netidee PROJEKTE

Open³ Toolbox Entwicklerdokumentation

Dokumentation | Call 15 | Projekt ID 5118

Inhaltsverzeichnis

1 Einleitung

Die Anzahl der verfügbaren Komponenten für ein IoT-System hat sich in den letzten Jahren vervielfacht und steigt weiterhin schnell an. Dies erweitert zwar den Anwendungsbereich, erschwert jedoch die Konfiguration der einzelnen Komponenten. Diese Konfiguration kann aus unserer Sicht durch entsprechendes Vorarbeiten und mittels geeigneter Tools, die entwickelt werden können, vereinfacht werden.

Diese Entwicklerdokumentation gibt einen Überblick über die im Rahmen des Netidee-Projektes „Open^3 Toolbox“ entwickelten Software-Bibliotheken.

1.1 Allgemeines/Struktur und Aufbau

Es wurden eine Vielzahl an Software-Projekten entwickelt (siehe Abbildung 1) die in den folgenden Kapiteln näher beschrieben werden und für zukünftige Entwicklungen als Ausgangsbasis und Nachschlagewerk dienen. In den *Apps*-Projekten befinden sich die Komponenten für das Konfigurationstool. In den *Core*-Projekten befinden sich die Komponenten für die „REST-API“, den „Server“, die „Datenbank“ und das „Gateway“. In den *SensorProjects* befinden sich die Komponenten für die „Sensoren“.

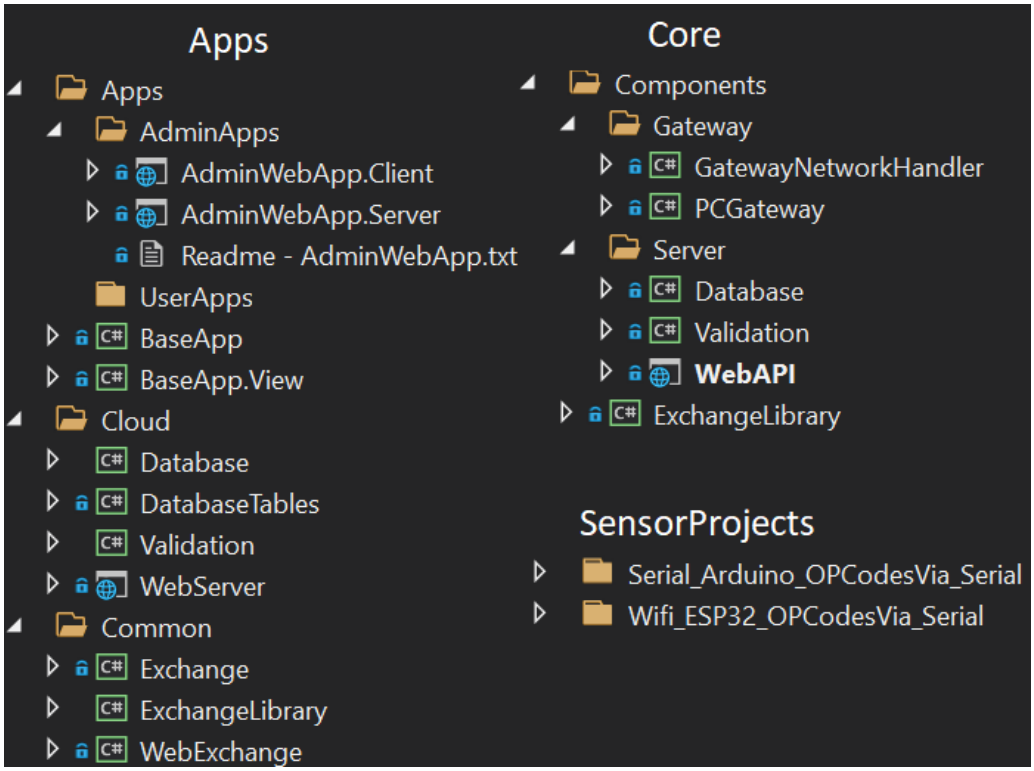

Abbildung 1 Projekt-Struktur

1.2 Source-Code

Der Source-Code befindet sich unter

https://github.com/FotecGmbH/Open3Toolbox

Der Source-Code ist dokumentiert, detaillierte Informationen über einzelne Funktionen kann somit direkt im Source-Code nachgelesen werden.

