeries[n].displayHintSampleRate: string`

Un'indicazione che aiuta le applicazioni integrate con Renishaw Central a selezionare la velocità di campionamento da utilizzare. Se omessa, la maggior parte delle app a valle presume che si debba utilizzare "default". Le velocità più note sono:

- Raw
- Default
- Hour
- Day
- Week
- Month

Questa proprietà è facoltativa.

`timeSeries[n].graphType: string`

Un'indicazione che aiuta le applicazioni integrate con Renishaw Central a selezionare il tipo di grafico da utilizzare per la serie temporale. I tipi più comuni sono:

- Bar
- Binary (specifico per visualizzare dati Acceso/Spento)
- Line

Questa proprietà è facoltativa.

`timeSeries[n].grouping: string`

Il gruppo con cui la serie deve essere raccolta. Questa proprietà è facoltativa.

`timeSeries[n].precision: unsigned integer`

Una registrazione della precisione con cui i dati sono stati acquisiti/calcolati. Questa proprietà è facoltativa.

`timeSeries[n]:` *Proprietà aggiuntive*

Il plugin può includere un numero qualsiasi di proprietà aggiuntive che verranno salvate in Renishaw Central e che saranno accessibili tramite software personalizzati. Non possono utilizzare i nomi riservati "machine" o "source".

Se la stringa restituita non è conforme allo schema illustrato in precedenza, verrà trattata come un errore e la stringa completa verrà riportata nel registro delle verifiche.

6.4.11.8 Metodo Process

DataHUB passa le sottosezioni dei dati della costruzione con chiamate sequenziali: il numero complessivo di chiamate varia in base alle dimensioni dei dati. In questo metodo il plugin dovrebbe svolgere la maggior parte del lavoro di analisi.

Parametro: `IReadOnlyCollection<Token>`

Una sezione del flusso di token della costruzione.

