Prof**. Salvatore Cavalieri**

A.A. **2019/2020**

Autori:

**Castagnolo Giulia** – O55000389

**Fugale Dario** – O55000394

Server opc ua .net cOre

Relazione finale - Progetto di Industrial Informatics

# Overview

**Obiettivo**:Implementare un Server OPC UA in .NET Core che permetta di esporre i dati relativi alle condizioni climatiche di diverse città quali:

* Pressione;
* Temperatura;
* Massima Temperatura Giornaliera;
* Minima Temperatura Giornaliera;

Si è scelto di prendere in esame 3 grandi città siciliane quali: Catania, Palermo, Messina. Tuttavia, la nostra soluzione permette di poter scalare facilmente aggiungendo altre città.

**Data Source:** Le informazioni meteo sono state recuperate sfruttando l’API offerte da *OpenWeatherMap[[1]](#footnote-1)*. La piattaforma mette a disposizione dell’utente, attraverso un piano “free” 60 chiamate giornaliere pertanto, questa soluzione è funzionale soltanto a fini didattici.

**Informazioni sullo stack:** Il progetto è stato realizzato mediante lo Stack di OPC offerto da OPC Foundation sviluppato interamente in *.NET Standard[[2]](#footnote-2)*.

L’intero codice della soluzione è disponibile qui:

<https://github.com/dariofugale95/server-opc-ua-dotnetcore>

# Avvio e Configurazione del Server

## Program.cs

Il main è contenuto nel file . All’interno di esso viene istanziato un oggetto di tipo il quale si occuperà di avviare il server mediante il metodo .

## ServerLauncher.cs

All’interno del metodo vengono effettuati i seguenti steps.

**Caricamento della configurazione:**

In un primo momento vengono caricate le informazioni relative alla configurazione del server. Tutte queste informazioni sono contenute all’interno del file:

In particolare, esso definisce:

* ***SecurityConfiguration***: include anche le security policies accettate per collegarsi con il Server, nel caso specifico sono supportate *Basic256, Sha256Basic256, Basic128Rsa15*;
* ***Path*** ***dove reperire/salvare i certificati Trusted e quelli Rejected;***
* ***TransportConfiguration***: informazioni relative al canale di trasporto come ad esempio lunghezza massima delle stringhe, lunghezza massima dei buffer, channel lifetime ecc;
* ***URI*** ***del Server;***
* ***Path per salvataggio file di log;***
* ***UserTokenPolicies***: nel nostro caso sono supportante *Anonymous*, *Certificate* e *User/Password*.

**Verifica del certificato**:

Dopo aver caricato le informazioni, viene effettuata una verifica del certificato dell’applicazione server nel caso in cui è già presente. Se così non fosse, viene effettuata la creazione di un nuovo certificato.

In questo contesto, il si occupa della gestione nel caso di certificati “*untrusted*”. La sua azione è subordinata dal settaggio del flag all’interno del file xml di configurazione del server. In particolare, se questo flag è impostato a , questa operazione non viene fatta poiché il server sta accettando anche i certificati “*untrusted*”. Ovviamente il flag va messo a solo a scopo di debug ma, in fase di produzione, va necessariamente messo a : accettare dei certificati “*untrusted*” esporrebbe inevitabilmente il sistema a delle grosse problematiche relative alla sicurezza.

**Start del Server:**

Se nelle fasi precedenti non viene sollevata nessuna eccezione, si procede allo start effettivo del server.

L’avvio del server è relativamente semplice da un punto di vista implementativo. Esso consiste nell’istanziare dapprima un oggetto di tipo e quindi invocare del metodo a cui viene passato l’oggetto Server appena istanziato.

## MyOPCServer.cs

Questa classe definisce l’oggetto appena citato. Essendo una classe figlia di , essa eredita i suoi metodi ed attributi. è una classe definita nello stack.

