|  |
| --- |
| Server OPCUA  in .NET Core |
|  |
| Elaborato finale di: Castagnolo Giulia (O55000389) Fugale Dario ( O55000394)    Corso di Industrial Informatics a.a. 19/20  Prof. Salvatore Cavalieri |

# Server OPCUA in .NET Core

|  |
| --- |
| Obiettivo Realizzare un Server OPCUA in .NET Core utilizzando lo Stack di OPC Foundation sviluppato in .NET STANDARD.  Lo scopo del Server è quello di esporre, data una città,nodi contenenti dati relativi a:   * Pressione * Temperatura * Max Temperatura Giornaliera * Min Temperatura Giornaliera   Questi dati, poiché non abbiamo potuto reperire sensori, sono prelevati mediante l’API OpenWeatherMap <https://openweathermap.org/>. |
|  |

1. Introduzione

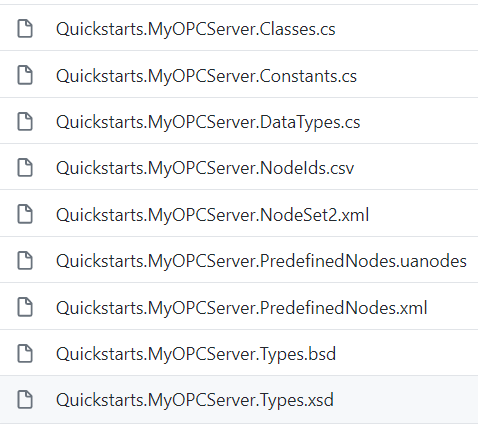
Per realizzare questo server è stato definito un information model custom, ovvero in un file .xml sono stati definiti i tipi di dato custom utilizzati nel nostro progetto.

Il file .xml presente nella repository dentro la directory ModelDesign, è stato compilato con il tool ModelCompiler:

[*https://github.com/OPCFoundation/UA-ModelCompiler*](https://github.com/OPCFoundation/UA-ModelCompiler)*.*

Il tool sopracitato ha permesso:

* Creazione del file .csv dove ad ogni tipo di dato è stato associato un ID
* Creazione dei file binari .uanodes i quali verranno caricati nel nostro progetto in .NET. Questa risorsa contiene i nostri nodi.
* Altri file contenenti definizione dei tipi, delle costanti ecc.

