

ABC4.IO

Post Quantum Cryptography Message Gateway

Peter Waher

ABC4.IO Chile SpA, Francisco Soza Cousiño 610, Concón, V, Chile
peter.waher@ieee.org

Abstract. This paper describes how the ABC4.IO service can be used to set up a secure Message Gateway between two peers using Post-Quantum Cryptography (PQC) and End-to-End Encrypted messages. If the network allows, peers will connect directly, otherwise one or two message brokers will connect the peers. PQC is achieved using ML-KEM (FIPS 203) and ML-DSA (FIPS 204).

Keywords: Decentralization, Interoperability, Security, Transparency, IEEE P1451.99.

1 Introduction

ABC4.IO® is a service that dynamically creates APIs and interfaces based on signed smart contracts¹. It can be hosted in different environments depending on how it is used. It is built on-top of the IoT Gateway™², which provides architecture for secure communication using End-to-End Encryption and Peer-to-Peer communication for Smart City applications and infrastructure components. As the IoT Gateway supports the end-to-end encryption interfaces published by the Neuro-Foundation³, it supports post-quantum cryptography algorithms such as ML-KEM⁴ for post-quantum protected key exchange and ML-DSA⁵ for post-quantum protected digital signatures. All standardized models⁶ are supported, providing the gateway with 128-, 192- and 256-bit security strength (or security categories 1, 3 and 5) using post-quantum cryptography.

As such, ABC4.IO provides an additional level of programmatic security to existing services being hosted in the same environment. If ABC4.IO is hosted alone on the IoT Gateway, it can run as a simple message gateway. If it is run together with LILS.IS⁷, which provides a secure decentralized social network for human-to-human communication, it adds a component of secure machine-to-machine communication using the same infrastructure. If running on the TAG Neuron® or Neuro-Ledger®⁸, it provides a mechanism to publish secure public APIs using smart contracts.

In the first two examples, ABC4.IO runs in local area networks behind firewalls, and can thus be used to interconnect local services in different domains in a secure manner. In the third example, ABC4.IO can run on a public node publishing its services on the Internet.

2 PQC Demo Setup

A simple demonstration has been created to show how PQC-protected messaging works. You can run this demo by following the subsequent steps.

Establishing a secure connection for secure Human Messaging

For the purposes of creating a Post-Quantum Cryptography protected message gateway, where messages from one local area network are securely transported to a device in another local area network, we will use ABC4.IO in one of the first two installations. The simplest is by downloading Lil'Sis'® from LILS.IS, as it comes with a simple installer for Windows⁹ which includes ABC4.IO as an embedded service.

After installing Lil'Sis', including ABC4.IO, on two local machines, we will connect them by creating a “friendship” between the two machines. Each instance of Lil'Sis' will connect securely to the XMPP network¹⁰. This means they use XMPP brokers on the Internet to relay instant messages between each other. Each client connects outwards to their broker. You can select one of the featured brokers shown during the configuration. You can also install and host one of your own¹¹. The brokers *interoperate* to relay messages across domains. This is often referred to as *federation*. It means clients can reside securely, each behind a separate firewall controlled locally. XMPP brokers authenticate all participants, and forward address information in all messages, minimizing risk of spoofing. Ubiquitous encryption makes sure all Internet connections are encrypted.