Restituice: `string`

Una stringa vuota, contenente “Success” o “Failure” oppure una stringa contenente un oggetto JSON conforme allo schema seguente:

```
{
   “result”:  string
   “message”: string
   “alerts”:
   [
      {
         “description”: string
         “level”:    string
         “name”:    string
         “time”:    unsigned integer
      }
   ],
   “analysisResults”:
   [
      {
         “name”: string
         “time”: unsigned integer
         Un numero qualsiasi di proprietà personalizzate
      }
   ],
   “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

Indica se il plugin ha elaborato i dati in modo corretto. Deve essere

- Il valore letterale “Success” per indicare che il plugin ha eseguito l’elaborazione correttamente oppure
- Il valore letterale “Failure” per indicare che il plugin ha riscontrato un errore. Il plugin rimane attivo e continua a ricevere i dati successivi.

Questa proprietà è obbligatoria.

message: string

Un messaggio contenente dettagli aggiuntivi verrà allegato al risultato e riportato nel registro di verifica. Questa proprietà è facoltativa – se non è presente, nel registro di verifica non verrà riportato nulla.

alerts: JSON Array

Un insieme contenente gli avvisi che il plugin intende pubblicare in Renishaw Central durante l'elaborazione. Questa proprietà è facoltativa – se non è presente o se l'insieme è vuoto, DataHUB non tenterà di pubblicare avvisi in Renishaw Central.

alerts[n].description: string

Una descrizione con dettagli riguardanti l'avviso. Questa proprietà è obbligatoria.

alerts[n].level: string

Uno dei seguenti valori letterali per indicare la gravità dell'avviso:

- Info
- Warning
- Error

Questa proprietà è obbligatoria.

alerts[n].name: string

Il nome dell'avviso. Questa proprietà è obbligatoria.

alerts[n].time: unsigned integer

L'ora in cui si è verificato l'avviso (in microsecondi), in relazione all'ora di inizio dello strato. Questa proprietà è obbligatoria.

analysisResults: JSON Array

Un insieme contenente i risultati dell'analisi che il plugin intende pubblicare in Renishaw Central durante l'elaborazione. Questa proprietà è facoltativa – se non è presente o se l'insieme è vuoto, DataHUB non tenterà di pubblicare i risultati dell'analisi in Renishaw Central.

analysisResults [n].name: string

Il nome del risultato dell'analisi. Questa proprietà è obbligatoria.

analysisResults [n].time: unsigned integer

L'ora in cui è stato creato il risultato dell'analisi (in microsecondi), in relazione all'ora di inizio dello strato. Questa proprietà è obbligatoria.

analysisResults[n] *Proprietà aggiuntive*

Il plugin può includere un numero qualsiasi di proprietà aggiuntive che verranno salvate in Renishaw Central e che saranno accessibili tramite software personalizzati. Tuttavia, queste non potranno utilizzare i seguenti nomi riservati:

- created
- machine
- source
- type

timeSeries: JSON Array

Un insieme contenente i dati della serie temporale che il plugin intende aggiungere alla serie temporale definita nel metodo *Initialise*. Questa proprietà è facoltativa – se non è presente o se l'insieme è vuoto, DataHUB non aggiorna i dati di nessuna serie temporale in Renishaw Central.

timeSeries[n].name: string

Il nome della serie temporale, definito al momento dell'inizializzazione. Questa proprietà è obbligatoria.

timeSeries[n].unitType: string

Il tipo di serie temporale, definito al momento dell'inizializzazione. Questa proprietà è obbligatoria.

timeSeries[n].unit: string

L'unità di tempo della serie temporale, definita al momento dell'inizializzazione. Questa proprietà è obbligatoria.

timeSeries[n].values: JSON Array

Un insieme contenente coppie tempo/valore da aggiungere alla serie temporale. Una serie temporale deve disporre di una proprietà "values" (valori), di una proprietà "limits" (limiti) o entrambe.

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

L'ora a cui corrisponde il valore (in µs), in relazione all'ora di inizio dello strato. Questa proprietà è obbligatoria.

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

Il valore. Questa proprietà è obbligatoria.

timeSeries[n].limits: JSON Array

Un insieme contenente i limiti dei dati. Sono disponibili tre serie di limiti:

1. Operating
2. Warning
3. Display

Le applicazioni integrate con Renishaw Central possono utilizzare i limiti per agevolare la visualizzazione e per innescare azioni quando i punti dati della serie temporale superano tali limiti.

I limiti vengono applicati dal momento definito nella proprietà "time" (ora) e durano fino al momento definito dai limiti aggiunti successivamente. Gli ultimi limiti definiti sono perpetui.

Non è necessario che un plugin fornisca limiti per una serie temporale. Ad esempio, è possibile che i dati non abbiano un intervallo operativo naturale o un particolare intervallo di visualizzazione. Inoltre, non è necessario modificare i limiti impostati. L'opzione è disponibile solo per i casi in cui tale operazione risulti necessaria.

Una serie temporale deve disporre di una proprietà "values" (valori), di una proprietà "limits" (limiti) o entrambe.

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

L'ora a partire dalla quale il limite deve essere applicato (in µs), in relazione all'ora di inizio dello strato. Questa proprietà è obbligatoria.

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

Il valore del limite inferiore di visualizzazione. Questa proprietà è facoltativa – se non è presente, il limite inferiore di visualizzazione non verrà modificato per il valore corrente.

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

Il valore del limite superiore di visualizzazione. Questa proprietà è facoltativa – se non è presente, il limite superiore di visualizzazione non verrà modificato per il valore corrente.

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

Il valore del limite inferiore dell'avviso. Questa proprietà è facoltativa – se non è presente, il limite inferiore dell'avviso non verrà modificato per il valore corrente.

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

Il valore del limite superiore dell'avviso. Questa proprietà è facoltativa – se non è presente, il limite superiore dell'avviso non verrà modificato per il valore corrente.

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

Il valore del limite inferiore di funzionamento. Questa proprietà è facoltativa – se non è presente, il limite inferiore di funzionamento non verrà modificato per il valore corrente.

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

Il valore del limite superiore di funzionamento. Questa proprietà è facoltativa – se non è presente, il limite superiore di funzionamento non verrà modificato per il valore corrente.

Se la stringa restituita non è conforme allo schema illustrato in precedenza, verrà trattata come un errore e la stringa completa verrà riportata nel registro delle verifiche.

6.4.11.9 Metodo *Shutdown*

Il metodo *Shutdown* viene richiamato esattamente una volta, quando il metodo *Initialise* restituisce un errore oppure se non vi sono dati ulteriori da elaborare (perché tutti i dati previsti sono stati elaborati o il lavoro è stato annullato). Dopo la chiamata, il plugin non riceverà altre chiamate di alcun tipo.

In questo metodo il plugin deve eseguire le elaborazioni finali dei dati di costruzione, rilasciare tutte le risorse che potrebbe avere acquisito e così via.

Parametri:

Nessuno.

Restituisce: `string`

Se la stringa contiene “Success”, il plugin si è chiuso correttamente. Altrimenti, il plugin ha riscontrato un errore. In entrambi i casi, la stringa viene riportata per intero nei registri di verifica.

6.4.11.10 Stringa dei dati di inizializzazione

DataHUB inizializza un plugin V2.0 con una stringa formattata JSON contenente valori per i parametri globali della costruzione e la stringa di inizializzazione del plugin specificata nel flusso di lavoro di DataHUB Generator per il lavoro di elaborazione del plugin per la costruzione.

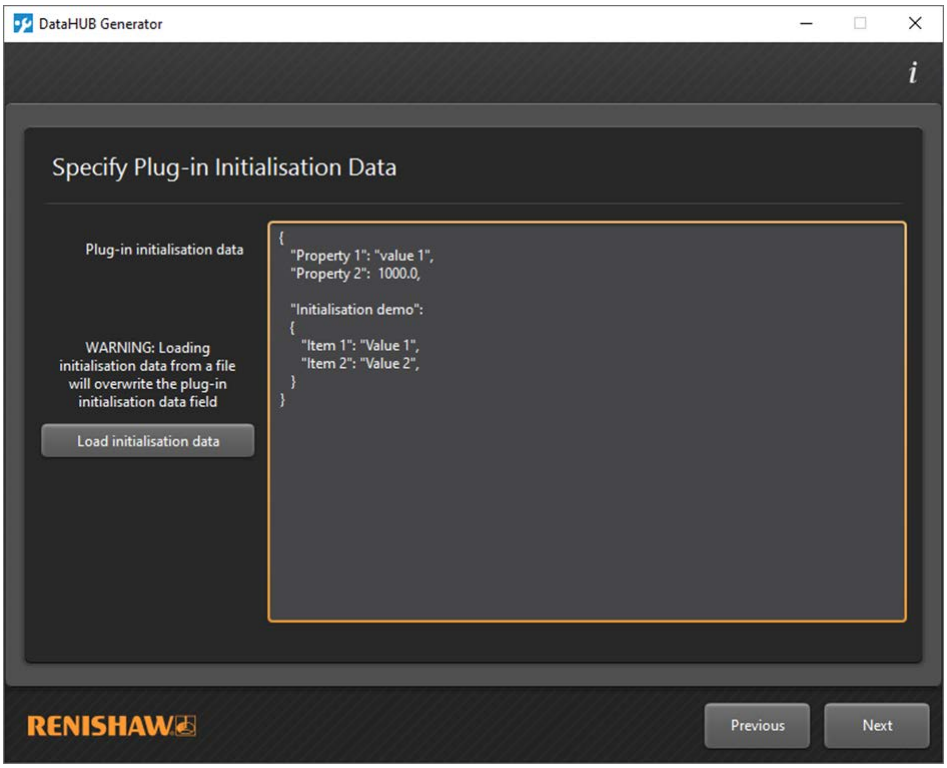


Figura 42 Stringa JSON di input del plugin

La stringa immessa nel flusso di lavoro DataHUB Generator ha già il formato JSON. Con l’aggiunta di proprietà e oggetti, è possibile inviare dati specifici per ciascun tipo di plugin al metodo *Initialise*

dell'istanza.

Di seguito viene proposto un esempio di stringa di inizializzazione del plugin:

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

L'utilizzo nel plugin di una libreria per il parsing JSON (ad esempio, Newtonsoft.Json o System.Text.Json) semplifica il processo di estrazione dei valori contenuti nella stringa di inizializzazione:

```
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"];
```

I dati di inizializzazione dell'istanza, specificati in DataHUB Generator formano una stringa JSON che viene sottoposta a parsing come JObject prima di accedere alle proprietà che contiene. I valori delle proprietà `Property1` e `Property2 e i valori delle` proprietà nell'oggetto demo `Initialisation` nidificato sono visibili nella finestra di VisualStudio:

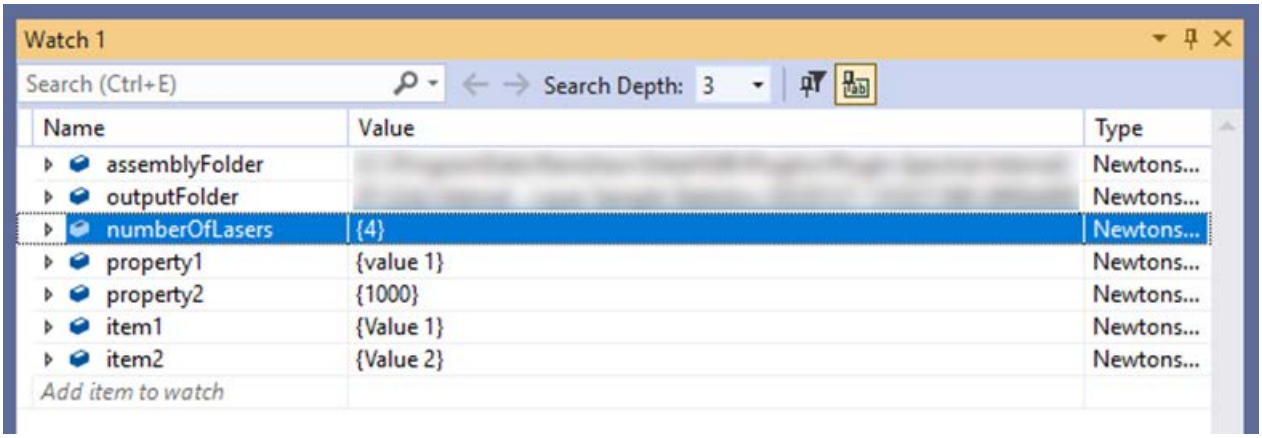


Figura 43 Stringa dei dati di inizializzazione del plugin nella chiamata Initialise

In genere, un DCPC utilizza più plugin, e la strategia consigliata per la formattazione della stringa DataHUB Generator consiste nello specificare un oggetto JSON con un nome diverso per ciascun tipo di plugin:

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

A questo punto, per accedere ai dati di inizializzazione specifici di un plugin basta accedere al relativo oggetto nidificato ovvero nel plugin TypeA e TypeB:

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

6.4.11.11 Tipi di token

6.4.11.11.1 DataSourceInformation

Questo token descrive gli elementi di un’origine dei dati di monitoraggio del processo.