1.3 Architektur des Gesamtsystems

Das System setzt auf einer baumartigen Architektur auf. Der Stamm bildet dabei eine Server-Applikation, die sämtliche Daten entgegennimmt, speichert und diese auch wieder zur Verfügung stellt. Sensoren liefern die Rohdaten für Messungen. Diese gesammelten Daten werden über eine standardisierte Schnittstelle zur Verfügung gestellt und können anschließend über eine mobile Applikation betrachtet werden. Dieser Aufbau ist in Abbildung 2 Architektur des GesamtsystemsAbbildung 2 dargestellt.

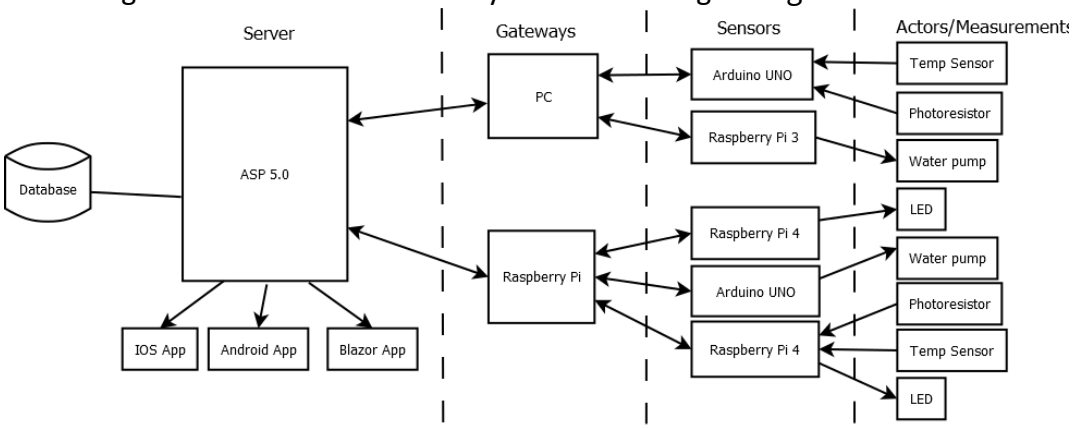

Abbildung 2 Architektur des Gesamtsystems

1 Open³ Toolbox Komponenten

Die Konfiguration der Komponenten werden in Klassen gespeichert und sind unter der Bibliothek „ExchangeLibrary“ im Unterordner „ExchangeForComponents/Sensordata/GeneratedModels/Generated/“ zu finden. Die Ordnerstruktur des Unterordners „ExchangeForComponents“ ist in Abbildung 3 zu sehen. Hierbei werden die Daten als Klassen „Project/Gateway/Sensor/CommunicationInterface/Measurement“ gespeichert und für den Austausch zwischen den Klassen und Bibliotheken verwendet.
Ein „Project“ hat hierbei eine Liste an „Gateway“s, ein „Gateway“ eine Liste von „Sensor“s, ein „Sensor“ eine Liste von „ComunicationInterface“s, ein „ComunicationInterface“ eine Liste von „Chip“s und ein „Chip“ eine Liste von „Measurement“s und „Actor“s.

