HAW
Communication Agent

Table of contents

1. Introduction

The student innovation laboratory for Internet of Things at the University of Applied Sciences Landshut is one of the ten laboratories funded by the Bavarian State Ministry for Education and Culture, Science and the Arts as part of the Center for Digitization in which students develop innovative prototypes in the field of digitization in group work using agile software and system development methods. The ideas for the projects come from different sources (e.g. companies, participating students, ZD.B theme platforms). With the IoT innovation laboratory the HAW Landshut wants to prepare students for the development and application of IoT technologies. This way, students should acquire competencies, that are becoming increasingly important for the professional and start-up world in relation to Industry 4.0.

The project covered in this documentation is part of the elective subject Innovation Laboratory IoT Project, provided by the innovation laboratory of the HAW Landshut.

The students identify and recognize reality based problems. Also they learn the creation of complex solutions using a wide variety of IoT platforms. They are able to analyse the environment of the problem and can work together with companies to solve it. They acquire Knowledge of design thinking, project management, independent execution of projects and teamwork. They are in a position to acquire interdisciplinary knowledge to integrate the product owner into the project in an agile manner and to obtain work results.

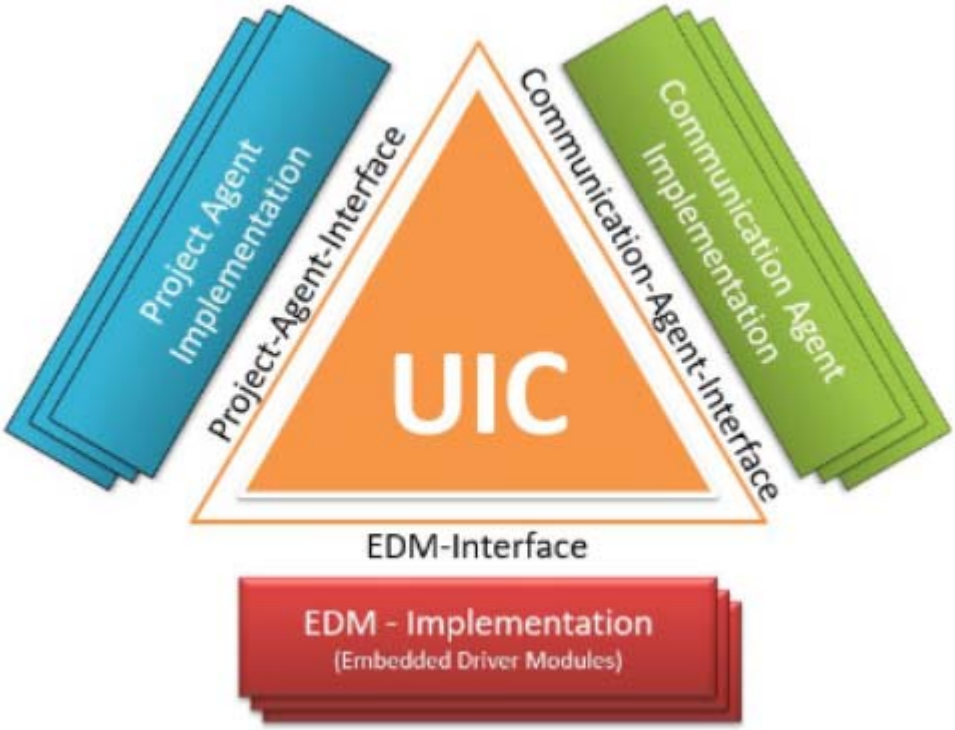
The topic of this project is provided by congatec AG and was defined as “UIC Integration to HAW Cloud Environment”.

This document represents the documentation of the project carried out.

The Universal IoT Connector is an public open standard hosted by the Standardization Group of Embedded Technologies. The functionality is to receive, collect and publish hardware data and bundling the data to information sets. The destination of the published hardware data is determined by the communication Agent (CA). The CA is a main component of the UIC architecture. Currently, the data can only be published to AZURE cloud via the M2Mgo Communication Agent.

Goal definition

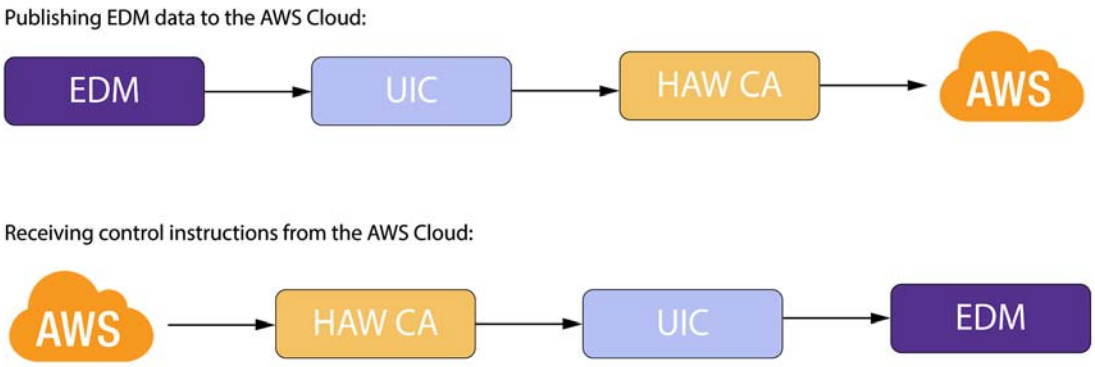
The main goal of this project is to extend the existing functionality of the UIC to connect to an AZURE Cloud environment by an additional Communication Agent which connects to an Amazon Web Service Cloud environment. This work is done as a Proof of Concept and therefore demonstrates the feasibility of the conceptual use cases.



UIC interface overview specified in the UIC Specification by SGeT

Hardware data is collected via Embedded Driver Modules(EDM's) by the UIC, then forwarded to the Communication Agent and then published on an AWS Cloud environment. The new Communication Agent should also be able to receive data and forward it to the lower EDM level in the UIC.

Forwarding the data to the EDM level is necessary for future use cases, where the hardware will be controlled via a cloud environment.



2. Solution approach

It is important to have a reliable connection to AWS. Amazon provides libraries for different programming languages, but at the current time there is no library for the AWS Iot Core in C#.

The non-existence of a C# library leads to the decision to introduce a second component - the UIC-AWS-ConnectionServer - to allow the use of existing libraries provided by Amazon.

The UAS is written in JAVA to be platform independent because Java code is runs on the Java virtual machine and is therefore platform independent.

Together with the build management tool Apache Maven and its dependency management, it is really easy to use Amazon´s libraries.

The downside of this is that another communication must be established between the HAW CommuniationAgent in the UIC and the UAS. This leads to an increase in complexity. For this communication REST APIs on both sides are used.

For detailed information please take a look at the chapters Architecture and Communication.

The UAS acts as an MQTT client that publishes and receives MQTT messages from the AWS IoT Broker. The publishes include the data such as the initialisation, data points and attribute points.

Further it will subscribe to the topic that is set in the configuration to receive messages on the backchannel and will forward it via REST to the HAW ConnectionAgent/UIC.

During our work we found out that the M2Mgo Communication Agent is initiated in the launcher directly. We needed to find a way to configure the UIC to load our Communication Agent. We extended the uic_config.json for this option (see Chapter Configuration for more information) and implemented a simple logic to initialize the desired Communication Agent in the launcher of the UIC.

Further we found a bug that prohibited the local loading of the project.json and reported it to the Product Owner. We fixed the bug ourselves but decided to use the official fix after its release.