```
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

Questo token segnala la fine di tutti i dati di uno strato.

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

6.4.11.11.3 EndOfLayerLaserData

Questo token indica la fine dei campioni di un laser, per uno strato.

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

6.4.11.11.4 PacketOfSamples

Questo token contiene una raccolta di campioni.

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

6.4.11.11.5 Sample

Un token Sample contiene i dati raccolti in un punto discreto nel tempo, durante il processo di costruzione dello strato. Per uno strato, per un laser, vi saranno molti token 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

Questo token non ha proprietà.

6.4.11.11.7 StartOfLayerData

Questo token indica l'inizio della serie di token relativi a uno strato.

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

6.4.11.11.8 StartOfLayerLaserData

Questo token indica l'inizio della serie di token di un laser, per uno strato.

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

6.4.11.11.9 SummaryPacket

Questo token contiene valori di tipo "Summary" prodotti dai metodi forniti dal filtro `SummarisePacketOfSamples`. La serie di valori "Summary" è contenuta in istanze della classe nidificata `SummaryValues`. Dato che i metodi "Summary" possono produrre risultati con valori frazionari, il tipo di ciascuna proprietà in `SummaryValues` è "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

La classe base da cui derivano tutti i token. Questa classe non contiene dati.

6.4.11.11.11 LaserID e LayerNumber

Il tipo sottostante degli ID laser e dei numeri di strato è “integer” (numero intero), quindi, per evitare che un numero di strato venga disallineato accidentalmente in un ID laser, questi prevedono un sistema di sicurezza durante la costruzione dei vari token indicati sopra.

6.4.11.12 Filtri

L’API fornisce una serie di filtri sviluppati per eseguire attività comuni durante l’elaborazione di un flusso di token.

6.4.11.12.1 EmitSampleOnlyIf

Questo filtro confronta il valore di una delle proprietà di un token `Sample` di input ed emette `Sample` solo se il valore supera il test. Tutti i tipi di token diversi da `Sample` vengono emessi nel flusso di output.

Vengono proposti i seguenti valori di proprietà:

- DemandX
- DemandY
- DemandFocus
- DemandLaserPower
- MeltVIEWPlasma
- MeltVIEWMeltpool
- LaserVIEW
- LaserOutputPower
- LaserBackReflection
- LaserModulate
- I test di uguaglianza sono:
- EqualTo
- NotEqualTo
- GreaterThan
- GreaterThanOrEqualTo
- LessThan

- LessThanOrEqualTo
- CustomTest
- IsOn (solo LaserModulate)

La variante finale del filtro consente la configurazione di un test di confronto personalizzabile. È possibile utilizzare qualsiasi funzione che utilizzi due numeri interi per restituire un valore booleano, ad esempio:

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

Questo test personalizzato fa passare i campioni con un valore DemandX pari.

6.4.11.12.2 SplitWhen

Questo filtro confronta il valore di una delle proprietà di un token `Sample` di input con il valore della proprietà del token precedente. Se i due sono diversi, aggiunge un token `Split` prima di produrre il token di input. In pratica, aggiunge una separazione (`Split`) fra i due token. Tutti i tipi di token diversi da `Sample` vengono emessi nel flusso di output.

Sono disponibili le seguenti varianti:

- DemandXChanges
- DemandYChanges
- DemandFocusChanges
- LaserModulateChanges
- DemandLaserPowerChanges

NOTA: vengono fornite solo le proprietà “Demand”: le altre proprietà di monitoraggio del processo hanno una variabilità elevata, da un campione a un altro e pertanto non risultano idonee per la separazione del flusso di token.

6.4.11.12.3 CollateInto

Questo filtro raccoglie tutti i token `Sample` vincolati da token `Split` in un unico token `PacketOfSamples`. I token `Split` e `Samples` vengono assorbiti. Tutti gli altri tipi di token emessi nel flusso di output.

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

Vale la pena soffermarsi sullo scopo di questa mappatura. Dato un flusso di token precedentemente diviso da token `Split` (ad esempio, se modulazioni laser, posizioni o altro vengono modificati), il flusso di output prodotto non conterrà token `Sample` o `Split`, che saranno stati sostituiti da token `PacketOfSamples`.

6.4.11.12.4 SummarisePacketOfSamples

Questo filtro elabora i token `PacketOfSamples` prodotti da `CollateInto` ed esegue una o più funzioni riepilogative dei valori dei dati campione che sono stati riuniti. Il costruttore di questo filtro permette di configurare una o più funzioni riepilogative con nome. I risultati di ogni funzione vengono salvati nel token di output.