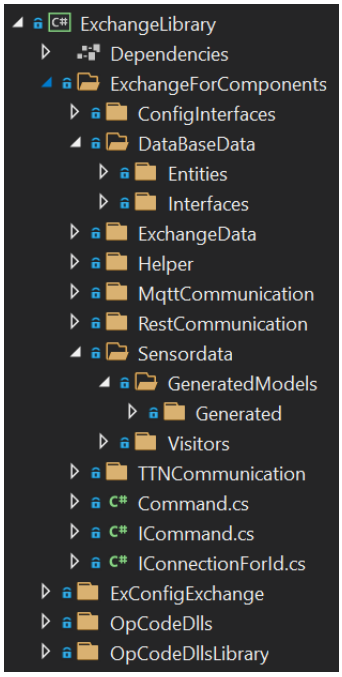

Abbildung 3 ExchangeLibrary-for-components Ordner Struktur

1.4 OpCodes

Die Arbeitsschritte, die von einem „Sensor“ abgearbeitet werden müssen, wurden mit *OpCodes* gelöst. Ein *OpCode* ist ein Byte, welches für eine spezielle Aufgabe steht. Es wird ein Byte Array mit den *OpCodes* an den „Sensor“ gesendet. Die Implementation des „Sensor“s arbeitet diese *OpCodes* nacheinander ab. Beispiele für *OpCodes* sind:

- 10 = DELAY(Ms) z.B. [10,100] wartet für 100 Millisekunden
- 25 = I2C.WRITE(ADDR,BYTE) z.B. [25,32,255] schreibt über I2C das Byte 255 auf die Addresse 32
- 60 = TRANSMIT_SENDBUFFER() z.B. [60] überträgt die gemessenen Werte

Die Bedeutungen der jeweiligen *OpCodes* sind im Dokument „Opcodes.xlsx“ zu finden. Die Klassen für den Umgang mit *Opcodes* sind in dem Projekt „ExchangeLibrary“ im Unterordner „ExchangeForComponents/ConfigInterfaces/“ wie in Abbildung 4 dargestellt, beziehungsweise im Unterordner „OpCodeDllsLibrary“, wie in Abbildung 5 dargestellt, zu finden.

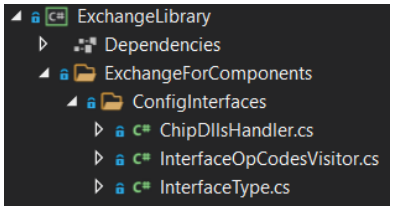

Abbildung 4 ConfigInterfaces Ordner Struktur

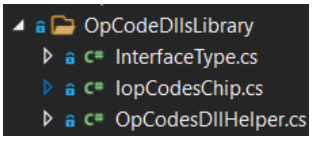

Abbildung 5 OpCodeDllsLibrary Ordner Struktur

1.4.1 OpCodesDllHelper

Der „OpCodesDllHelper“ vereinfacht den Umgang mit *OpCodes* und erstellt *OpCodes* aus leserlicheren Methoden wie:
I2CWrite(byte address, byte writeByte) => new byte[] {25, address, writeByte}

1.4.2 ChipDllsHandler

Der “ChipsDllsHandler” unterstützt dabei, die Implementationen der „IopCodeChip“s aus allen .dll Dateien innerhalb eines angegebenen Pfades zu erhalten.

1.4.3 DemoOpCodeDlls

Unter dem Ordner „DemoOpcodeDlls“ wird die *Solution* „OpCodeDlls“ gefunden. Diese enthält, wie in Abbildung 6 dargestellt, Klassen, welche das „IopCodesChip“ Interface implementieren. Hierzu gehören derzeit:

- BMP280
- GpioStandard
- PCF8574

„GpioStandard“ representiert hierbei den Standardfall eines Gpio Ein-/Ausgangs wobei 0 oder 1 gelesen oder geschrieben wird.

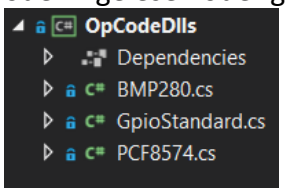

Abbildung 6 OpCodeDlls Struktur

1.5 IopCodesChip

Um einen neuen Chip hinzuzufügen, muss eine Klasse erstellt werden, welche das „IopCodesChip“ Interface implementiert. Ansammlungen von Chips können in .dll Dateien liegen, müssen jedoch in dem dafür vorgesehenen Ordner liegen, um von dem Programm erkannt zu werden. Derzeit muss solch eine .dll Datei als „OpCodeDlls.dll“ bezeichnet werden, in *Visual Studio* in das Projekt ExchangeLibrary in den Unterordner „OpCodeDlls/“ gelegt werden und unter dessen Eigenschaften die Eigenschaft *Copy to Output Directory* auf *Copy always,* wie in Abbildung 7 dargestellt, gestellt werden.
Die wichtigsten Eigenschaften eines „IopCodesChip“s sind:

- Chiptype = der Name für den Vergleich z.B. „bmp280“
- InterfaceType = über welches „CommunicationInterface“ wird kommuniziert? z.B. „i2cinterface“
- BytesToRead = Wie viele Bytes werden bei einer Lese-Aufgabe gelesen
- GetOpCodes = wie von dem Chip gelesen und wie darauf geschrieben wird
- ParsingReadedValueToValue = Wie die gelesenen Daten konvertiert werden müssen

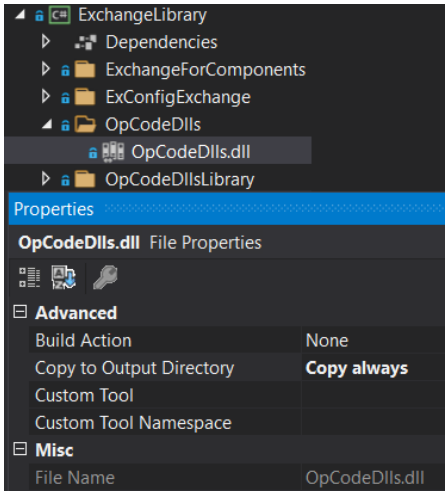

Abbildung 7 OpCodeDlls Vorgehensweise

1.5.1 GetOpCodes

Chips benötigen verschiedene Aufgaben, um einen Lese- oder Schreib-Vorgang abzuhandeln. Deshalb muss zum Beispiel bei dem „BMP280“ für einen Lesevorgang, zuerst zwei Bytes auf spezielle Register des Chips geschrieben werden, danach 50 Millisekunden gewartet werden und erst dann 6 Bytes gelesen werden. Ohne dieser Abfolge können die gewünschten Daten von dem Chip nicht ausgelesen werden.

1.5.2 ParsingReadedValueToValue

Chips lesen Bytes oder Bits aus. Diese müssen konvertiert werden, um den gewünschten Messwert zu erhalten. Somit müssen zum Beispiel bei dem Chip „BMP280“, die Bytes umgewandelt werden, um den gewünschten Temperaturwert zu erhalten.

1.6 Gateway

1.6.1 PCGateway

Der PCGateway erlangt die “Gateway” Konfigurationen von dem „GatewayHandlerInitializer“ und startet den „GatewayHandlerInitializer“ je nach Konfiguration. Derzeit wird diese Konfiguration direkt in dem Ordner des Projektes *Apps* gespeichert mit dem Namen „gateConfigs.json“.

1.6.2 GatewayHandler

Der „GatewayHandler“ benötigt eine Klasse für die Kommunikation mit dem Server („IClientForServer“) und eine Klasse für die Kommunikation mit den „Sensor“s („IServerForSensors“). Zusätzlich wird die Konfiguration des „Gateway“s oder die Kommunikation („IConnectionForId“), wie die Konfiguration des „Gateway“s erlangt wird, benötigt.

1.6.3 RestClientForServer

Eine Implementation für das „IClientForServer“ Interface ist der „RestClientForServer“. Dieser ermöglicht es über eine „ConcurrentQueue“ „MeasurementValue“s zu einer Rest-Schnittstelle zu versenden. Dieser verwendet die „ServerHttpClient“ Klasse, welche einen „HttpClient“ und vorgefertigte Methoden zum Senden von Anfragen an den Server hat.

1.6.4 SerialServerForSensors

Eine Implementation für das „IServerForSensors“ Interface ist der „SerialServerForSensors“. Dieser ermöglicht die Kommunikation über die serielle Schnittstelle zu den „Sensor“s des „Gateway“s. Der „SerrialServerForSensors“ führt das Überprüfen der verfügbaren Geräte, das Senden der Konfigurationen und das Lesen von übermittelten Messdaten über die serielle Schnittstelle kontinuierlich durch.
Bei der Übertragung der Konfigurationen werden die jeweiligen Konfigurationen mithilfe von *Opcodes* an den jeweiligen „Sensor“ übermittelt, welche daraufhin anfangen diese *Opcodes* in einer Schleife abzuarbeiten.
Sobald Daten von einer seriellen Schnittstelle erhalten werden, werden diese über den „InterfaceValueVisitor“ zu „MeasurementValue“s konvertiert. Diese „MeasurementValue“s werden anschließend in der „ConcurrentQueue“ des „SerialServerForSensor“s hinzugefügt. Der „Sensor“ misst all seine „Measurement“s in einem Intervall von „MeasureInterval“, wobei die Werte erst nach dem „MeasureXTimesTillSend“ vielten Mal gesendet werden.