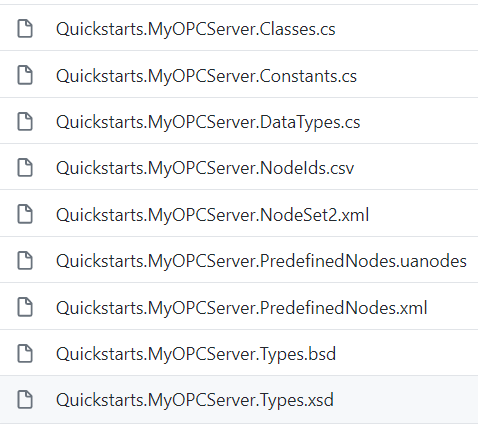
All’interno di è stato fatto *l’override* di alcuni metodi presenti nella classe base. In particolare, qui vengono effettuate le seguenti operazioni:

* **Caricamento delle**  mediante *l’override* del metodo . In questo metodo si definiscono proprietà non modificabili neanche dall’amministratore di sistema poiché identificano il server mediante , , , .
* **Per ognuna delle fasi che attraversa il Server, viene definito un metodo che si occupa di effettuare azioni appropriate**. I metodi usati sono *override* di metodi della classe base.
  + : invoca il metodo il quale si occupa di validare l’identità dell’utente.
  + : istanzia l’oggetto il quale si occupa di notificare e gestire l’evento “cambio di identità dell’utente” mentre la sessione è attiva.
* **Creazione del** (*override*): si istanzia un oggetto di tipo e viene aggiunto nella lista dei . Anche se è possibile istanziare diversi , qui nel nostro esempio ne abbiamo utilizzato solo uno, in maniera tale da gestire un solo spazio degli indirizzi.
* ***User Validation Functions***: in questa “*region*” sono definiti i metodi per validare e gestire l’identità degli user.
  + : si occupa di creare l’oggetto per validare l’identità dell’utente. Ovviamente, tutto ciò viene fatto in accordo con le specifiche di dichiarate nel file di configurazione xml: il metodo riceve come parametro di ingresso la configurazione del server.
  + : è una *callback* per convalidare l’ che viene passato nella sessione. Il parametro di ingresso di questo metodo è proprio la sessione. All’interno viene richiamato il metodo , viene controllato il tipo di ruolo del client e viene richiamato il metodo .
  + : si occupa di validare la password. Questo metodo “entra in gioco” nel momento in cui è prevista, come , un’autenticazione di tipo .
  + : in questo metodo viene verificato se certificato dell’utente è trusted. Ovviamente in caso negativo viene restituita un’eccezione.

# Information Model

Nella soluzione proposta, è stato definito un Information Model “custom”. All’interno di un file .xml (*nuovo\_modello.xml*) sono stati definiti i tipi di dato custom utilizzati nel nostro progetto.