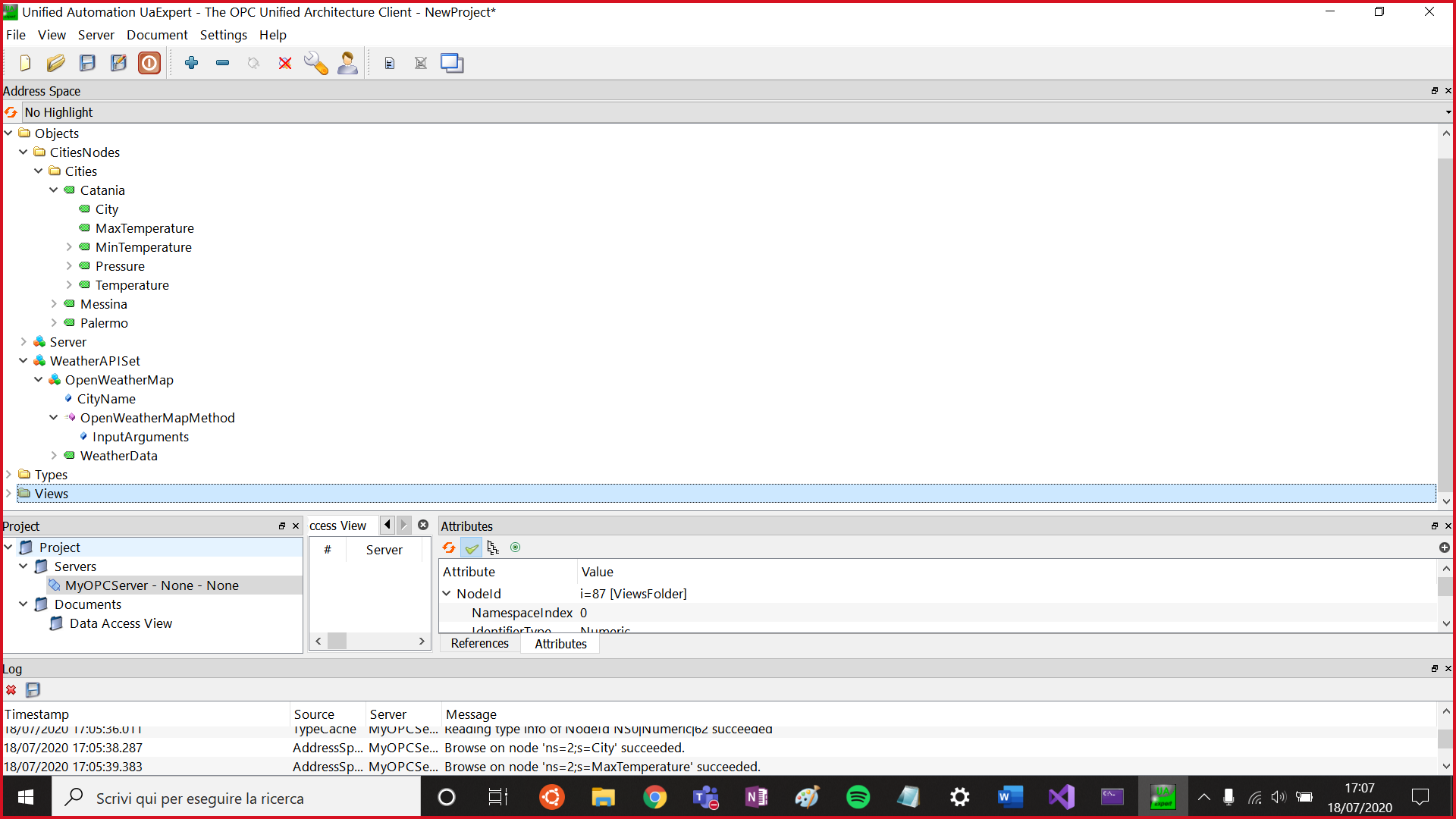


Il Model Design definito nel file .xml contiene i seguenti tipi:

* DataType:
  + AnalogData : struct – rappresenta il Data Type per un dato di tipo analogico
    - Data : float – rappresenta il valore del dato
    - Info: EUInformation – rappresenta le informazioni relative all’unità di misura, quindi codice dell’unità di misura, nome ecc.
  + WeatherData: struct – rappresenta il Data Type che contiene i dati meteo realtivi ad una città
    - Temperature: AnalogData
    - MaxTemperature: AnalogData
    - MinTemperature: AnalogData
    - Pressure: AnalogData
    - CityName: String
* VariableType:
  + AnalogVariableType: è un VariableType con DataType AnalogData, il BaseType è BaseVariableType. Questa variabile espone due “children”: Data ( float) e Info ( EUInformation).
  + WeatherMapVariableType: è un VariableType di DataType WeatherData, il BaseType è BaseVariableType. Questa variabile espone 6 children: temperature, MaxTemperature, MinTemperature, Pressure, Data,City.

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 value. Questo migliora, a nostro parere, la leggibilità del dato, risparmiando anche tempo.



* ObjectType:
  + OpenWeatherMapType:
    - Property: CityName
    - Variable: WeatherData (tipo WeatherMapVariableType)
    - Method: OpenWeatherMapMethod (tipo WeatherMethodType)
* MethodType:
  + WeatherMethodType
    - Input-> City : String
    - Input->MeasureOfTemperature: String
* Object:
  + OpenWeatherMap, object di tipo OpenWeatherMapType.
  + WeatherAPISet, object di tipo BaseObjectType raccoglie tutte le istanze di OpenWeatherMap.

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

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

Per quanto riguarda l’oggetto OpenWeatherMap lo scopo di questo oggetto è esporre un nodo generico, che a sua volta espone un metodo, il quale 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, Enna. 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 OpenWeatherMap e ricevere le informazioni richieste.

1. Avvio e configurazione del Server

Il Program.cs contiene il main nel quale viene istanziato un oggetto ServerLauncher il quale si occuperà di mettere in Run il server.