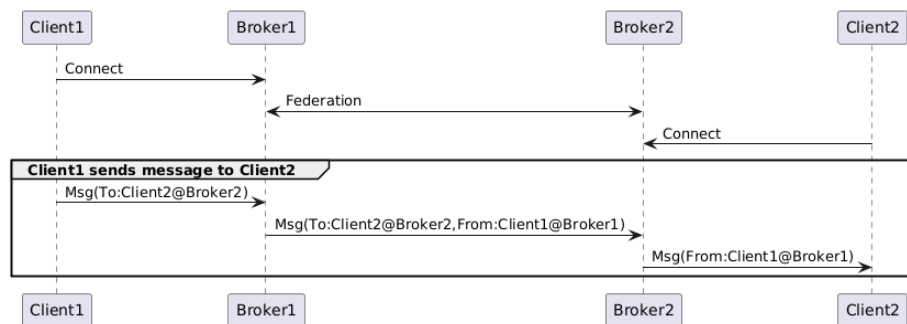


Fig. 1. XMPP: Federated communication network

Even though XMPP is a quite secure communication protocol over the Internet by itself, we will consider it an open protocol. It solves a network topology problem, allowing clients to communicate with each other in real-time even though they reside behind separate firewalls (if Internet connectivity exists). We use End-to-End encryption to ensure end-to-end security of messages. Transport encryption only protects connections. When a message is transported from one point to another it typically

needs to be passed over various connections. In each jump, from one connection to another, the message will be decrypted and re-encrypted for the next leg. This creates a vulnerability, allowing operators of each node to access the message, unless it is end-to-end encrypted. End-to-end encryption encrypts the entire message by itself, before sending it on the encrypted transport connection. This two-tier encryption ensures only the intended recipient can decrypt the message. The recipient can also validate that it was indeed the proposed sender that sent the message.

Once a client can connect to the XMPP network, it will receive a public XMPP Address called a Bare JID, which looks like an e-mail address. Once both installations have such an XMPP address, each one can “*subscribe to the presence of the other*”. This is an XMPP operation, and establishes a relationship between the two, if both accept the other’s request. Once accepted, the requesting party will learn of the online presence of the accepting party. In this online presence, public keys for End-to-End encryption will be present. These are required to communicate using End-to-End encryption. So, both parties will need to accept each other’s presence, for end-to-end encryption to be established.

During the installation and configuration of Lil’Sis’, you will get the XMPP Bare JID when connecting to the network. If you lose this information, you can easily find it again, by going selecting *Contacts* from the main menu, and *Connections*. At the bottom of the page that appears, you can see your address. You can also type in the Bare JID of the other party and press *Connect* to send a presence subscription to that address. The request will appear on the page as well, at the top, where you get a chance to accept it. Once the connection is established, it will appear in the list of contacts.

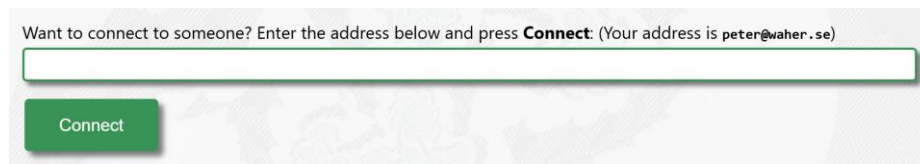
The screenshot shows a web interface for adding a connection. At the top, there is a text prompt: "Want to connect to someone? Enter the address below and press **Connect**: (Your address is peter@wahr.se)". Below this prompt is a long, empty text input field with a green border. Underneath the input field is a green button with the word "Connect" in white text. The background of the interface is light gray with a faint world map.

Fig. 2. Adding a connection

Viewing actual real-time communication

The IoT Gateway on which Lil’Sis’ runs allows you to view actual XMPP communications in real-time. This allows you to ensure communications are indeed end-to-end encrypted using PQC.

First, when opening a chat with a connection, the left-hand part of the view shows if the connection is secure or not. A lock symbol indicates the connection is encrypted end-to-end. A flash symbol indicates the connection is peer-to-peer, i.e. there’s a connection between the peers that is serverless (i.e. does not pass brokers). This second state is not required for end-to-end encryption but may be of interest. While end-to-end encryption will always be possible, peer-to-peer connectivity will only be available if the network permits.

To view the actual communication, select *Settings* from the *main menu*, and then *XMPP*. In the view that appears, find the button named *Sniffer*. It allows you to sniff on your own communication. When clicked, a new tab is displayed showing communication from your end, in real-time. End-to-end communication is indicated by elements as described by Neuro-Foundation interfaces for End-to-End Encryption¹². Post-quantum cryptography is indicated using references to Module Lattice ciphers for 128-, 192- or 256-bit security strength respectively, with the names `m1128`, `m1192` and `m1256`. White background signifies information sent. Blue background (unless selection) means information received. Green background is an informational note for you, i.e. in the example below, it contains the message before being E2E-encrypted. Red background shows errors.