1.6.5 InterfaceValuesVisitor

Der „InterfaceValuesVisitor“ holt sich die „IopCodeChips“ über den „ChipDllsHandler“. Der „InterfaceValuesVisitor“ geht die Chips des jeweiligen „CommunicationInterface“s durch und erstellt pro „Measurement“ des jeweiligen Chips ein „MeasurementValue“. Da die vom Chip ausgelesenen Werte noch konvertiert werden müssen, wird die „ParsingReadedValueToValue“ Methode des jeweiligen „IopCodeChips“ mit den gemessenen Bytes aufgerufen. Der „InterfaceValuesVisitor“ ist in dem Projekt „ExchangeLibrary“ in dem Unterordner „ExchangeForComponents/Sensordata/Visitors/“ zu finden.

1.6.6 GatewayHandlerInitializer

Der „GatewayHandlerInitializer“ unterstützt bei der Erstellung eines „GatewayHandler“s. Hierbei werden zum Beispiel Methoden bereitgestellt, welche einen „GatewayHandler“ mit einem „SerialServerForSensors“ oder einem „UdpServerForSensors“ in Kombination mit einem „RestClientForServer“ erstellt. Des Weiteren unterstützt die Methode „GetConfigs“ dabei, die Konfiguration des „Gateway“s aus der der Datei der Konfiguration zu erhalten.

Derzeit befindet sich diese Datei in dem Ordner des „Apps“ Projektes und ist als „gateConfigs.json“ bezeichnet.

1.7 Sensor

Das Programm des Sensors muss je nach Kommunikation ausgewählt werden. Anschließend muss die „SensorId“ in dem jeweiligen Programm umgeschrieben werden, sodass sie der im Konfigurationstool eingestellten „SensorId“ für den jeweiligen „Sensor“ entspricht. Diese Programme sind in dem Ordner „SensorProjects“ zu finden. Die Bedeutungen der *OpCodes* befinden sich in der Datei „Opcodes.xlsx“.

1.7.1 Serial_Arduino_OPCodesVia_Serial

In diesem Programm wird sowohl die Konfiguration als auch die Kommunikation auf dem Arduino über die serielle Schnittstelle abgehalten. Im *setup* wird die Methode WRITE_EEPROM() aufgerufen.

1.7.2 WRITE_EEPROM

Diese Methode wartet bis Bytes über die serielle Schnittstelle zur Verfügung stehen, das erste Byte 90 und das zweite Byte die *SensorId* dieses Sensors ist. Dies bedeutet, dass die darauffolgenden Bytes die *OpCodes* dieses „Sensor“s sind. Die folgenden Bytes werden in den EEPROM gespeichert. Wird das Byte 90 gelesen, wird das Beschreiben des EEPROMs beendet und der Arduino wechselt in die *loop*.

1.7.3 Loop

Falls derzeit ein „delay“ aktiv ist, soll der *OpCode* gleich 90 (=No Operation) sein. Andernfalls wird der *OpCode* aus dem EEPROM gelesen und der *OpCode* ausgeführt.

1.7.4 SOFTDELAY

9 steht für SOFTDELAY und wandelt die nächsten 2 Bytes zu einem 16 Bit Integer um, welcher die Zeit des *delays* beschreibt. Nun wird die *delayPeriod*, der *sleepCounter* und der *delayStart* gesetzt. Daraufhin wird bei dem nächsten *loop* Durchlauf so lange 90 (=No Operation) als *OpCode* genommen, bis der *delayStart* + *delayPeriad* größer als die derezeitigen Millisekunden sind.