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

Un'istanza SummaryMethod viene costruita con un nome e una funzione riepilogativa:

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

6.4.11.12.5 Pipeline

Questi due helper sono indispensabili per trasformare i filtri in pipeline:

```
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 Helper

L'API fornisce alcune classi che definiscono le operazioni comuni utilizzate durante la scrittura dei plugin.

6.4.11.13.1 PositionHelpers

Questa classe offre funzioni per arrotondare i valori doppi in modo coerente:

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

6.4.11.13.2 Helper delle stringhe di ritorno

L'API fornisce metodi di helper per costruire le stringhe di ritorno di cui DataHUB necessita in tutte le fasi del plugin (*Initialise*, *Process* e *Shutdown*).

NOTA: DataHUB prevede di ricevere una stringa di ritorno JSON valida e conforme all'API della stringa di ritorno. È importante che i caratteri speciali delle stringhe abbiano una barra rovesciata come carattere di escape. In caso contrario, una stringa JSON non valida potrebbe essere inviata a DataHUB e scartata. Ad esempio:

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

Un secondo esempio:

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

6.4.11.13.3 Ritorno di tipo Initialise

Gli helper per la creazione di semplici stringhe di ritorno dei plugin da Initialise a DataHUB sono:

```
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);
```

Per creare una o più serie temporali in Renishaw Central, si possono utilizzare i seguenti helper:

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

È necessario creare un oggetto `TimeSeries` che abbia un nome, un tipo di unità e un'unità. L'oggetto `TimeSeries` fornisce una buona interfaccia con unità e tipi di unità ben conosciuti, e contribuisce a semplificare il processo di costruzione, riducendo i rischi di errore. Vedere l'esempio di seguito:

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

6.4.11.13.4 Ritorno di tipo Process

Gli helper per la creazione di semplici stringhe di ritorno dei plugin da *Process* a DataHUB sono:

```
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);
```

Con i seguenti helper, si possono usare le stringhe di ritorno dei plugin della funzione Process per indicare il successo o l'errore e inviare a Renishaw Central dati quali serie temporali, aggiornamenti, avvisi e risultati dell'analisi.

```
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)
```

`TimeSeries` definite durante la fase di *inizializzazione* possono essere aggiornate con valori, limiti o con entrambi. L'oggetto `TimeSeries` fornisce un metodo Update che restituisce un oggetto `UpdateTimeSeries`, come dimostrato nell'esempio di seguito:

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

Per generare avvisi in Renishaw Central, costruire un oggetto `Alert` e passarlo al metodo `SendData` appropriato dell'helper. Il factory `Alert` fornisce una semplice interfaccia di costruzione, come mostrato nell'esempio di seguito:

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

```
                                Time(350);
```

Infine, per generare risultati dell'analisi in Renishaw Central, costruire un oggetto `AnalysisResult` e passarlo al metodo Send Data appropriato per l'helper. Il factory `AnalysisResult` fornisce una semplice interfaccia di creazione, come mostrato nell'esempio di seguito:

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

6.4.11.13.5 Ritorno di tipo Shutdown

Gli helper per la creazione di semplici stringhe di restituzione plugin da *Shutdown* a DataHUB sono:

```
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 Pacchetti di esportazione dei plugin

Gli hardware LaserVIEW e MeltVIEW raccolgono i dati campionati con una velocità di 100 kHz. Ciascun

campione discreto contiene diversi valori. Questa risoluzione può risultare utile in alcuni casi, ma per la maggior parte delle attività di analisi è sufficiente una rappresentazione dei campioni con una risoluzione minore. Per questo, il plugin quantizza i dati in “pacchetti”, definiti da questi eventi:

- il laser è acceso e
- le posizioni DemandX e DemandY sono costanti.

Questa definizione raccoglie la serie di campioni acquisiti durante una singola esposizione al laser sul letto di polvere. Da ciascuna serie, vengono prodotti i valori riepilogativi medi e mediani dei valori del sensore di monitoraggio del processo di Laser/MeltVIEW. Nota: viste le caratteristiche di funzionamento dei campioni laser, all’inizio di un’esposizione i campioni potrebbero contenere valori Laser/MeltVIEW leggermente bassi. Pertanto, il valore mediano potrebbe fornire un riepilogo migliore dei dati del campione rispetto al valore della media.

6.5.1 Esecuzione del plugin

Per utilizzare questo plugin, eseguire DataHUB Generator e selezionare la cartella *Build Data* contenente i file di monitoraggio del processo della costruzione. Selezionare una cartella per l’output del plugin. Se necessario (ad esempio, se non fosse possibile trovare il file *BuildStarted aaaa.mm.gg.HH.mm.ss.dhxml* con le informazioni sulla costruzione all’interno della cartella di input), assicurarsi che il *numero di laser* impostato sia corretto e appropriato per la costruzione. Il plugin non elabora i dati nel caso di una mancata corrispondenza fra il numero di laser selezionato e quello effettivo utilizzato dall’hardware LaserVIEW/MeltVIEW nella macchina AM che ha prodotto i dati. Gli altri due valori, *Layer Spacing* (Distanza fra gli strati) e *Build Height* (Altezza costruzione) possono essere lasciati come sono, perché il plugin non li richiede.

Immettere una descrizione e premere Next (Avanti).