Technology stack

.NET 4.5.2

“.NET is a free, cross-platform, open source developer platform for building many different types of applications.” (Source: https://www.microsoft.com/net/learn/what-is-dotnet)

The **UIC** is based on .NET and uses its features.

JAVA 8

The UAS is based on JAVA 8 and uses its features.

MQTT

"MQTT stands for MQ Telemetry Transport. It is a publish/subscribe, extremely simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks. The design principles are to minimise network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery. These principles also turn out to make the protocol ideal of the emerging "machine-to-machine" (M2M) or "Internet of Things" world of connected devices, and for mobile applications where bandwidth and battery power are at a premium." (*Source: mqtt.org*)

More Information under: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf

REST API

"REST defines a set of architectural principles by which you can design Web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages. If measured by the number of Web services that use it, REST has emerged in the last few years alone as a predominant Web service design model. In fact, REST has had such a large impact on the Web that it has mostly displaced SOAP- and WSDL-based interface design because it's a considerably simpler style to use." (*Source:* https://www.ibm.com/developerworks/webservices/library/ws-restful/)

Tutorials:

Creating a REST API Client under c#:
https://docs.microsoft.com/de-de/dotnet/csharp/tutorials/console-webapiclient

Creating a REST API Server under c#:
https://msdn.microsoft.com/en-us/library/mt481589(v=vs.110)

Creating a REST Service under c#:
https://msdn.microsoft.com/en-us/library/system.servicemodel.web.webservicehost(v=vs.110).aspx

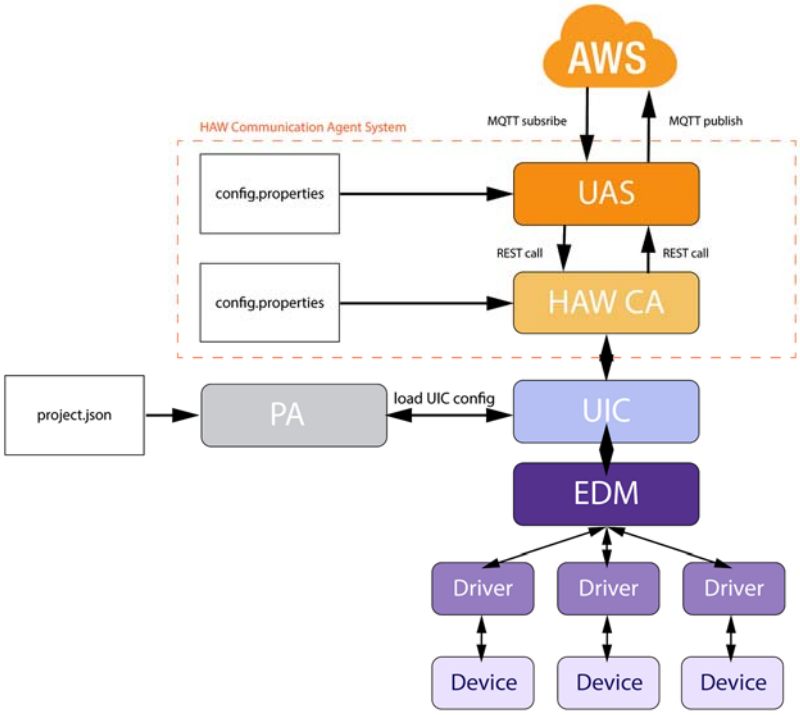
AWS IoT Core

"AWS IoT Core is a platform that enables you to connect devices to AWS Services and other devices, secure data and interactions, process and act upon device data, and enable applications to interact with devices even when they are offline."
(Source: https://aws.amazon.com/iot-core/features/?nc1=h_ls)

The AWS IoT Core provides a MQTT message broker. With this broker, it is possible to publish and receive data of a special topic. The topic is not static and is determined by the sender. A receiver subscribes to this topic and will receive every data package which is published

3. Architecture

Overview over the complete Architecture

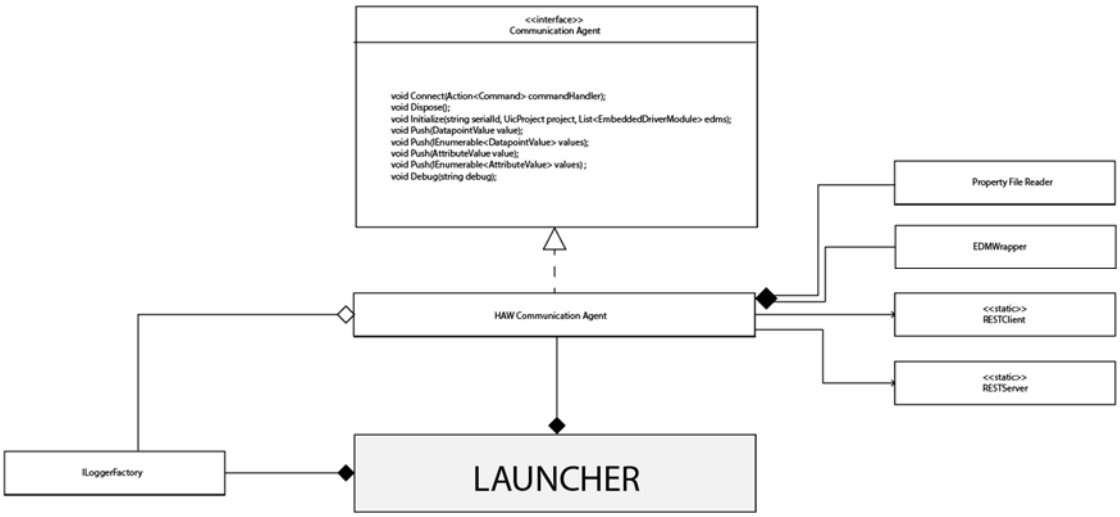


Caption:

EDM - Embedded Driver Module
(https://pst.m2mgo.com/cms/node/39d32ad6-5f9d-4d91-9ea0-67a4ce541b98)

PA – Project Agent
HAW CA - HAW Communication Agent
UAS - UIC Amazon Connectionserver

UML Class diagram HAW CA



Note: this is no complete UML class diagram. It should only show the most important classes to give a quick overview.

Explanation:

The Launcher is the central part of the UIC. It creates a Communication Agent object. For the HAW CA to work properly, the Launcher had to be edited.

The HAW CA implements the Communication Agent Interface. The image above shows, how the HAW Communication Agent is linked to the launcher.

HAW Communication Agent

Interface Communication Agent

Each Communication Agent must implement the `UIC.Framework.Interfaces.Communication.Application.CommunicationAgent` interface.

The interface includes the following methods:

```
// this method is called while establishing a connection
public void Connect(Action<Command> commandHandler);

// this method  is called when the Communication Agent is disposed
public void Dispose();

// this method hands important information over to the Communication Agent
public void Initialize(string serialId, UicProject project, List<EmbeddedDriverModule> edms);

// sends an single set of sensor data to the cloud
public void Push(DatapointValue value);

// sends several sets of sensor data to the cloud
public void Push(IEnumerable<DatapointValue> values);

// sends an single set of information on the device to the cloud
public void Push(AttributeValue value);

// sends several sets of information on the device to the cloud
public void Push(IEnumerable<AttributeValue> values);

// debug method to hand a string to the Communication Agent for testing purpose
public void Debug(string debug);
```

All of these methods are called by the UIC.

Implementation

The HAW Communication Agents acts as a layer between the UAS and the UIC that on one hand calls REST Resources of the UAS and on other hand receives REST calls on its REST API.