1.7.5 Send Counter

12 steht für INCREASE_SEND_COUNTER, was einen *counter* an der EEPROM stelle 510 erhöht.
63 steht für TRANSMIT_SENDBUFFER_WHEN_COUNTER. Dies liest das nächste Byte und überprüft, ob dieses Byte den *counter* auf 510 bereits erreicht hat. Falls ja, wird der *counter*

auf 510 auf 0 gesetzt und die eigene *SensorId* in Kombination mit allen Werten aus dem SENDBUF über die serielle Schnittstelle gesendet.
Diese Kombination ermöglicht das Verhalten, dass die Daten ein Mal pro, zum Beispiel, 10 Messungen übertragen werden.

1.7.6 Einschränkungen

Der Sendepuffer „SENDBUF[500]“ ist auf 500 Bytes beschränkt. Dies bedeutet, dass die gemessenen Werte aller „Measurement“s mal der Anzahl wie oft gemessen wird bevor gesendet wird (=“MeasureXTimesTillSend“), nicht mehr als 500 Bytes betragen darf.

1.8 Datenbank

Mithilfe von *EFCore* werden die Daten auf einem SQL-Server gespeichert.
Das Speichern in der Datenbank erfolgt über die Klassen „TableProject-/TableGateway-etc“. Diese benötigen eine andere Datenstruktur als „Project/Gateway etc.“, da sie sonst nicht in der Datenbank gespeichert werden können.
Hierbei bilden die Klassen „TableBaseProject/TableBaseGateway etc.“ die Basis der Daten. Die Klassen „TableUserProject/TableUserGateway etc.“ werden für die gespeicherten Konfigurationen der Benutzer benutzt, wobei „TablePublishedProject/TablePublishedGateway etc.“ für die veröffentlichten Konfigurationen benutzt werden.

1.8.1 Mapping

Mithilfe der Klassen „ProjectMapping/GatewayMapping etc.“ können die jeweilig definierten Klassen für „TEntity“ auf „IOutputDTO<TEntity>, beziehungsweise von „IInputDTO<TEntity>“ auf „TEntity“ gemappt werden.

1.8.2 IEntityTypeConfiguration

Pro „IEntity“ wurde eine Konfiguration erstellt, welche die Konfigurationen der Tabelle wie *PrimaryKey* und *ForeignKey* bestimmt. Im Beispiel von „TablePublishedProject“ schaut dies wie folgt aus. *PrimaryKey = builder.HasKey(x => x.Id); ForeignKey = builder.HasMany(x => x.Gateways).WithOne(x => x.Project).HasForeignKey(x => x.ProjectId);*

1.9 REST-API

Der „RestBaseController“ liefert die Basis der RestController und bietet die Standard-Methoden für Get/Post/Put/Delete. Die weiteren Controller wie „ProjectsController/GatewaysController etc.“ bieten die Implementierungen eines „RestBaseController“s, sowie spezielle Methoden für diese.
Die Rest-Controller müssen eine Klasse für „TEntity“, „IInputDTO<TEntity>“ und „IOutputDTO<TEntity>“ definieren.

Im Falle des „ProjectsController“ wäre dies „TablePublishedProject“, „Project“ und „Project“. Hiermit kann zum Beispiel bei einem „Get“ Aufruf, die aus der Datenbank geholten Daten als „Project“s abgegriffen werden, beziehungsweise bei einem „Post“ Aufruf ein „Project“ gesendet werden, ohne sich über die Konvertierung Gedanken machen zu müssen.

2 Konfigurationstool

Bevor Sensoren aktiviert und verwendet werden können, muss eine Konfiguration des gesamten Projektes und aller sich darin befindenden Komponenten erstellt werden. Das Konfigurationstool ist für diese Konfiguration zuständig. Hierbei können Projektkonfigurationen erstellt, gespeichert und veröffentlicht werden.

2.1 ConfigItems

Die Idee hinter „IExConfigItem“s ist das dynamische Erstellen von Konfigurationsmöglichkeiten, ohne auf die Datentypen angewiesen zu sein. Hiermit soll es zum Beispiel möglich sein, Klassen aus einer .dll zu laden und UI-Elemente entsprechend den Eigenschaften der Klasse zu generieren. Die dafür benötigten Klassen befinden sich zum großen Teil in dem Projekt „ExchangeLibrary“ in dem Unterordner „/ExConfigExchange/“, wie in Abbildung 8 dargestellt.