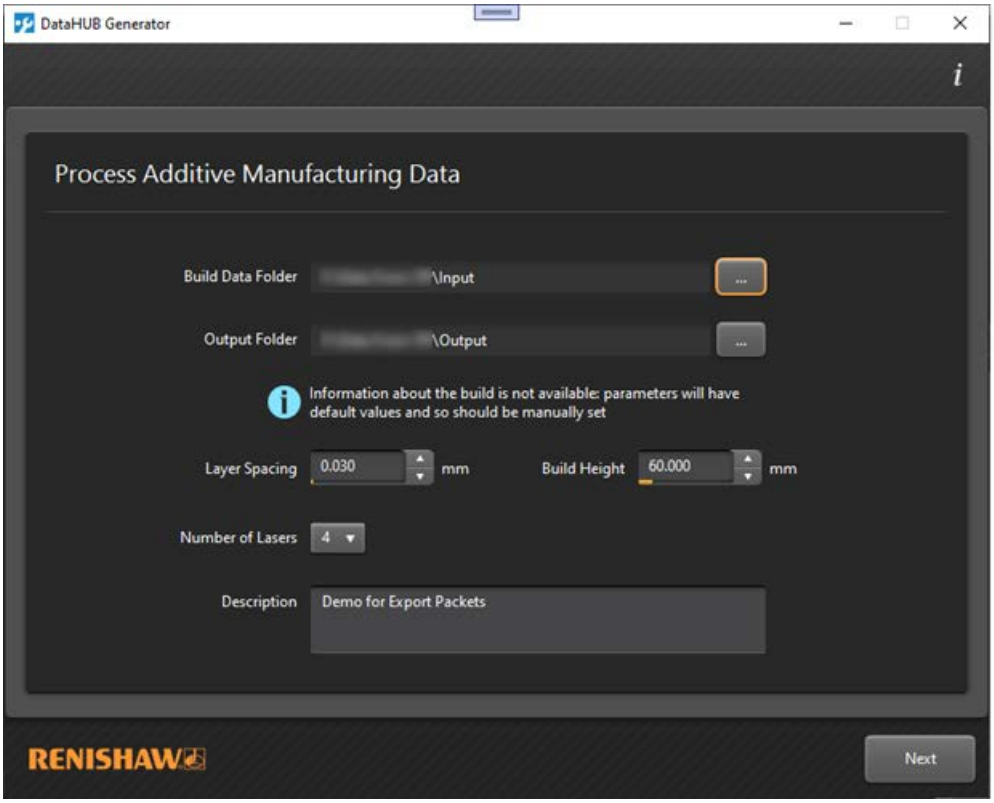


Figura 44 Fase 1 del flusso di lavoro – configurazione dei dati

Nella parte successiva del flusso di lavoro, selezionare “Process LaserVIEW and/or MeltVIEW data via plug-ins” (Elabora dati LaserVIEW e MeltVIEW con i plugin) e premere Next (Avanti).

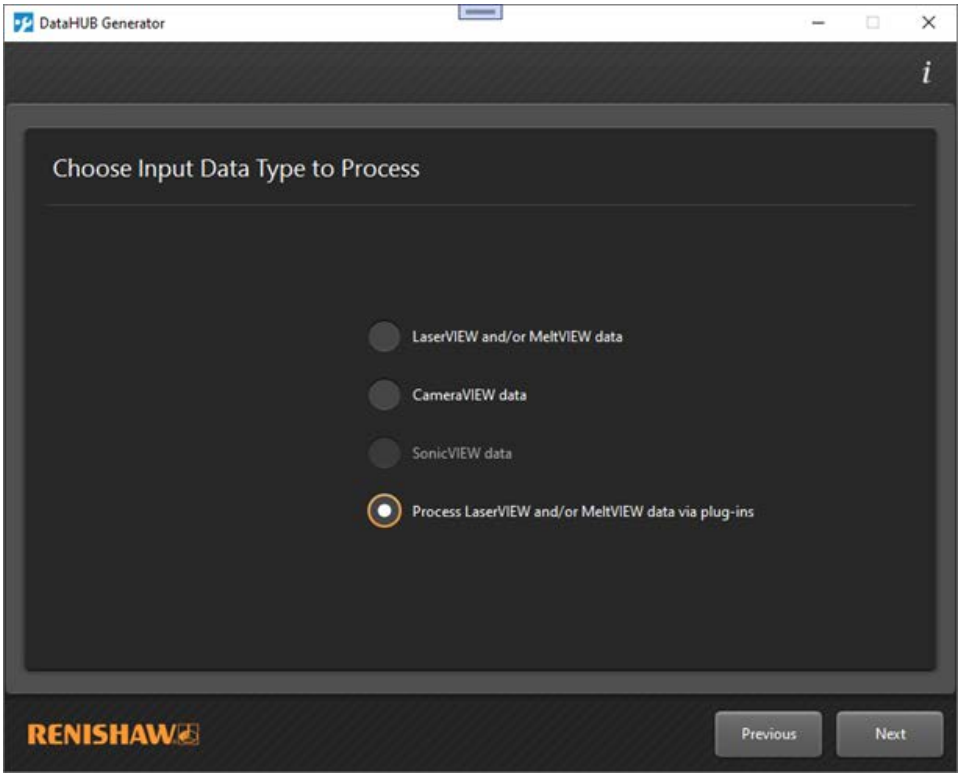


Figura 45 Fase 2 del flusso di lavoro – selezione dell’elaborazione del plugin

Questo plugin non richiede dati di inizializzazione e non è necessario compilare la casella di testo. Premere Next (Avanti).