The JSON format is used for transportation and the parsing is done by a library included in the C# framework.

Adding the CA to the launcher

The M2mgo Communication Agent was initiated directly in the launcher without the possibility to change it with a configuration.

We added a if-else cascade to make the launcher able to initiate the Communication Agent according to the uic.config.json.

If another Communication Agent should be implemented it needs to be added to the if-else cascade. With more Communication Agents it might make sense to change the coding to a switch-case.

Backchannel

The backchannel is implemented via a REST API by the HAW Communication Agent. The UAS uses this REST API and transfers the commands via a http requests. The commands are mapped by the HAW Communication Agent and delivered to the corresponding EDM.

The mapping of the EDM identifiers and the actual EDMs is needed for the backchannel. In the initiation method the list and details of the EDMs is handed over to the Communication Agent. A mapping is build from the details and makes it possible to find the right EDM for each incoming command. After the EDM received the command it looks up the right method to be called and executes it.

Communication to the UAS (REST)

The communication between the HAW Communication Agent/UIC and the UAS is done via REST API calls :

POST: http://localhost:UAS_PORT/rest/iot/init

This REST Resource is called for initiation.

POST: http://localhost:UAS_PORT/rest/iot/push

This REST Resource is called to publish data via the broker.

UAS (UIC-AWS-ConnectionServer)

General

REST calls on the UAS are done via http-requests. It is possible to hand over different information for the initialisation and for pushing sensor and device data.

This data is delivered via MQTT to the AWS broker, from which the data can be analyzed and progressed. For this communication a Amazon library is used.

The authentication on the Amazon broker is done via .x509 certificates handed out by AWS. This ensures a secure communication to the broker as MQTT provides no build-in encryption.

The port for the REST API of the UAS is set in its configuration file. Please take a look in the Configuration chapter for more information.

For security reasons the UAS supports binding only on the localhost. This is because we do not use https which can enable an attacker to easily obtain information. Furthermore network traffic between the HAW Communication Agent and the UAS is avoided.

Configuration framework

Aeonbits Owner is used as a configuration framework. This framework was chosen because we have experience with it. The general idea is to easily define the configuration with annotations in interfaces.

Logging

To ensure traceability of the process the UAS is using the wide-spread logging library log4j2. It is possible to define the logging behavior via the log4j2 file that is next to the UAS-jar-file.

In the default version there are two appenders. One shows the logging messages on the console and the other saves the messages in a file.

For more detail on the configuration of the logger, please visit the site: https://logging.apache.org/log4j/2.x/manual/configuration.html

REST API Implementation

The REST service is implemented via JAX-RS. A Jetty server is used to publish the REST ressources. There are two different paths for the REST API. One is for the initialisation and hands over a list of the edms and the serialid of the device on which the UIC operates. This serialid can later be used for the topics, as the token {serial_id} will be replaced by the actual serial_id.

Initialize

Information on the edms is handed to be cloud

Example topic: *myproject/uicinfo/<serial_id>*

URL: localhost:<UAS_PORT>/iot/rest/init (POST)

Push

Sensor or device data is pushed to the cloud

Example topic: myprojekt/uicinfo/<uic-id>

URL: localhost:<UAS_PORT>/iot/rest/push (POST)

Communication to the UIC(REST)

The communication between the UAS and the HAW Communication Agent/UIC is done via REST API calls :

POST: http://localhost:UIC_PORT/backchannel

This REST call is used to hand back channel messages published by the AWS IoT Broker over to the HAWS Communication Agent/ UIC. Detailed information is given in the chapter Communication.

AWS IoT-Core

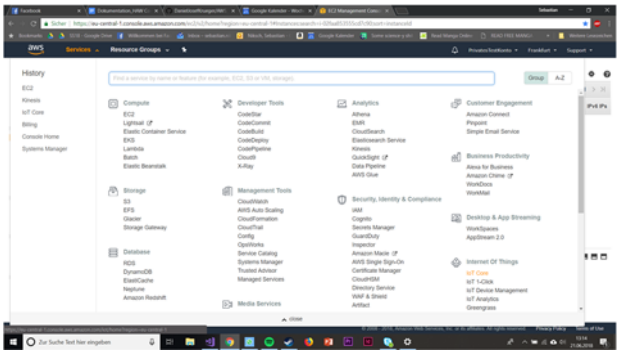
IoT Core Amazon offers a central IoT Broker. This is set up as follows:

Amazon provides a central MQTT Broker with AWS IoT Core.

Setup

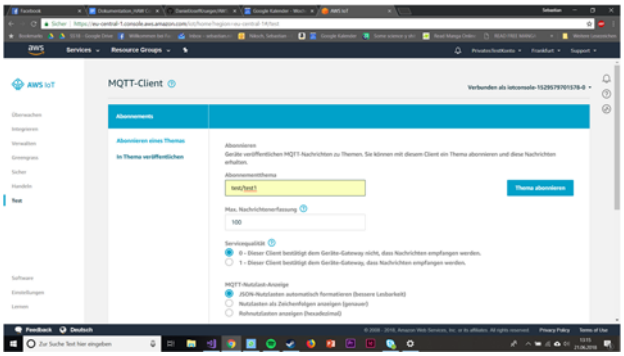
Create an account at https://portal.aws.amazon.com/billing/signup#/start Payment information must be provided.

Publishing and Subscribing of MQTT messages



Change to IoT Core.

Under "Test" you can subscribe to or publish MQTT topics.

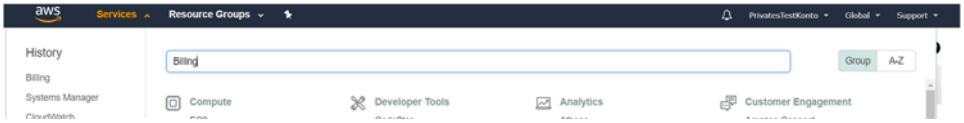


Learn more about connecting to AWS:

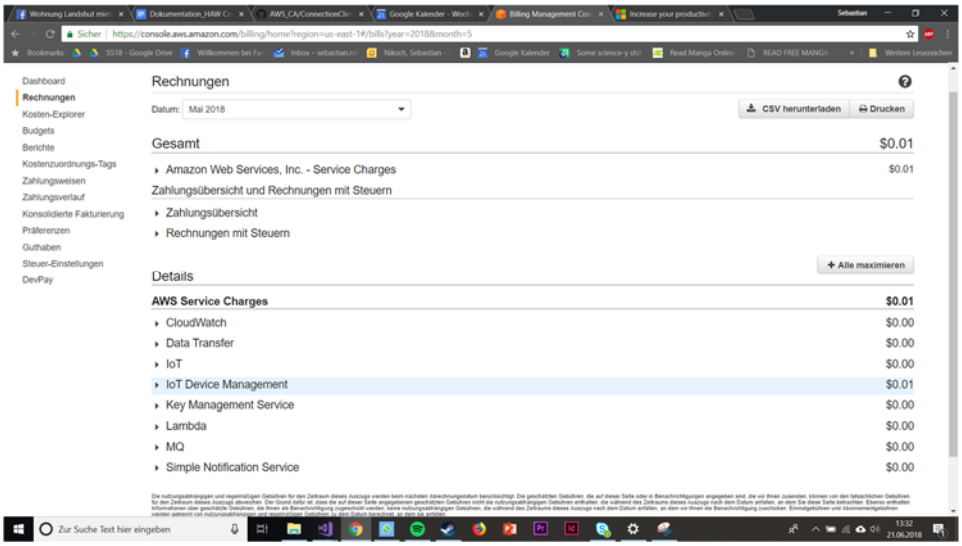
https://github.com/aws/aws-iot-device-sdk-java#sample-applications

Cost control

Using the AWS account may cost you money. This should occur only with very many messages. Nevertheless, you should regularly get an overview to avoid any nasty surprises.