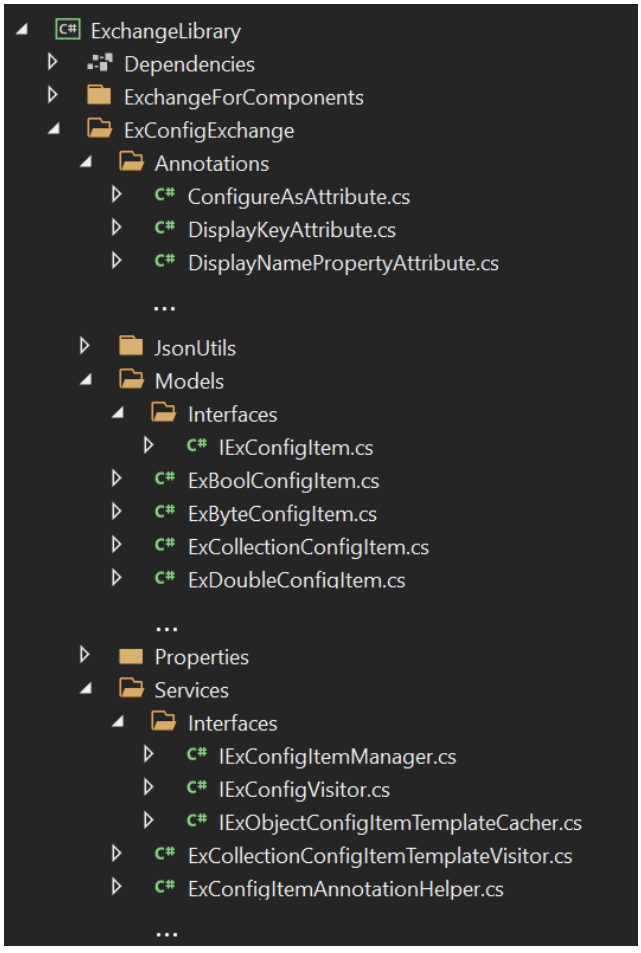

Abbildung 8 ExConfigExchange Ordner Struktur

2.1.1 IExConfigItems

Ein „IExConfigItem“ kann einen „IExConfigVisitor<T>“ akzeptieren und hat folgende Eigenschaften:

- string DisplayKey = Welchen Darstellungstext das Element in der UI hat

- bool Hidden = Ob das Element in der UI dargestellt wird
- bool ReadOnly = Wenn das Element nur gelesen werden darf

2.1.2 ExCollectionConfigItem

Ein „ExCollectionConfigItem“ ist ein “IExConfigItem” mit einer *ObservableCollection* an „IExConfigItem“s.

2.1.3 ExConfigItemUIVisitor

Der „ExConfigUIVisitor“ besucht ein „IExConfigItem“ und erstellt daraus ein „Xamarin.Forms.View“ Element. Dadurch kann das UI an die jeweiligen „IExConfigItem“s angepasst und direkt an die darunter liegenden Klassen gebunden werden.
Bei einem „ExEnumConfigItem“ wird ein *dropdown* Element erstellt.
Bei einem „ExCollectionItem“ wird ein *addbutton* erstellt. Pro Element der *ObservableCollection* wird der „ExConfigItemUIVisitor“ besucht und das daraus resultierende „Xamarin.Forms.View“ Element inklusive eines *deletebuttons* erstellt.

2.1.4 ExConfigItemValidationVisitor

Der „ExConfigItemValidationVisitor“ ist für die Validation der „IExConfigItem“s verantwortlich. Dieser bestimmt die validen Werte von z.B. byte = 0 bis 255.

2.1.5 ExConfigItemSetterVisitor

Der „ExConfigItemSetterVisitor“ ist für das Setzen der Werte von „IExConfigItem“s zuständig.