Timestamp	Content
20:39:01	<pre><message type="chat"><body>Hi</body><content xmlns="urn:xmpp:content" type="text/markdown" n">Hi</content><html xmlns="http://jabber.org/protocol/xhtml-im"><body xmlns="http://www.w3.org/1999/xhtml"><P>Hi</P></body></html></message></pre> <pre><iq type='set' id='47' to='test_2025_03_09_1@lab.tagroot.io/6fcb9e4f0d4c3ccf94c90867a3ea56b'><qos:assured xmlns:qos='urn:nf:iot:qos:1.0' msgId='e466700d7e5923a51f204ed1fc27be4f'><message id='302bbfe5-25b7-678c-1473-583d82fe50b6'><aes xmlns="urn:nf:iot:e2e:1.0" r="m1128" c="7" k="c4nx2l9Oy8vE5D/ZksxlXkxhklVHnRPVwSlgQuluoBxr6H7UJtoEjix0VSS45/ECe6TXy8PfQ+Sxy+qHSDJfER0FVH1mVFELHf27f89vT0kCCsbikXN1pLINMdqL+WmrOqDhM8dxhWMCFUL8ixYMTUobi+gTktZbHipoLZIT7RVmIlrITWfG9fH4rIl2VfvtvfnGHDYhm/VVwMpeIiVfYK1lMa9w7l5i7u6MA9++lQrFn7h4WWWM2V7WwItv</pre>

Fig. 3. A sniffed E2E-Encrypted Message using ML-KEM-512 and ML-DSA-44 (`m1128`)

Establishing a PQC channel for machine-to-machine messaging

In the previous steps a secure connection was established between the two Lil'Sis' installations allowing for secure communication between two human actors. A method was presented whereby you could verify the security of the communication, verifying that it is indeed end-to-end encrypted using PQC. In this step, we will add a simple machine-to-machine message gateway between the two endpoints. We will do this by adding a *distributed API* using ABC4.IO.

A distributed API using ABC4.IO is an API that is instantiated in multiple hosts at once using a single smart contract definition. Different parts may be instantiated on different hosts, for different purposes, using a single document. In the following example, we will instantiate on one of the hosts a simple receiving REST API that receives incoming messages. On the other host an XMPP API will be created that receives messages from the first host. The first host ensures the messages are sent using End-to-End encrypted communication using PQC. If the second host can decrypt an incoming message, and it originates from the first host, it will save the message unencrypted to a folder on the machine. A very simple PQC-encrypted message gateway.

An ABC4.IO distributed API can be instantiated either via a smart contract that is signed by the parties involved, or using a definition file in the dedicated ABC4.IO Definitions folder for each participant. This first method is the preferred method in production environments. The second method is the simplest method used during development of ABC4.IO decentralized APIs. For the purposes of this demo, we will

use the second method. Download the following file to a folder on your local machine:

<https://abc4.io/Downloads/DemoPqcGateway.xml>

Once downloaded, edit the file, and enter the Bare JID of the first gateway (first Lil'Sis' installation) into the GatewayA_BareJID resource field. Likewise, enter the Bare JID of the second gateway (second Lil'Sis' installation) into the GatewayB_BareJID resource field. Enter the name of the folder where files should be created in the GatewayB_FileFolder resource field. The local resource name of the REST API is controlled by setting the GatewayA_WebFolder resource parameter. Once these edits have been saved, copy the file into the ABC4.IO definitions folders on both machines. By default¹³, this should be: C:\ProgramData\IoT Gateway\Definitions). Once the files have been copied, and no errors are found in the files, the APIs will be automatically created. You can open the real-time event log view if you are interested in monitoring what is happening under the hood¹⁴.