Search for "Billing" under Services. All costs incurred are listed there.



4. Communication

In this chapter the communication between the HAW Communication Agent - UAS - AWS is explained more in detail.

PUSH CommuniationAgent -> UIC-AWS-ConnectionServer
Protocol: HTTP
URL: localhost:<UAS_PORT>/rest/iot/push
MIME Type: Application/json

The given json message will be published on the topic that is provided in the configuration.

Further the token {serial_id} will be replaced in the topic to make it possible for the business logic to keep different UICs apart and react accordingly.

PUSH UAS -> AWS-IoT-Broker
Protocol: MQTT
URL: Broker-Url (set in the UAS configuration)
topic: set in the UAS configuration

payload: Payload of the REST API call

INITIALIZE CommuniationAgent -> UIC-AWS-ConnectionServer
Protocol: HTTP
URL: localhost:<PORT>/rest/iot/init
payload: two form parametres:

serialid =<serial id of the device>

edms = <information on the edms as a json>

INITIALIZE UIC-AWS-ConnectionServer-> AWS-IoT-Broker
Protocol: MQTT
topic: set in the UAS configuration
Payload: <information on the edms as a json>

BACKCHANNEL AWS-IoT-Broker -> UIC-AWS-ConnectionServer
Protocol: MQTT
Payload:

```
{
        "id":"uid des EDM",
        "commandId":"CommandoID/URI",
        "commandPayload":"Payload as a string"
}
```

Example:

```
{
        "id":"1e7b861c-d6a0-4993-812e-0ffd42a4c77d",
        "commandId":"7f72d80d-43f8-49ae-a808-536026960c49",
        "commandPayload":"Hallo Welt"
}
```

Backchannel UIC-AWS-ConnectionServer -> HAWS CommuniationAgent
Protocol: HTTP
URL: localhost:<UIC_PORT>/backchannel
Payload: a json consisting of the topic and the the message received by the broker

```
{
        "topic":"<TOPIC_OF_THE_PUBLISH>",
        "payload":{
                "id":"uid des EDM",
                "commandId":"CommandoID/URI",
                "commandPayload":"Payload as a string"
        }
```

}

5. Configuration

This chapter aims at the configuration of both components.

There are five configuration files:

- uic_config.json (configuration for the uic)
- project.json (configuration for the uic project)
- config.properties (for the HAW Communication Agent)
- config.properties (for the UAS)
- log4j2.xml (configures the logging behavior of the UAS)

UIC

uic_config.json

Different configuration settings for the UIC are set here. It works as the startup configuration where settings are loaded and set for further execution.

If the file does not exist, a default version will be used from the Table below.This is the startup configuration for the UIC where presettings will be loaded and set for further execution. This file must be in the same directory as the launcher.exe. In case this file does not exist, a default version will be used from the Table below.

Example:

```
{
      "ProjectKey": "26895846c960465ebd89f28d10e6460c",
      "IsEdmSnychronizationEnabled": true,
      "ProjectJsonFilePath": ".\\project.json",
      "IsRemoteProjectLoadingEnabled": true,
      "CommunicationAgent": "AWS"
}
```

KEY	Description	DEFAULT
ProjectKey	Project key for this project	"26895846c960465ebd89f28d10e6460c"
IsEdmSnychronizationEnabled	true: the UicConnector synchronizes the EDMS with the configuration for this system	True
ProjectJsonFilePath	Path to the project.json file	“.\project.json"
IsRemoteProjectLoadingEnabled	true: the project.json is loaded from the cloud. false: the local file is used	True
CommunicationAgent	Which Communication Agent is used: **AWS:** HAW Communication Agent **M2MGO**:M2MGO CommuncationAgent	**M2MGO**

project.json:

The name and path of this file is set in the uic_config.json !

This file includes all settings for the UIC project. If this file does not exist, a default version will be loaded with the default settings from the table below.

KEY	Description	Default
ProjectKey:	UID of the project	"26895846c960465ebd89f28d10e6460c"
Name:	Project name	"JU Test"
Description:	Description of the project	"Maini JU Test Project"
Owner:	Owner of the project	"SGeT"
CustomerForeignKey:	UID of the customer	"12a52d5d-6ce5-407d-b557-26dd44c9ae9c"

Attributes		
Attributes:	Array, Bord/System-Informationen	
Id:	Id for the cloud	"b68df3f9-4748-4c9d-9bda-567c87fab855"
Label:	Label of the attribute	"Timestamp"
Description:	Description of attribute	"Simple DateTime string"
Datatype:	The datatype of the attribute value	0
Uri:	Path to the function of the EAPI	"UIC.EDM.Test.Mockup.MockupEdm.attribute.timestamp"
DatapointTasks		
DatapointTasks:	An Array of Objects with CA configurations of the respective Datapoints	
PollIntervall:	Query interval	“30”
Description:	Short description of the task	null
Definition:	Object, Description of the EDM's	
Id:	UID of the EDM	"fbd3e390-ffb7-455b-b0dc-695b13329eb6"
Label:	Label of the EDM	"Random String"
Description:	Short description of the EDM	"messaging mockup"
DataType:		0
Uri:	Path to the function	"UIC.EDM.Test.Mockup.MockupEdm.datapoint.String_mock"
ReportingCondition:	Object, Cloud reporting configuration	
ReportingThresholdInpercent:	Reporting threshold in %	0.0
MinimalAbsoluteChange:		-1.0
ReportingThresholdInMilliSecs:	Reporting threshhold in ms	10000

MetaData:	Object, additional data	
ExpectedMaximum:	Maximum of the object value	0.0
ExpectedMinimum:	Minimum object value	100.0
WarningThreshold:	Warning threshold	80.0
ErrorThreshold:	Error threshold	90.0
IsInverseThresholdEvaluation:	Inverse Scaling, default from min to max	false
Tags:	Data description	"SimpleValue"

Configuration of the HAW Communication Agent:

config.properties

This file must be in the same folder as the .exe file. It may be the same file that is used for the UAS. However, only the following properties are used:

```
# port for the REST API of the uic
port_uic = 8081

# port for the REST API of the UIC-AWS-ConnectionServer
port_uas = 8080
```

Configuration of the UAS:

config.properties

All settings for the UAS are stored here. The config.properties file must be in the same folder as the JAR of the UAS.

```
# The region of the aws server that is used (f.e. server: data.iot.us-west-2.amazonaws.com
=> region: us-west-2)
region=us-west-2

# the certificate file
cert=INSERT_CERTIFICATE_FILENAME
```

```
# the ClientID for this Instance. MUST be unique in the whole project
clientid=CHANGEME123

# port for the REST API of the uic
port_uic = 8081

# port for the REST API of the UIC-AWS-ConnectionServer
port_uas = 8080

# The praefix of the aws server that is used (f.e. server: data.iot.us-west-2.amazonaws.com
=> prasefix: data   )
praefix=data

# the file that contains the private key of the certificate
private_key_file=INSERT_PRIVATE_KEY_FILENAME

# the quality of service that is sued for publishes
qos=1

# the topic that is used for pushing data such as Attributes and DataPoints
push_topic=myProject/{serialid}/push

# the topic that is used for the Initiation
# CAN be overwritten in the http-post-request
init_topic=myProject/{serialid}/init

# the topic that is used for the subscription and the backchannel feature
backchannel_topic=myProject/back
```