Invocazione di Run() all’interno di ServerLauncher:

* + caricamento della configurazione del server. Quest’ultima è definita nel file Quickstarts.MyOPCServer.Config.xml e contiene la definizione di:
    - 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 username.
  + Verifica del certificato dell’applicazione server nel caso in cui è già presente, creazione nel caso opposto.
  + Istanziamento di CerficateValidatorEventHandler il quale si occupa gestire i certificati untrusted. Questo è subordinato alla settaggio del flag AutoAcceptUntrustedCertificates all’interno del file xml di configurazione del server. Se questo flag è impostato a True questa operazione non viene fatta poiché il server sta accettando anche i certificati untrusted. Ovviamente il flag va messo a true solo a scopo di debug, ma in fase di produzione va necessariamente messo a false poiché diversamente vi sarebbero problematiche relative alla sicurezza.
  + Start del server se non viene generata nessuna eccezione. Lo start consiste nell’istanziare un oggetto di tipo MyOPCServer e conseguente invocazione del metodo Start a cui viene passato l’oggetto Server appena istanziato.

1. MyOPCServer

Questa classe definisce l’oggetto MyOPCServer ed eredita metodi ed attributi da StandardServer, la quale è definita nello stack. In questa classe è stato fatto l’override di alcuni metodi presenti nella classe base.

Nella classe MyOPCServer si effettuano le seguenti operazioni:

* Caricamento delle ServerProperties mediante l’override del metodo LoadServerProperties(). In questo metodo si definiscono proprietà ~~non modificabili neanche dall’amministratore di sistema~~ poiché identificano il server mediante ManufacturerName, ProductName, ProductUri, SoftwareVersion ecc.
* 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.
  + OnServerStarting(): invoca il metodo CreateUserIdentityValidators() il quale si occupa di validare l’identità dell’utente.
  + OnServerStarted(): istanzia l’oggetto ImpersonateEventHandler il quale si occupa di notificare ~~e gestire l’evento “cambio di identità dell’utente” mentre la sessione è attiva~~ .
* Creazione del MasterNodeManager: override del metodo della classe base. Qui si istanzia un oggetto di tipo MyOPCServerNodeManager e viene aggiunto nella lista dei nodeManagers. Anche se è possibile istanziare diversi node manager nel nostro esempio ne abbiamo utilizzato solo uno, in maniera tal da gestire un solo spazio degli indizzi.
* User Validation Functions: in questa region sono definiti i metodi per validare e gestire l’identità degli user.
  + CreateUserIdentityValidators() si occupa di creare l’oggetto per validare l’identità dell’ utente, ovviamente questo è fatto in accordo con le specifiche di UserTokenPolicies dichiarate nel file di configurazione xml, infatti questo metodo riceve come parametro di ingresso la configurazione del server.
  + SessionManager\_ImpersonateUsers () è una callback per convalidare l’UserIdentityToken che viene passato nella sessione. Il parametro di ingresso di questo metodo è proprio la sessione. All’interno viene richiamato il metodo VerifyPassword(), viene controllato il tipo di ruolo del client ( Administrator ecc) e viene richiamato il metodo VerifyUserTokenCertificate.
    - VerifyPassword() si occupa di validare la password, ovviamente questo metodo entra in gioco nel momento in cui è prevista come UserTokenPolicy l’autenticazione mediate username e password.
    - VerifyUserTokenCertificate() in questo metodo viene verificato se certificato dell’utente è trusted. Ovviamente in caso negativo viene restituita un’eccezione.