2.1.6 Annotations

Annotations sind Attribute für die Verwendung von „IExConfigItem“s. Diese sind vor allem für das dynamische Einlesen von Klassen notwendig.

- „DisplayNamePropertyAttribute“ = kennzeichnet die Eigenschaft, welche in der UI für diese Klasse als Darstellungsnamen angezeigt wird.
- „HiddenAttribute“ = kennzeichnet, wenn eine Eigenschaft in der UI nicht angezeigt werden soll.
- „ImplementationRequiredAttribute“ = kennzeichnet eine Eigenschaft, welche gesetzt werden muss.
- „ReadOnlyAttribute“ = kennzeichnet eine Eigenschaft, welche nur gelesen werden kann.

2.1.7 IExConfigurable

Für die Darstellung und Konfiguration von Projekten gibt es die „ExProject“ „ExGateWay“ und „ExSensor“ Klassen, welche alle das „IExConfigurable“ Interface implementieren. Diese besitzen ein *ObservableDictionary<string,IExConfigItem>* welches durch *Reflection*

aus den originalen Modellen ausgelesen wird. Der Schlüssel beschreibt hierbei den Eigenschaftsnamen, wobei der Wert das „IExConfigItem“ beschreibt.

2.1.8 ExConfigurableManager

Der „ExConfigureableManager“ ist für konfigurierbaren Modelle zuständig. Die Methode *GetConfigurationFor<T>* erstellt die konfigurierbaren Eigenschaften. Weiters kann der „ExConfigurableManger“ „Project/Gateway/Sensor“s zu „ExProject/ExGateway/ExSensor“s konvertieren.

2.1.9 VmConfigurationTool

In der Methode „OnActivated“ werden einerseits die Templates für „ExProject/ExGateway/ExSensor“s geladen, wie diese zu konfigurieren sind und andererseits die vom Benutzer erstellten Konfigurationen geladen. Mittels der Methode „InititalizeProjects“ werden die vom Benutzer erstellten Konfigurationen zu *Viewmodels* konvertiert.

2.1.10 SensorOpCodesHelper

Der „SensorOpCodesHelper“ liest mittels „ChipDllsHandler“ alle „IopCodeChip“s aus dem für diese vorhergesehenen Ordner aus. Mittels „InterfaceOpCodesVisitor“ werden die *Opcodes* aller „Measurements“ generiert und dem angegebenen „Sensor“ hinzugefügt. Zusätzlich werden *OpCodes* für eine „SoftDelay“- mit „MeasurementInterval“, eine „Increase_Send_Counter“- und eine „Transmit_Sendbuffer_when_Counter“-Aufgabe hinzugefügt.

2.1.11 Abfolge IopCodesChip

Wird zum Beispiel ein „I2cChip“ zu einem „I2CInterface“ hinzugefügt, werden die konfigurierbaren Eigenschaften des „I2cChip“s hinzugefügt. Die Eigenschaft „ChipType“ ist hierbei mit dem Attribut „ConfigureAs(typeof(IopCodesChip))“ gekennzeichnet. Hierbei werden die „IopCodeChip“s aus dem für diese vorhergesehenen Ordner mittels „ChipDllsHandler“ ausgelesen. Nun wird für die Auswahl des „ChipType“s ein *dropdown* Element erzeugt, welches die „ChipType“s der ausgelesenen „IopCodeChip“s darstellt. Bei Drücken auf „Als nächstes: Gateways verbinden“ wird in dem „VmConfigurationTool“ in dem Befehl „CmdToGatewayConnectorView“ das selektierte Projekt herangezogen. Mittels „SensorOpcodesHelper.AddOpCodesTo“ wird jedem „Sensor“ dessen OpCodes hinzugefügt.

2.1.12 Download Gateway Settings

Bei Drücken auf „download Gateway Settings“ werden die Konfigurationen von dem ausgewählten „Gateway“ heruntergeladen. Diese beinhalten alle die gesamte Konfiguration des „Gateway“s sowie dessen „Sensor“s und dessen OpCodes. Derzeit werden diese Konfigurationen nur in dem Ordner des Projektes mit der Bezeichnung „gateConfigs.json“ abgelegt.