KEY	Description	Default
region	The region of the AWS server, for example for the server data.iot.us-west-2.amazonaws.com the region would be **us-west-2**	
cert	Path to the certificate for the IoT Device issued by AWS. (See chapter Setting up)	
clientid	A self-assignable clientID for this UAS (must be unique to the project)	testClientID
port_uic	Port of the REST interface of the UIC	8081
port_uas	Port of the REST interface of the UAS	8080
praefix	The prefix of the AWS server, for example for the server data.iot.us-west-2.amazonaws.com prefix would be **data**	data
private_key_file	Path to the PrivateKeyFile for the IoT device issued by AWS. (See chapter Setting up)	
qos	The QualityOFService for message exchange	1
push_topic	Topic on which the data obtained by the UIC from the push methods are published to	myProject/{serialid}/push

init_topic	Topic on which the data received from the UIC from the initiates are published to	myProject/{serialid}/init
backchannel_topic	Topic on which a subscription is used to receive messages in the backchannel	myProject/{serialid}/back

log4j2.xml

Here the logging of the log4j2 logger is set. For more information, please visit https://logging.apache.org/log4j/2.x/manual/configuration.html

The file must also be in the JAR folder of the UAS.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <Appenders>
        <RollingFile name="ROLLING" fileName="pushserver.log"
                     filePattern="pushserver-%d{yyyy-MM-dd}.log">
            <PatternLayout>
                <pattern>[%-5level]       %d{yyyy-MM-dd       HH:mm:ss}
UIC_AWS_CONNECTIONSERVER %c{1} - %msg%n</pattern>
            </PatternLayout>
            <Policies>
                <TimeBasedTriggeringPolicy/>
                <SizeBasedTriggeringPolicy size="30 MB"/>
            </Policies>
            <DefaultRolloverStrategy max="10"/>
        </RollingFile>
        <Console name="CONSOLE" target="SYSTEM_OUT">
            <PatternLayout        pattern="%d{HH:mm:ss}        UIC-AWS-
ConnectionServer %-5level - %msg%n"/>
        </Console>
    </Appenders>
```

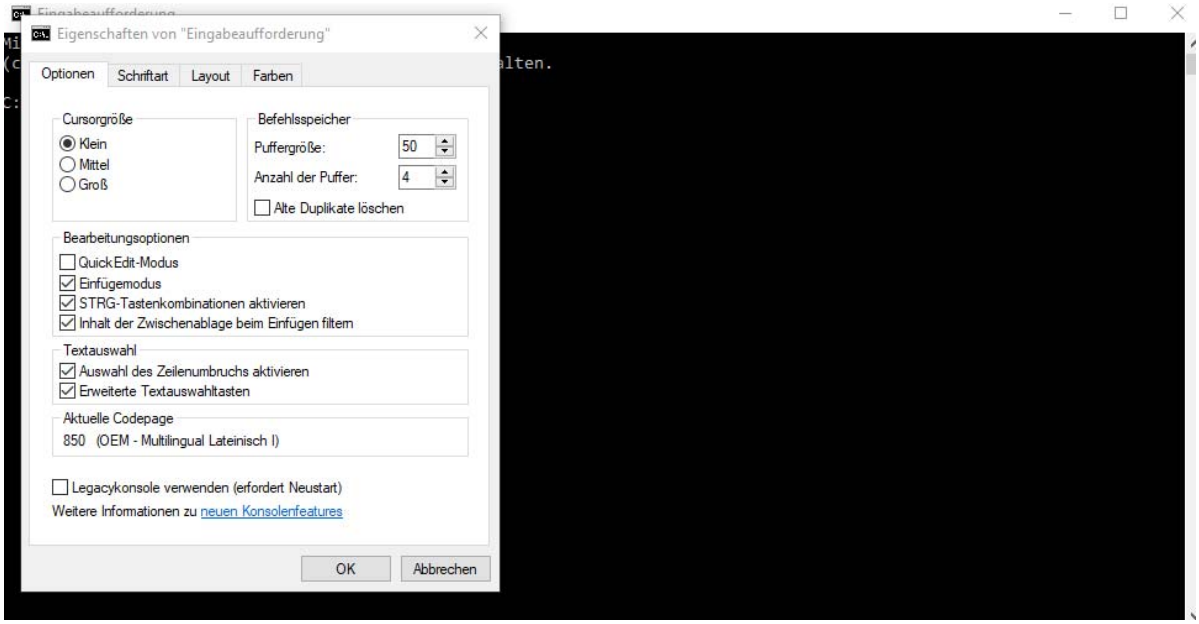
```
    <Loggers>
        <Logger name="de" level="info">
            <AppenderRef ref="CONSOLE"/>
        </Logger>
        <Root level="trace">
            <AppenderRef ref="ROLLING"/>
        </Root>
    </Loggers>
</configuration>
```

6. Setting up the project