Using REST API to send PQC encrypted messages

Assuming that GatewayA_WebFolder is set to PqcDemo, you can access the REST API part of the decentralized API via /PqcDemo/Send on the domain or machine defined by GatewayA_BareJID (which can be a domain if run on a Neuron, or a Bare JID if run on Lil'Sis'). The resource is protected using the following two instructions in the definitions file. The first controls authentication:

```
<BasicAuthentication requireEncryption="true"
                        minStrength="128"/>
```

If you cannot access the endpoint using HTTPS, you need to set the requireEncryption to false. Note however, that BASIC authentication sends passwords in the clear, and should not be used if connection is not encrypted and should be avoided in production. ABC4.IO provides other better mechanisms for these purposes (such as Bearer token using JWT). But, for the purposes of the demo, BASIC authentication is simple to use and is available in most web clients.

The second instruction controls required privileges:

```
<RequiredPrivilege>ABC4IO.PQC.Send</RequiredPrivilege>
```

Once access instructions and privileges are defined, you need to ensure a corresponding user with corresponding privileges is defined. You define users and roles and their corresponding privileges under the Security menu in /Admin.md on the domain (Fig 4.). Once a user with the corresponding privileges is available, you can use the REST API. For the purposes of this demo, we illustrate this using Postman, a popular development tool that allows the user to make requests to REST APIs (Fig 5.). Press the Send button, to execute the call (Fig 5). The XML message included

will be PQC-encrypted and sent to the second endpoint, which will save the file. Make sure to check the communication to validate the communication is End-to-End encrypted using the appropriate PQC algorithm. Finally, check the reception folder (Fig. 6) for the file. PQC encryption not only encrypts the message but signs it also, ensuring the immutability of the contents during the transfer.



Fig. 4. Security Menu

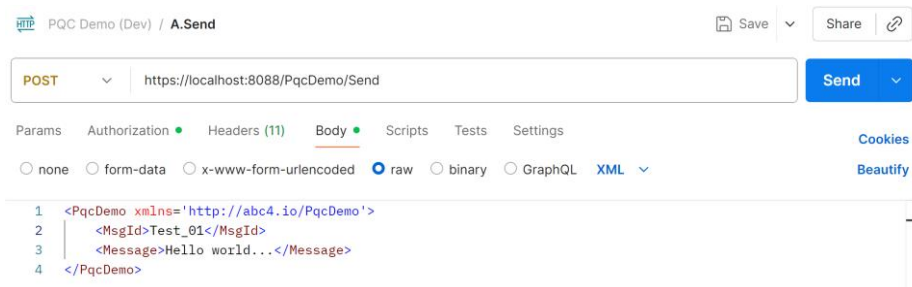


Fig. 5. Postman interface for testing REST API

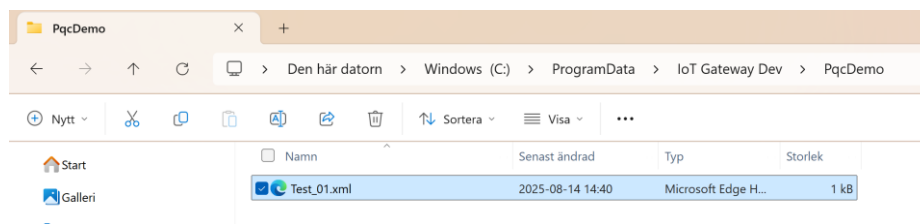


Fig. 6. The reception folder where transferred files will be stored.

3 Understanding the Decentralized API

The decentralized API created in this demo is based on a single XML document, or API description. The XML document can either be provided to the gateway as a file stored in the Definitions folder or be made available by signing a smart contract containing the file as its machine-readable payload. The XML document needs to comply with the ABC4.IO decentralized API XML schema, which can be downloaded from:

<http://abc4.io/v1.0.xsd>

Make sure to review this XML schema to learn what actions and control structures are available. By using a good XML editor, with XML schema support and code in-

sight as well as code completion is recommended. It will make your task of creating decentralized APIs much easier.