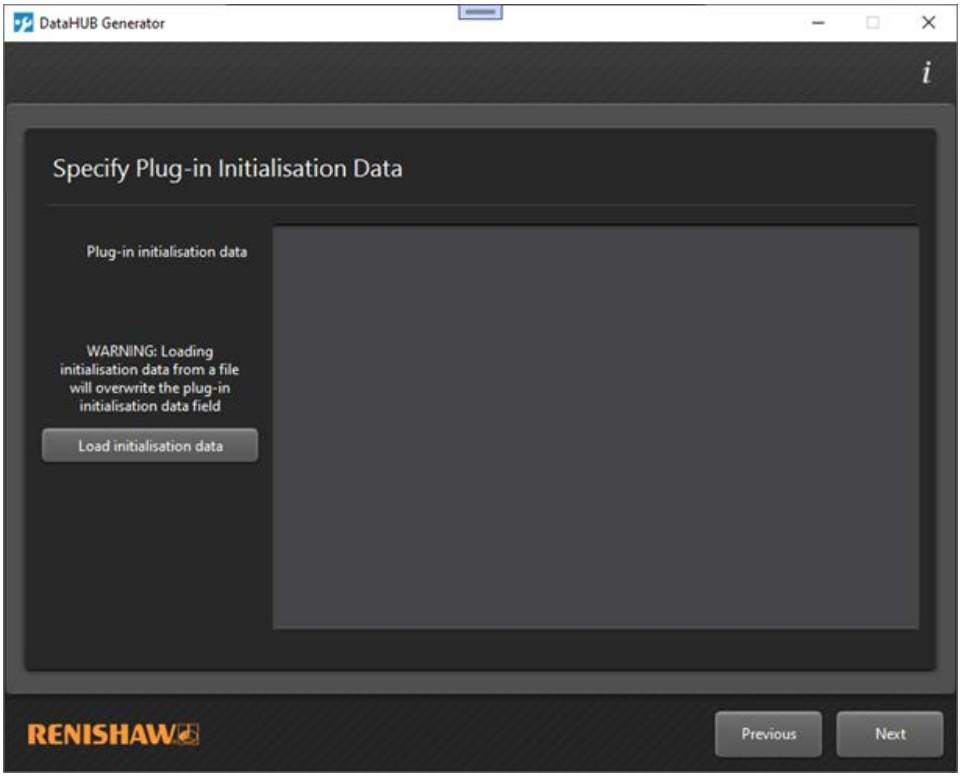


Figura 46 Fase 3 del flusso di lavoro – inizializzazione del plugin

Nella parte finale del flusso di lavoro, premere “Trigger LaserVIEW/MeltVIEW Plug-in Processing” (Avvia elaborazione plugin LaserVIEW/MeltVIEW).