This chapter provides a quick introduction to setting up a working project.

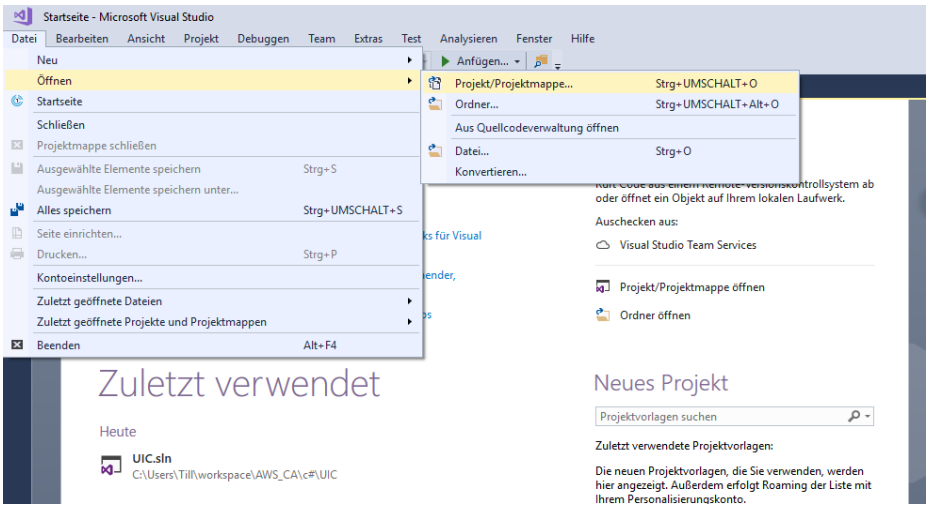
IMPORTANT:

Windows 10 provides a special feature called QuickEdit. This enables the user to copy text from the console easier. However the program that is currently running in this console window is stopped at IO operations. As this is not intended it is important to disable this feature by right-clicking on the title bar and selecting properties. Disable the QuickEdit feature in the newly opened window.

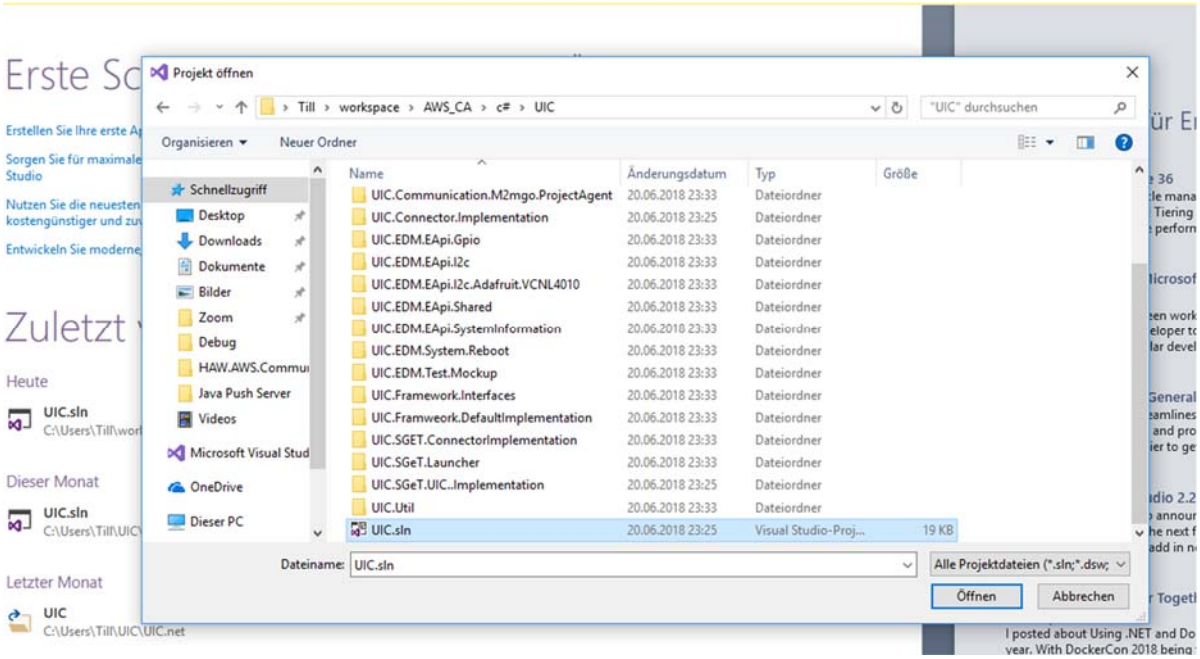


1. Setting up the UIC

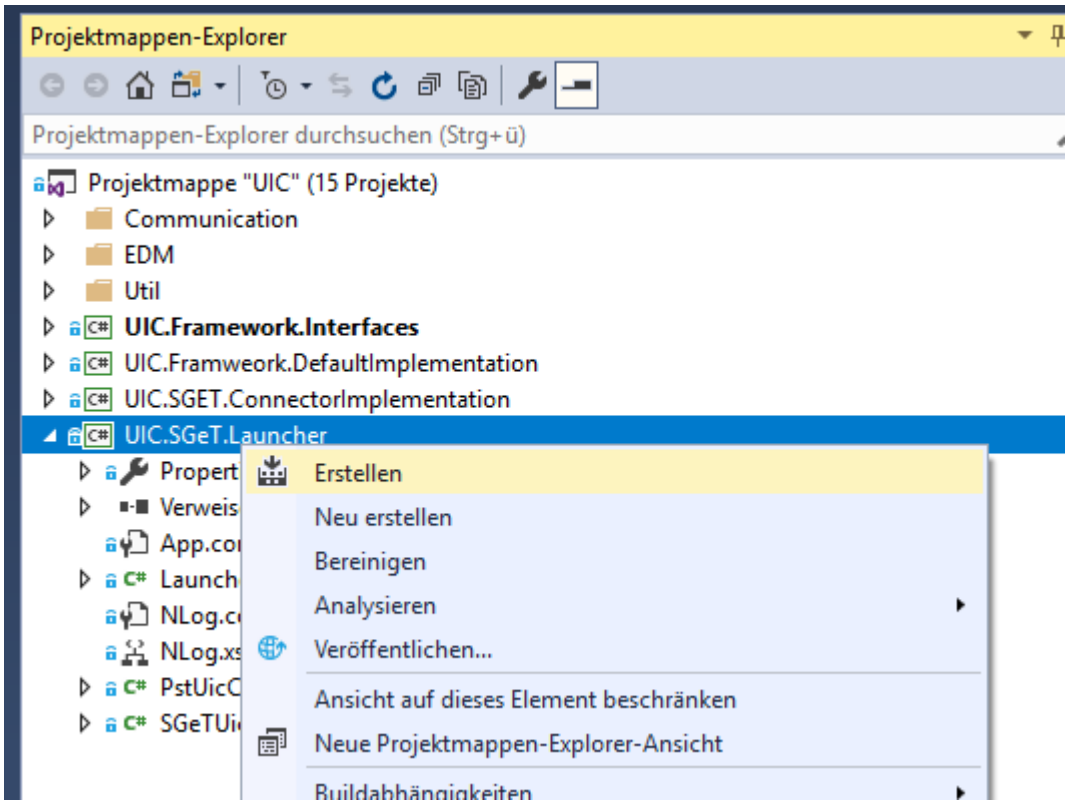
1. Clone the repository under https://github.com/sgetuic/AWS_CA/pulls
2. Open Visual Studio
3. Click File → Open → project map



4. Navigate in the UIC directory “\AWS_CA\c#\UIC” and select the “UIC.sln” file



5. In the project map - explorer right-click on “UIC.SGeT.Launcher” and select build



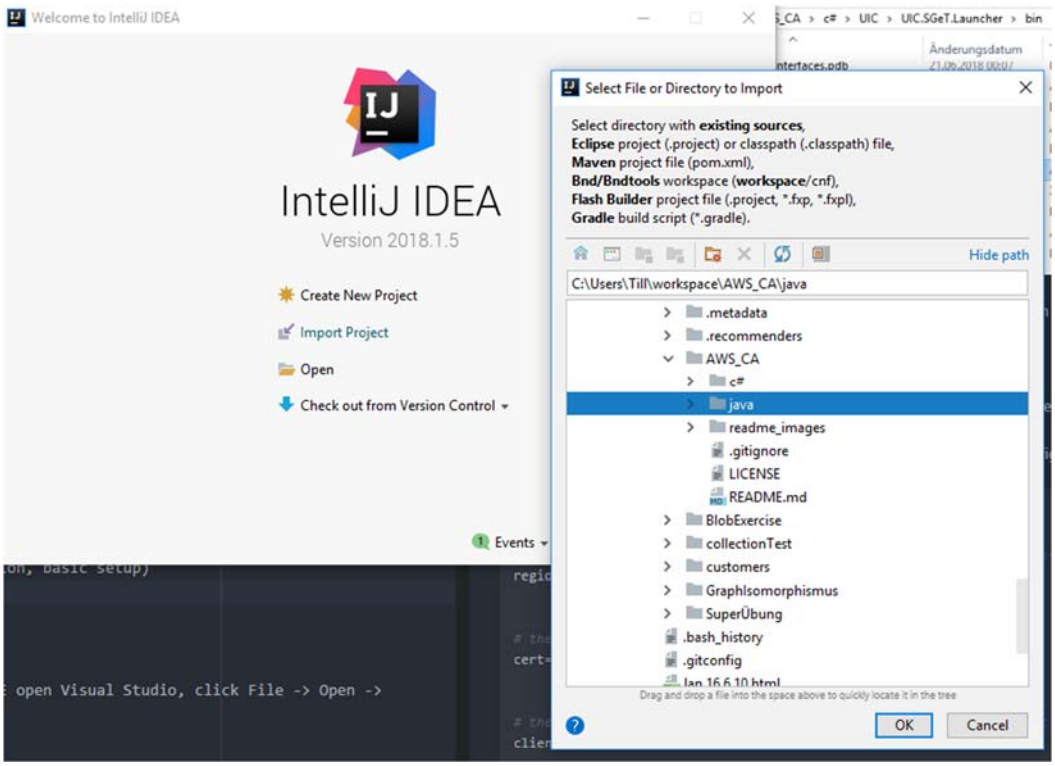
6. Wait until Visual Studio is done building the executable file
7. The “UIC.SGeT.Launcher.exe” can be found under “AWS_CA\c#\UIC\UIC.SGeT.Launcher\bin\Debug”
8. You can start the UIC by double-clicking this file

IMPORTANT:

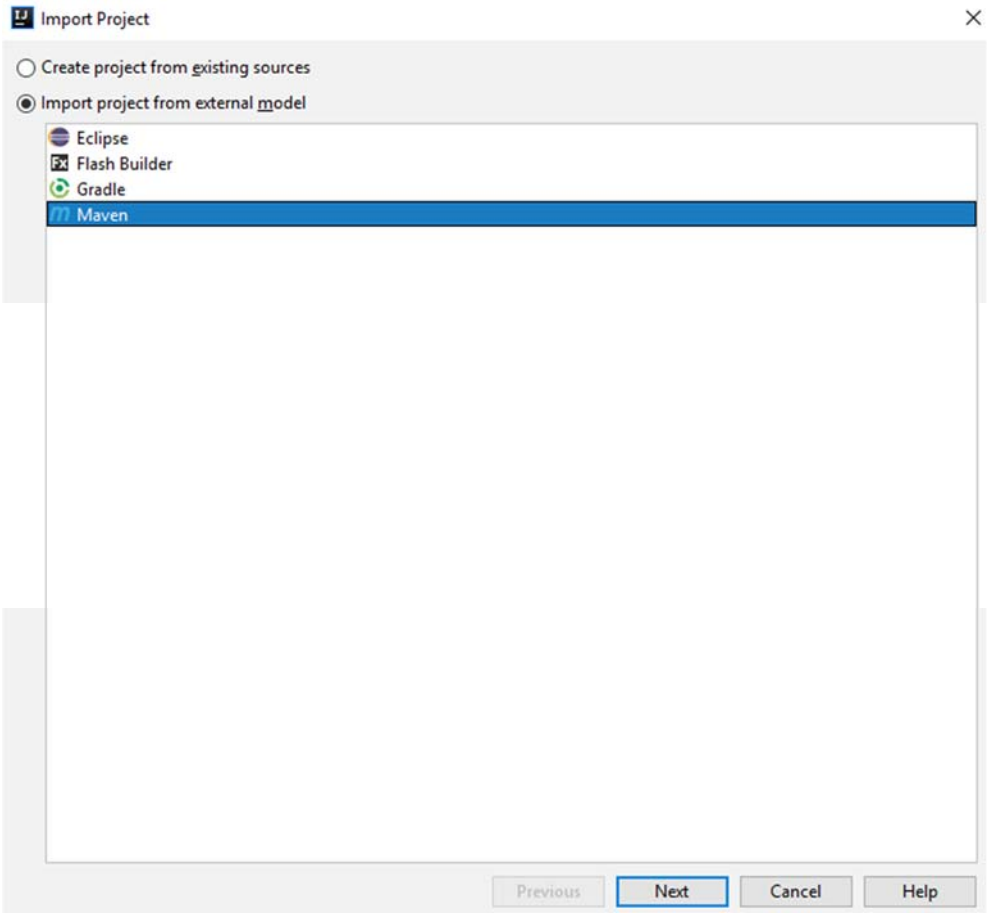
The UIC needs to run in admin mode to be able to start its REST API. As a consequence the root directory of the console will be the system32 folder. You will need to write the command cd <directory of the exe> as the first line to be able to start it via a batch file.

2. Setting up the UAS

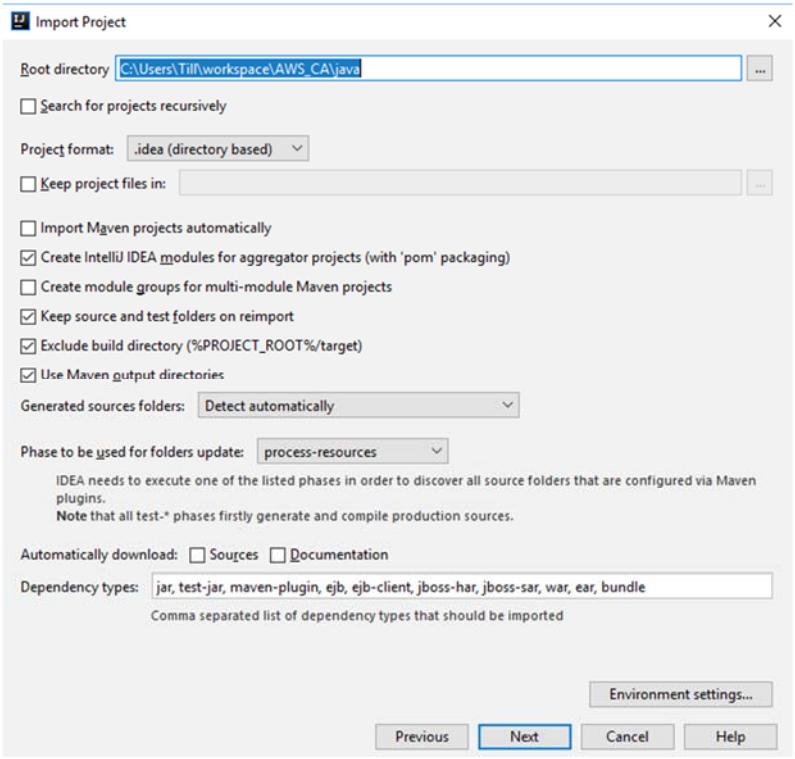
1. Clone the repository under https://github.com/sgetuic/AWS_CA if not already done in “1. Setting up the UIC”
2. Start the IntelliJ IDE and choose import project
3. Select \AWS_CA\java as project to import



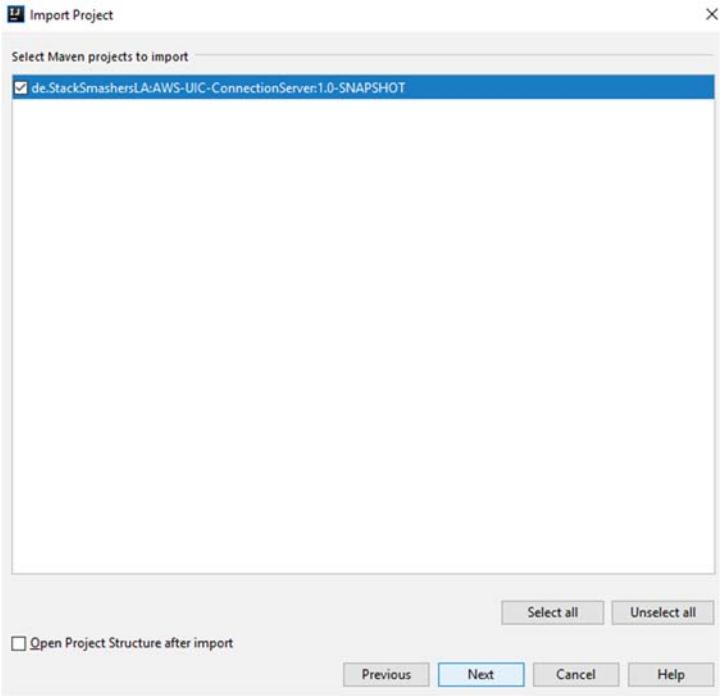
4. Choose “Import project from external model” and select “Maven”



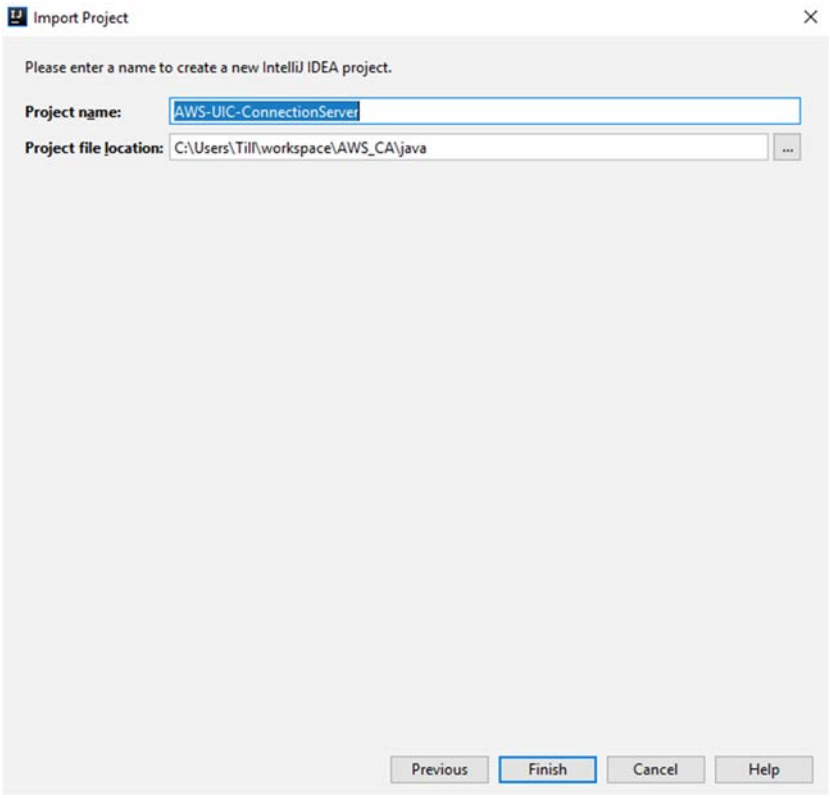
5. Check that the settings of the project resemble the ones shown in the following image



6. Select the Maven project de.StackSmashersLA:AWS-UIC-ConnectionServer:1.0-SNAPSHOT and press “Next”



7. Press “Finish” to finally import the maven project of the UAS

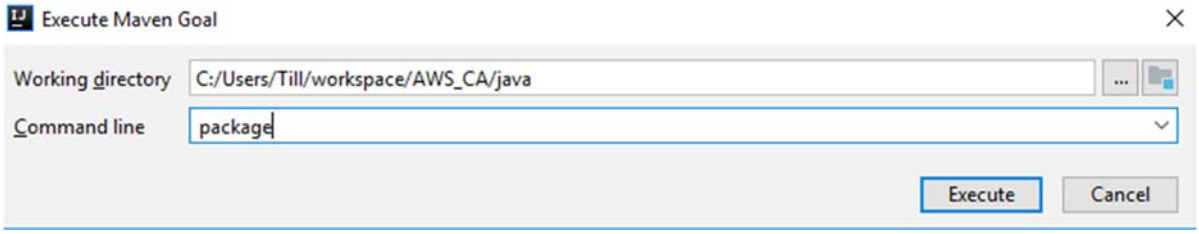


8. Wait until the Maven project is imported. The progress is shown on the lower right corner of the IDE.

9. Open the Maven Projects explorer by going to View → Tool Windows → Maven Projects