1. MyOPCServerNodeManager

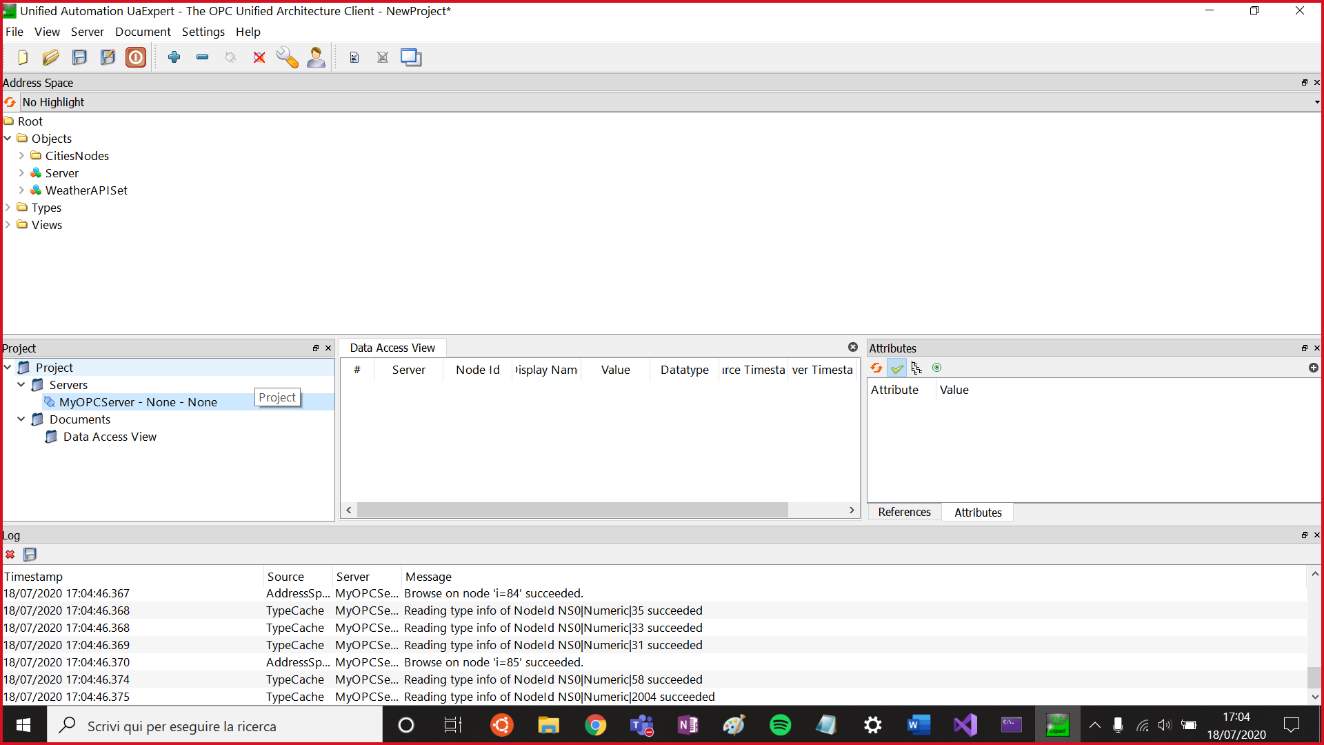
Questa classe, la quale rappresenta il fulcro di tutto il progetto, è ereditata dalla classe CustomNodeManager2 presente nello stack.

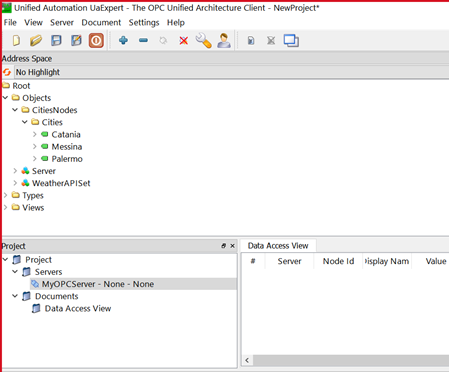
In questa classe viene definito l’address space mediante l’override del metodo CreateAddressSpace e vengono caricati dalla funzione LoadPrefinedNode() nodi già definiti nel model.xml.