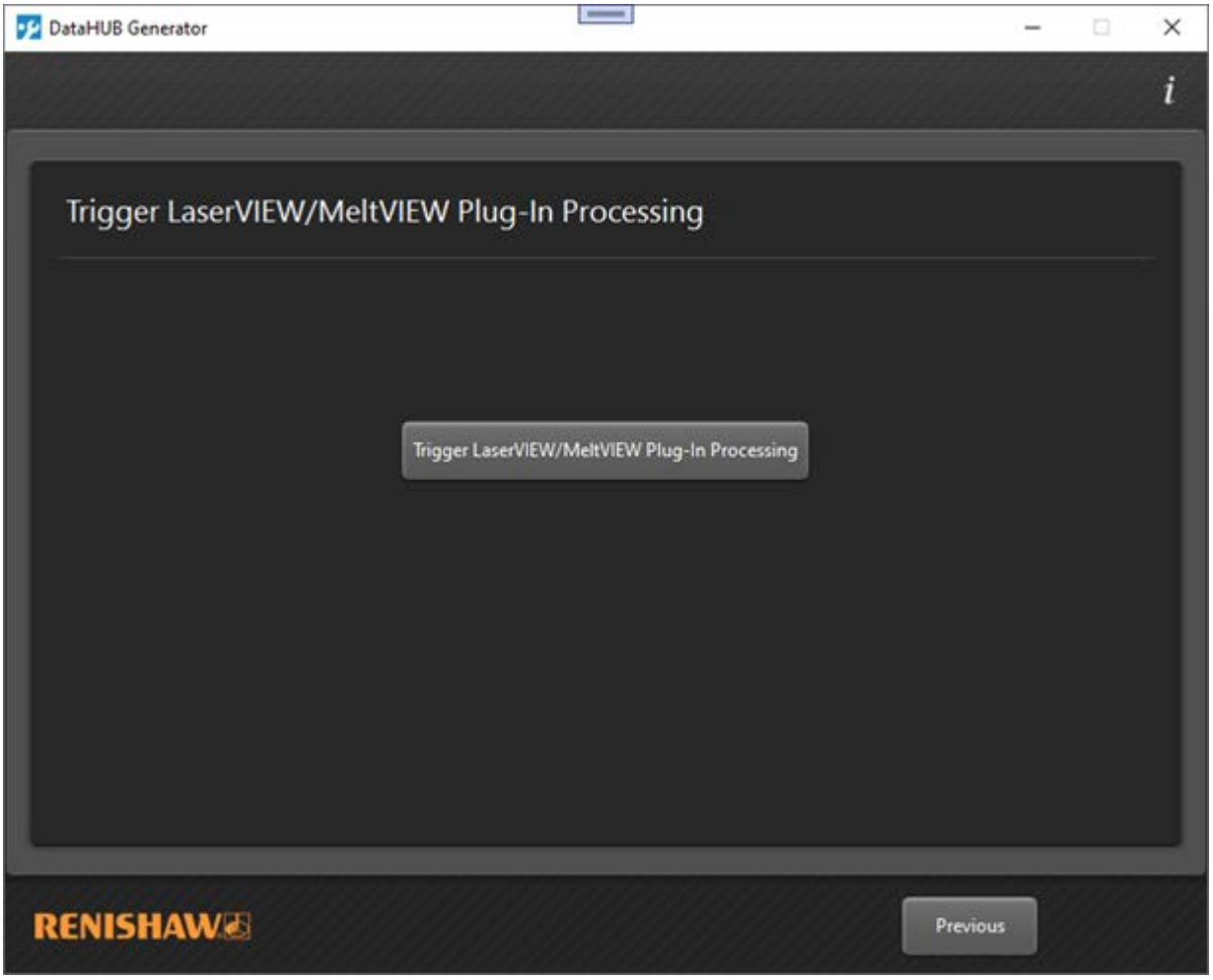


Figura 47 Fase 4 del flusso di lavoro – avvio dell'elaborazione

È possibile uscire dall'app DataHUB Generator.

6.5.2 Nome e posizione del file

La cartella *Output* specificata nel flusso di lavoro rappresenta la cartella principale usata dal plugin per scrivere il proprio output. Il plugin crea una sottocartella con il nome seguente:

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

dove “aaaaMMgg-HHmmss.ff” indica la data e l’ora in cui il plugin ha iniziato l’elaborazione dati e “GUID” è un codice univoco. Questa denominazione è stata selezionata per garantire che se il plugin viene eseguito più volte con gli stessi dati, gli output non vadano a sovrascrivere quelli precedenti.

Nella sottocartella il plugin scrive un file per laser, per strato e gli assegna il nome “Packet data for layer x, laser y.txt”, dove x indica il numero dello strato e y il numero del laser. Nota: anche se il file usa l’estensione “txt”, ha un semplice formato delimitato da tabulazioni e può essere importato in molto pacchetti software senza alcuna rielaborazione.

6.5.3 Contenuto del file

Ogni file inizia con una riga con le intestazioni delle colonne:

- Start Time: indica l’inizio dello strato (µs)
- Duration: la durata, derivata dal tempo di acquisizione del primo e dell’ultimo campione del pacchetto (µs)
- Demand X: la posizione del pacchetto sul letto di polvere (da sinistra –125.0 a destra +125.0 mm)
- Demand Y: la posizione del pacchetto sul letto di polvere (da fronte –125.0 a retro +125.0 mm)
- Demand focus: la messa a fuoco del fascio laser sul letto di polvere (±5.0 mm)
- Demand laser power (mean): la media dei valori dei campioni presenti nel pacchetto
- MeltVIEW Plasma (mean): la media dei valori dei campioni presenti nel pacchetto
- MeltVIEW Melt Pool (mean): la media dei valori dei campioni presenti nel pacchetto
- LaserVIEW (mean): la media dei valori dei campioni presenti nel pacchetto
- Laser back reflection (mean): la media dei valori dei campioni presenti nel pacchetto
- Laser output power (mean): la media dei valori dei campioni presenti nel pacchetto
- Demand laser power (median): il valore mediano dei campioni presenti nel pacchetto
- MeltVIEW plasma (median): il valore mediano dei campioni presenti nel pacchetto
- MeltVIEW melt pool (median): il valore mediano dei campioni presenti nel pacchetto
- LaserVIEW (median): il valore mediano dei campioni presenti nel pacchetto
- Laser back reflection (median): il valore mediano dei campioni presenti nel pacchetto
- Laser output power (median): il valore mediano dei campioni presenti nel pacchetto

Segue poi una serie di righe con il riepilogo dei valori di ciascun pacchetto di dati 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

Pagina lasciata intenzionalmente vuota.

RENISHAW
apply innovation™

www.renishaw.it/contatti

#renishaw

+39 011 966 67 00 **italy@renishaw.com**

Codice: H-5800-6857-01-A
Pubblicato: 10.2023