10. Compile the UAS by pressing the “Execute Maven Goal” button

11. In the prompt write “package” as command and press “Execute”



12. The required .jar will now be built and can be found in the target folder under \AWS_CA\java

3. Generate PrivateKey and certificates from AWS

In order to register your device in AWS IoT Core, follow the directions of the following link:

https://docs.aws.amazon.com/iot/latest/developerguide/register-device.html

4. Create batches to start both applications

Example UIC:

IMPORTANT: Must be started as admin!

cd <filePath to the folder containing the UAS-jar>

./launcher.exe

pause

Example UAS:

Java 8.X and below:

cd <filePath to the folder containing the launcher.exe>

java -jar AWS-UIC-ConnectionServer-1.0.jar

pause

Java 9.X:

// Unfortunately you must add the option --add-modules java.xml.bind for java 9

cd <path to the folder containing the UAS-jar>

java --add-modules java.xml.bind -jar AWS-UIC-ConnectionServer-1.0.jar

pause

5. Configure

As the last step the UAS and UIC must be configured.

1.Copy the sample configuration files

There is a sample configuration located in the repository under Java/sample-configuration.

Copy the file **config.properties** to the **UIC-folder** containing the launcher.exe and to the **UAS-folder** containing the UAS-jar.

Copy the file **log4j2.xml** to the **UAS-folder** as well.

2.Set ports (can be ignored if you want to use the default ports 8080 for UIC and 8081 for UAS)

in both config.properties file set the port_uic and port_uas to your wished ports.

3.Set the certificate.file and private key.file

in the config.properties file of the UAS, set the value of the key “cert” to the relative or absolute path of the certificate file. It is recommended that the cert file is next to the jar file.

Further set the value of the key “private_key_file”to the relative or absolute path of the private Key file.

4.Set the broker url

As the next step you have to tell the UAS to which AWS Broker it should connect.

In our example the broker url is **data.iot.us-west-2.amazonaws.com**.

The praefix is here “**data”** and the region is ”**us-west-2”**.

In the config.properties file of the UAS, you must set the keys “praefix” and “region” accordingly:

praefix=data

region=us-west-2

6. Start the applications with the batches