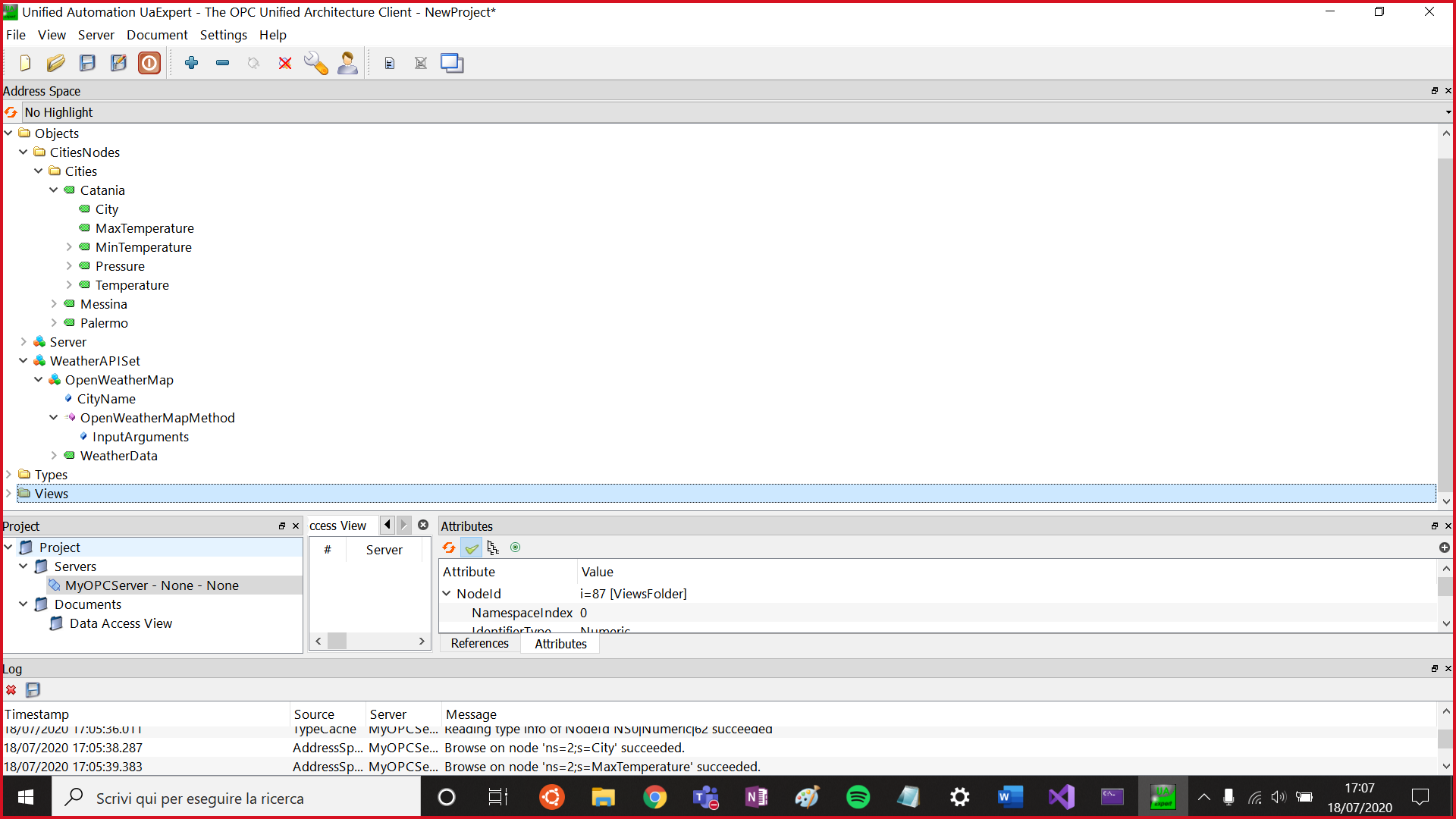
**:** Tale file, presente all’interno della directory , è stato compilato con il tool [[3]](#footnote-3):  Questo strumento si è rivelato utile per effettuare:

* La creazione del file *nuovo\_modell.csv* dove, ad ogni tipo di dato, è stato associato un ID.
* La generazione automatica dei file binari i quali verranno caricati nel nostro progetto in .NET. Questa risorsa contiene i nostri nodi.
* la generazione automatica di altri file contenenti definizione dei tipi, delle costanti (illustrati nella figura sottostante).

Il *Model Design* definito nel file .xml contiene i seguenti tipi:

* – rappresenta il *Data Type* per un dato di tipo analogico. Esso è formato da:
  + – rappresenta il valore effettivo del dato;
  + – rappresenta le informazioni relative all’unità di misura, quindi codice dell’unità di misura, nome ecc;
* – rappresenta il Data Type che contiene i dati meteo realtivi ad una città. In particolare, tale struttura contiene i seguenti campi:

* : è una il cui *DataType* è . Il *BaseType* è . Questa variabile espone due “*children*”: () e ().
* : è un il cui *DataType* . Il *BaseType* è . Questa variabile espone 6 “*children*”: , , , , , .