We will use the `DemoPqcGateway.xml` file, going through some important steps, to explain how the decentralized API is built. Changing the file and re-saving it will automatically take the old API down and bring the new API up. Removing the file will simply take the API down, in real-time. So, feel free to experiment with the file and see how the changes affect the operation of the decentralized API. Make sure to always have the event log open, to be able to fetch any errors introduced, and review the functioning of the API. Also make sure to log entries into the event log for debug purposes. You can remove these later when the API is done. Also note that white space has been inserted into the XML below, for readability purposes only. In the original file such white space is not available.

Entities

Each participant in the decentralized API is called an *Entity*. The root of the document is the `<Entities>` element.

```
<Entities xmlns="http://abc4.io/v1.0.xsd">
```

At the top, some common resources may be defined. This may include actual computational resources. It may also include constant definitions that will be reused throughout or API description.

```
<Resources>
  <ConstantString id="GatewayA_BareJID">
    peterdev@waher.se</ConstantString>
  <ConstantString id="GatewayB_BareJID">
    peterdev@waher.se</ConstantString>
  <ConstantString id="GatewayA_WebFolder">
    PqcDemo</ConstantString>
  <ConstantString id="GatewayB_FileFolder">
    PqcDemo</ConstantString>
</Resources>
```

These are the strings you had to change during the setup of the demo. The strings will identify the gateways involved and the roles they play.

Setting up a Web Server

On the entity we call *Gateway A*, we want to define a web resource called `/PqcDemo/Send`. This is a straightforward procedure. We simply define the entity, reference the web server inside it, define a web folder and then the resource we will attach logic to, as follows:

```

<Entity domain="{GatewayA_BareJID}" role="GatewayA">
  <WebServer>
    <Folder name="{GatewayA_WebFolder}">
      <Resource name="Send"
        endpointVariable="Endpoint">

```

Defining web pages

We can return web content from our resources, by responding to the HTTP GET method. In the file we define this as follows. For each method we define support for, we associate with an *Action* that will be executed if validation succeeds. In this case, we will simply return documentation on how the Send resource works.

```

<GET defaultAction="ReturnSendDoc"/>

```

Actions are defined later in the document under the <Actions> element. We can return a simple web page, using Markdown¹⁵, embedded as a CDATA construct, to avoid confusion with the encompassing XML document. The Markdown (as well as other sections of the API definition document) we use script¹⁶ to provide logic.

```

<Action id="ReturnSendDoc">
  <Return>
    <Markdown parseMeta="true">
      <![CDATA[Master: Master.md

```

```

=====

```

```

## Send

```

```

Send a `POST` with XML data (`Content-Type: applicat-
ion/xml`) to `/{GatewayA_WebFolder}/Send` to send an
End-to-End encrypted XML document from
`/{GatewayA_BareJID}` to `/{GatewayB_BareJID}`.
]]>
  </Markdown>
</Return>
</Action>

```

Once the page is defined and the definition file is saved, you can view the page in the browser (Fig. 7). The file Master.md controls the overall look and feel, theme and style usage, main menu, etc. Here, the default Master.md is used, provided by the ABC4.IO service.