At this point you should be able to start the batch files and both applications should work. The UIC should build default versions of the uic.json and the project.json and use mock-up EDMs.

IMPORTANT: Start the UIC batch in admin mode, because the privileges are needed for the REST API.

Conclusion

Our work

We provided the ability for the UIC to connect to the AWS Cloud and publish and receive data from the cloud. To do this we implemented a second application - the UAS - in Java and connected the HAW Communication Agent with it via REST API on both sides. This enabled not only the ability to publish sensor and device data but also to receive commands from the cloud to change device behavior via commands. We used up-to-date libraries like log4j2, JBoss´s RESTEASY and Jetty Server. Furthermore the use of the AWS libraries ensures a reliable connection to the AWS Cloud.

Possible users

With our solution hardware providers can provide a possibility for their customers to connect their hardware to the AWS Cloud, where they can implement their own business logic and control the behavior of their devices via the backchannel. For productive use cases a more advanced version of our Proof of Concept should be implemented.

Possible further steps

Further steps may include a dynamic loading of the wished Communication Agent without the need to adjust the launcher. If another Communication Agent is implemented it needs to be added to the if-else cascade. With more Communication Agents it might make sense to change the coding to a switch-case.

Another possible improvement would include the loading of the configuration from the AWS Cloud, for example from a S3 Bucket. The UAS could be extended for this functionality.

Because the current version of the HAW CA relies on the external component of the UAS the complexity level the HAW CA is increased and as a consequence the risk of errors. To solve this problem an own MQTT library which can communicate with the AWS Cloud has to be implemented. Alternatively, it is possible to wait for the release of an official AWS library in C#.

For productive environment use we recommend to implement automatic testing. Also a penetration test of the whole architecture should be performed to find possible weaknesses or vulnerabilities. Another important test case is a long term running test in a integration testing environment with a business logic in the cloud.