Il motivo per il quale esponiamo i “children” nelle variabili è perché stiamo utilizzando una modalità ibrida ovvero: visto che abbiamo dati molto complessi (poiché i campi di una struct sono a sua volta delle struct), oltre a inserire il valore nel campo value della variable, esponiamo i singoli campi come *children*, in maniera tale da poter accedere direttamente la valore richiesto piuttosto che recuperarlo dal campo “”. Questo migliora, a nostro parere, la leggibilità del dato, risparmiando anche tempo.

* :
  + (tipo )
  + (tipo )

**:**

* , object di tipo.
* , object di tipo raccoglie tutte le istanze di .

I nodi rappresentanti le città sono di tipo e come esplicato prima questi espongono dei *children*, quindi i dati relativi al meteo saranno presenti in due parti:

* Nel campo di questa variable
* Nel relativo *children*.

Per quanto riguarda l’oggetto esso ha lo scopo di è esporre un nodo generico, che a sua volta espone un metodo. Quest’ultimo, se chiamato settando opportunamente gli input, fornisce dati meteo relativi ad una città che non è esposta.

**Esempio**: I nodi presenti nel nostro address space sono Catania, Messina, Palermo. Questi nodi sono statici, quindi il client può effettuare un monitoraggio e viene costantemente aggiornato con i dati meteo relativi a queste città. Tuttavia, se il Client vuol sapere informazioni relative ad una città che non è esposta, può invocare il metodo presente nell’object e ricevere le informazioni richieste.

## MyOPCServerNodeManager.cs

Questa classe, la quale rappresenta il fulcro di tutto il progetto. Essa è ereditata dalla classe presente nello stack.

In questa classe viene definito l’address space mediante *l’override* del metodo e vengono caricati i nodi (già definiti nel file *nuovo\_modello.xml*) mediante la funzione .

:

questo metodo, già definito nello stack, permette il caricamento del file binario con estensione , nel quale sono presenti i nodi che sono ottenuti mediante la compilazione con il *ModelCompiler* del file *nuovo\_modello.xml.*

:

poiché il nostro server deve essere “*compliant*” allo standard OPC UA, sfruttiamo l’ereditarietà dall’oggetto padre, per richiamare su di esso il metodo , il quale consentirà di esporre tutto *l’information* *model* *standard* nel nostro server. In questo modo abbiamo una *root folder*, una *object folder*, una *type folder* ed una *view folder*.

Dopo aver fatto questo si procede alla definizione del nostro address space, il quale contiene:

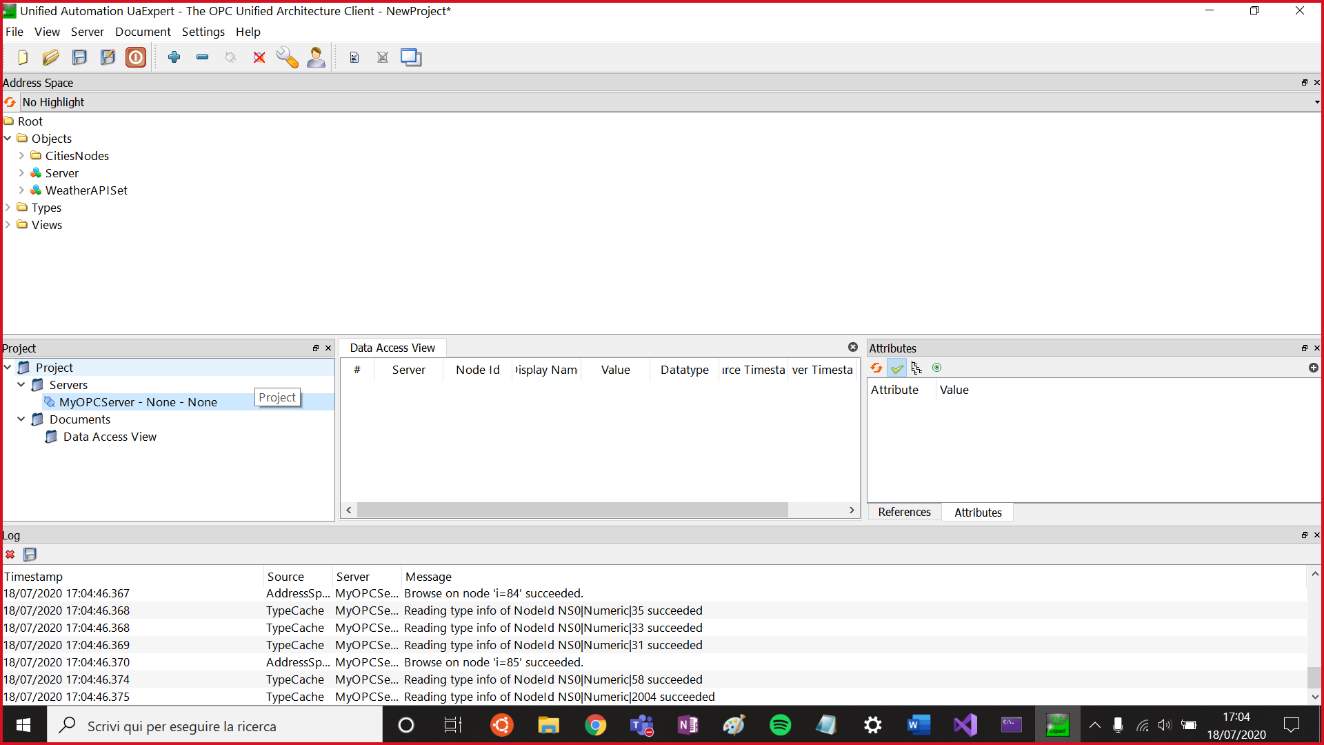
* la folder “”: nodi delle varie città ed i loro *children.*