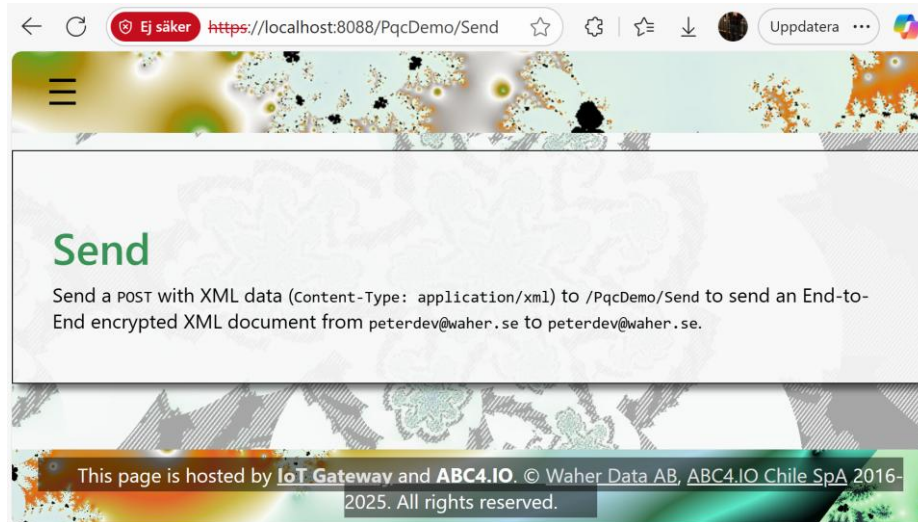


Fig. 7. Simple web page created by the decentralized API

Defining a POST REST web service

A POST REST API web service is added by adding a POST method definition, together with information about how to authenticate the client and validate the request.

```
<POST>
  <BasicAuthentication requireEncryption="true"
                        minStrength="128"/>
  <Content type="application/xml"
          action="ForwardMessage"
          variable="Message"
          requestVariable="Request">
    <RequiredPrivilege>
      ABC4IO.PQC.Send</RequiredPrivilege>
    <ValidationScript><![CDATA[
      MsgId:=select /default:PqcDemo/default:MsgId
                MsgId from Message
                ?? BadRequest("Missing Message ID");
      if MsgId not like "[a-zA-Z0-9_]+" then
        BadRequest("Message ID must be alpha numeric " +
                  "(underscores permitted).");
      ]]></ValidationScript>
    </Content>
  </POST>
```

We must map the representation of the data in the request to the internal structure we will use in the action script logic. Each Internet Content-Type to be supported needs its own `<Content>` element definition. While we will only use XML representation of the content sent to the web service, you could easily extend the web service to support other types of content as well.

Each Content-Type also defines required privileges by the authenticated user, and has its own validation script, used to validate the input before executing the associated action `ForwardMessage`. Note that it is easy to use XPATH directly inside script¹⁷.

Logging information to event log and Neuro-Ledger

For transparency reasons, or for purposes of debugging, you can log information to the internal event log. You can also perform custom logging to the Neuro-Ledger, if running on a Neuron. If you run the gateway using Lil'Sis', you only have access to a local event log, not the Neuro-Ledger. Any ledger entry instructions will be ignored.

To log information to the event log, use any of the `<Log*>` action elements. The example logs the reception of a successful REST API call to forward a message:

```
<Action id="ForwardMessage">
  <LogDebug body="Message received.">
    <Tag key="Source">{Endpoint}</Tag>
    <Tag key="MsgId">{MsgId}</Tag>
  </LogDebug>
```

Notice script is embedded in the element instructions between curly braces. Sometimes, when the element explicitly refers to script, such curly braces are not necessary. In the example, information extracted from the content in the call is logged to the event log, so you can match the event with the REST API call that was made.

An entry in a Neuro-Ledger can likewise be made using any of the entry elements, as follows:

```
<NewEntry archivingTime="365"
  collection="PqcDemo"
  mirrorInDatabase="true"
  objectType="MessageReceived">
  <Property name="MessageId">
    <Variable>MsgId</Variable>
  </Property>
  <Property name="Source">
    <Variable>Endpoint</Variable>
  </Property>
</NewEntry>
```

Sending a PQC-encrypted XMPP message

We can use the XMPP connection available by the hosting environment to easily send XMPP messages or make XMPP information query requests. Lil'Sis' is already connected to the XMPP network, so to send an XMPP message using the Lil'Sis' account, we simply execute the following instruction:

```
<Message to="{FullJid(GatewayB_BareJID)}"
      qos="Assured"
      e2ee="AssertE2EPQC">
  <Body>
    <Script><![CDATA[
      <Message
xmlns=https://abc4.io/Schema/DemoPqcGateway.xsd
      source=Endpoint
      msgId=MsgId>
      <[Message]>
    </Message>
    ]]></Script>
  </Body>
</Message>
```