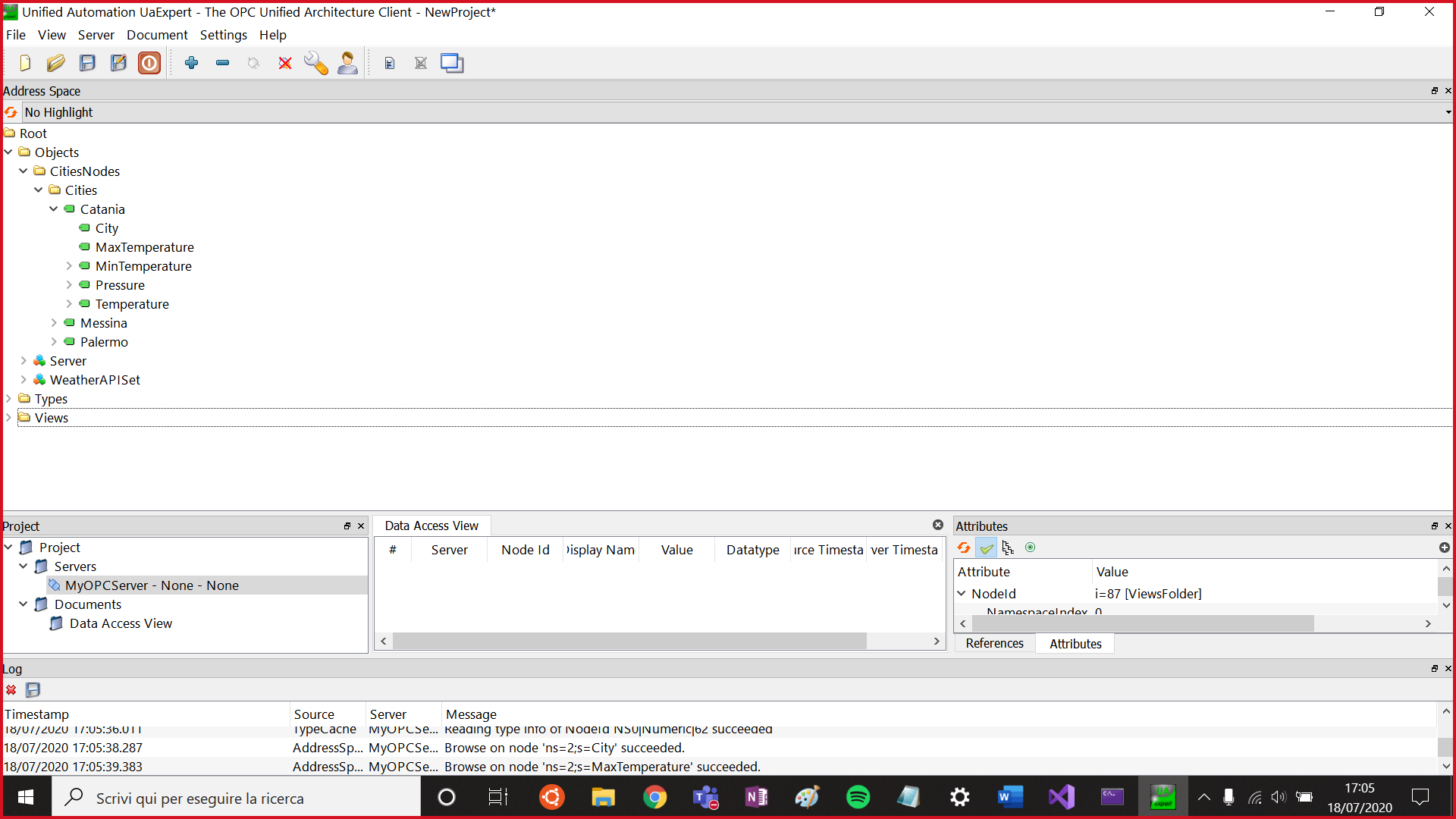
* LoadPrefinedNode(): questo metodo, già definito nello stack, permette il caricamento del file binario con estensione uanodes, nel quale sono presenti i nodi che sono ottenuti mediante la compilazione con il ModelCompiler del file model.xml
* CreateAddressSpace(): poiché il nostro server deve essere compliant allo standard OPCUA sfruttiamo l’ereditarietà dall’oggetto padre, CustomNodeManager2, per richiamare su di esso il metodo CreateAddressSpace, 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 Cities
    - nodi delle varie città ed i loro children