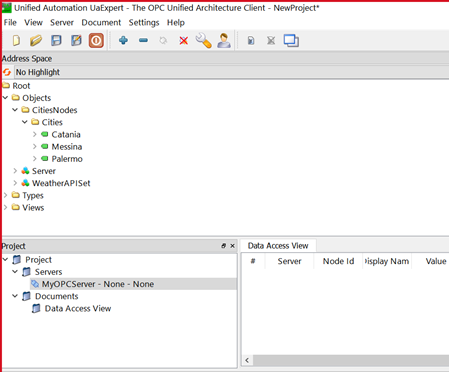
Dentro questo metodo richiamiamo il metodo il quale provvede ad effettuare delle configurazioni, come ad esempio il comportamento del metodo quando questo viene chiamato.

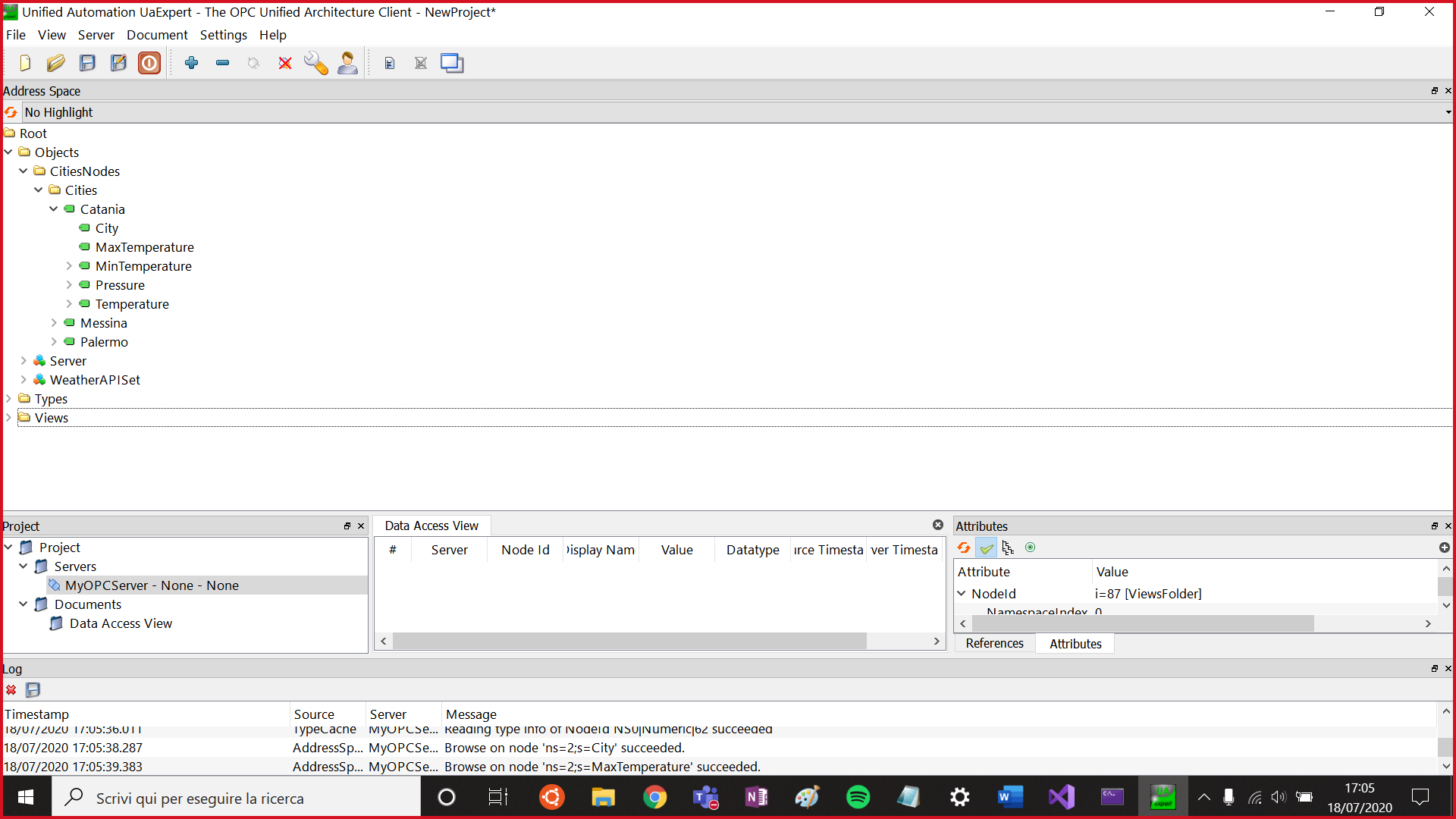
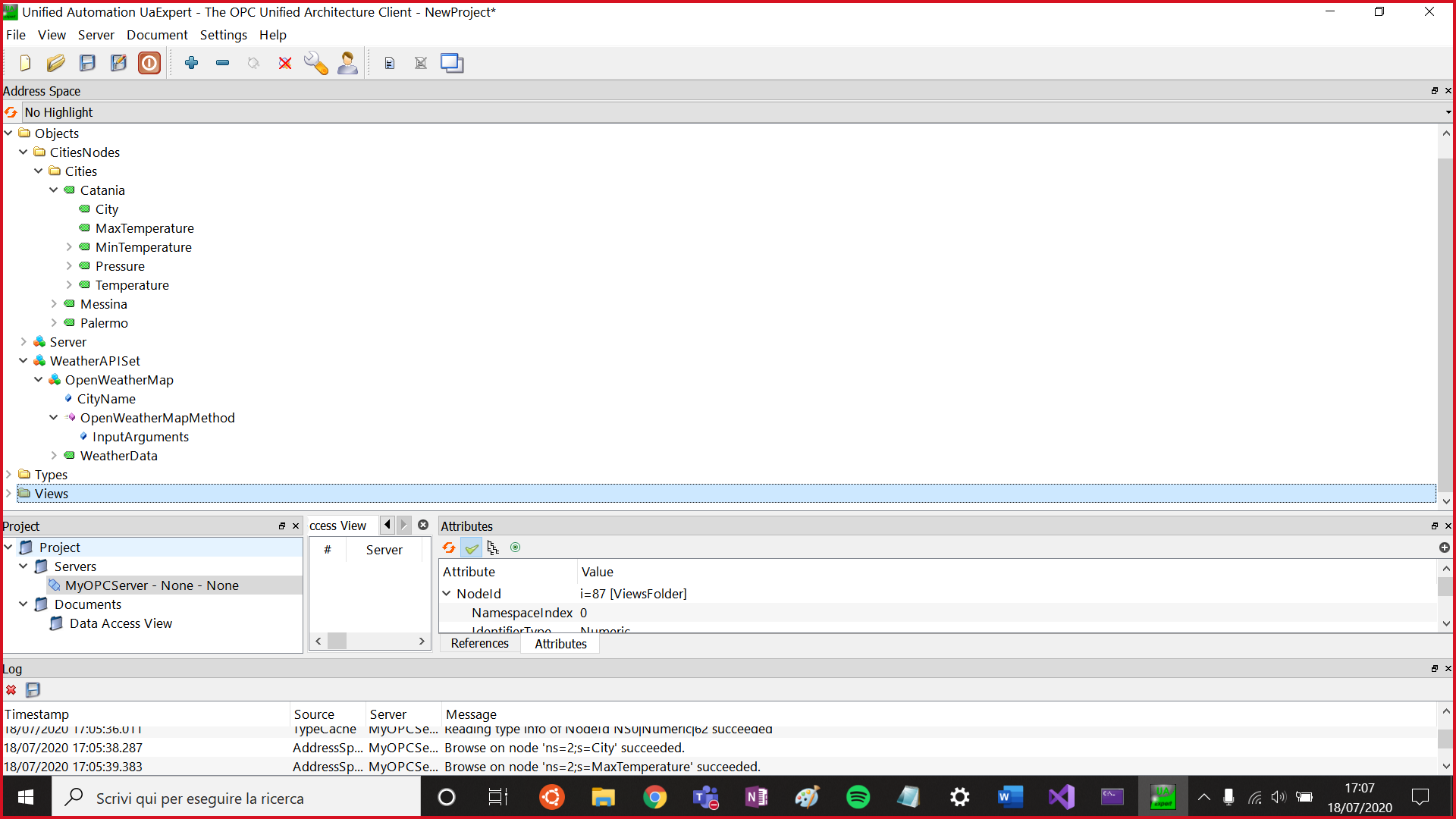
I nodi delle varie città vengono creati mediante la funzione , la quale si occupa di settare tutti i campi previsti per una variable, come ad esempio , , e via dicendo. Il *DataType* dei nostri nodi che rappresentano le città è custom: esso è definito dentro la classe generata automaticamente dal *ModelCompiler* chiamata .