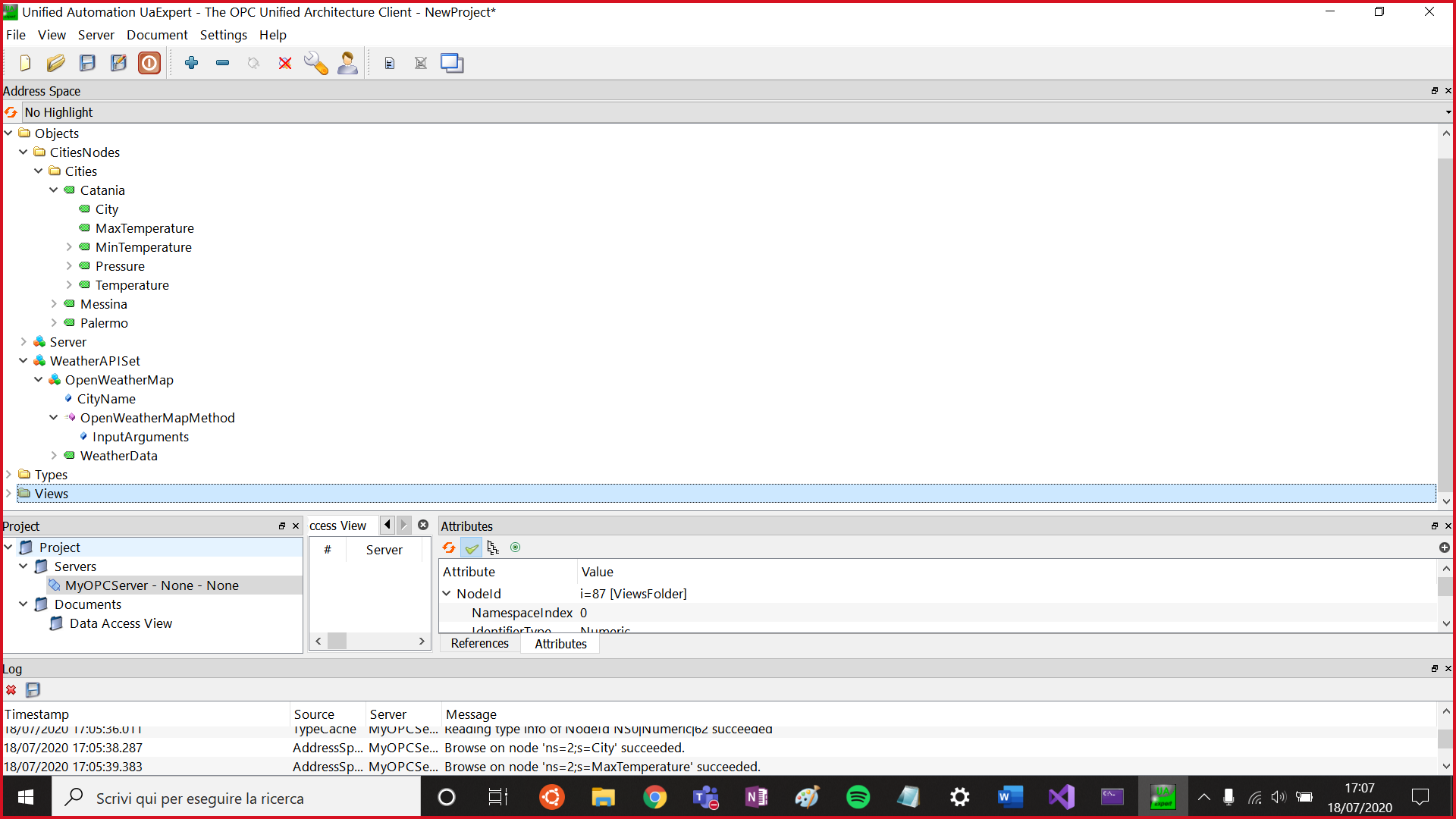
Dentro questo metodo richiamiamo il metodo SetupNodes() il quale provvede ad effettuare delle configurazioni, come ad esempio il comportamento del metodo OpenWeatherMapMethod quando questo viene chiamato.

* I nodi delle varie città vengono creati mediante la funzione CreateVariable, la quale si occupa di settare tutti i campi previsti per una variable, come ad esempio Value, Parent, SymbolicName ecc. Il DataType dei nostri nodi che rappresentano le città è di tipo custom, infatti esso è definito dentro la classe generata automaticamente dal ModelCompiler DataTypeIds.
  + Il Value della Variable viene ricavato mediante la funzione GetNewValue() la quale si occupa di effettuare la chiamata Rest all’API OpenWeatherMap per ottenere i dati richiesti. E’ importante sottolineare che nel momento in cui viene prodotto un nuovo dato parte un timer, allo scadere del tempo, impostato a circa 20s, verrà invocato il metodo OnRaiseSystemEvents che si occuperà 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 UpdateValues(), inoltre in questo metodo per ogni nodo padre si chiama la seguente callback ClearChangeMasks(SystemContext, true) la quale si occupa di lanciare un evento per informare del cambiamento valore del nodo padre ed eventualemente dei nodi figli. Questo è necessario per informare i client del cambiamento del valore dei nodi che stanno monitorando.









In questa classe è stato utilizzato l’override del metodo AddBehaviourToPredefinedNode(), il quale ci permette di sostituire un nodo Base con un nostro nodo Custom. Questo è necessario per permettere l’esposizione del nostro nodo Custom OpenWeatherMap. E’ importante sottolineare che l’object OpenWeatherMap definito nell’xml è l’unico object di tipo custom, poiché nel caso, ad esempio di WeatherApiSet esso è un BaseObject.