* Il della Variable viene ricavato mediante la funzione la quale effettua la chiamata Rest all’API per ottenere i dati richiesti.
  + **Aggiornamento dei valori:** È importante sottolineare che nel momento in cui viene prodotto un nuovo dato parte un timer, allo scadere del tempo, impostato a circa 20s, verrà invocata la callback che permette di aggiornare i dati nei vari nodi. L’aggiornamento dei nodi prevede anche l’aggiornamento di eventuali nodi figli. La produzione del nuovo dato è delegata alla funzione . Sempre all’interno dello stesso metodo, per ogni nodo padre si chiama la seguente callback la quale lancia un evento per informare del cambiamento valore del nodo padre ed eventualmente dei nodi figli. Questo è necessario per informare i client del cambiamento del valore dei nodi che stanno monitorando.

Questo metodo ci permette di sostituire un nodo Base con un nostro nodo Custom. Questo è necessario per permettere l’esposizione del nostro nodo Custom.

È importante sottolineare che l’object definito nell’xml è l’unico object di tipo custom, poiché nel caso, ad esempio di esso è un .





1. <https://openweathermap.org/> [↑](#footnote-ref-1)
2. <https://github.com/OPCFoundation/UA-.NETStandard> [↑](#footnote-ref-2)
3. [***https://github.com/OPCFoundation/UA-ModelCompiler***](https://github.com/OPCFoundation/UA-ModelCompiler) [↑](#footnote-ref-3)