The `qos` attribute determines the Quality of Service we want to use. The `Assured` service is the highest. It ensures that the message is delivered exactly once, even if retries are required to propagate the message. The `e2ee` attribute is set to `AssertE2EPQC`, which means that the message will only be sent if End-to-End encryption can be established using Post-Quantum Cryptography. The body contains script, embedded in a `CDATA` construct to avoid confusion with the XML document. The script dynamically generates the XML document that is encrypted and sent over the XMPP network. Note that it is easy to work with dynamic XML in script¹⁸.

Setting up an XMPP message handler

Setting up an XMPP message handler is like setting up a web server resource. We have to specify the entity that will set up the message handler. Instead of a web folder we define a *namespace*. We then define the message handler, and any validation script used to validate the received information. There is no client authentication necessary, since all participants in the XMPP network are already authenticated, and their identities are forwarded in all messages. This includes the broker identities, and the client identities associated with each broker. We will use this information to validate the source of the information, and ignore any message received from anyone other than Gateway A.

```
<Entity domain="{GatewayB_BareJID}"
      role="GatewayB">
```

```

    <XmppServer>
      <Namespace
ns="https://abc4.io/Schema/DemoPqcGateway.xsd">
        <MessageHandler name="Message"
          variable="Message"
          action="SaveMessage"
          endpointVariable="Endpoint">
          <ValidationScript><![CDATA[
            if BareJid(Endpoint)!=GatewayA_BareJID then
              Forbidden("Message source invalid.");
            MsgId:=select /default:Message/@msgId
              from Message
              ?? BadRequest("Missing Message ID");
            Source:=select /default:Message/@source
              from Message
              ?? BadRequest("Missing Source");
            if MsgId not like "[a-zA-Z0-9_]+" then
              BadRequest("Message ID must be alpha "+
                "numeric (underscores permitted).");
            if empty(Source) then
              BadRequest("Empty source.");
          ]]></ValidationScript>
        </MessageHandler>
      </Namespace>

```

Saving a file to a local folder

There is no action element that specifically saves contents to a file on the local machine. So, we need to use script to accomplish this. The script engine can interact with the .NET Core environment hosting the IoT Gateway on which the ABC4.IO service runs. We can use this to call .NET Core methods to save the file in the corresponding folder. To accomplish this, we do as follows in the corresponding action that gets executed when a valid XMPP message is received:

```

<Script><![CDATA[
  Folder:=System.IO.Path.Combine(
    Waher.IoTGateway.Gateway.AppDataFolder,
    GatewayA_WebFolder);
  if !System.IO.Directory.Exists(Folder) then
    System.IO.Directory.CreateDirectory(Folder);
  FileName:=System.IO.Path.Combine(Folder,
    MsgId+".xml");
  Body:=select /default:Message/* from Message;
  SaveFile(Body,FileName);
]></Script>

```

4 Summary

The ABC4.IO service can be used to easily create distributed APIs using a single source document for all entities involved in the distributed environment. Since the ABC4.IO service can be hosted on the IoT Gateway and Lil'Sis, or Neuron environments, all supporting End-to-End encrypted communications using Post-Quantum Cryptography, the ABC4.IO can be used to easily set up a PQC message gateway between participants in different networks across the Internet. While the simple example described in this document showed how to map a REST API to files being saved on another machine, the ABC4.IO definition is sufficiently flexible to allow for most types of integrations on either end of the gateway.

For more information, see <https://abc4.io/>. You can also return any feedback and request more information at <https://abc4.io/Feedback.md>.

¹ ABC4.IO, Executive Summary, 2021-08-25,
<https://abc4.io/doc/ABC4.IO,%20Executive%20Summary.pdf>

² IoT Gateway repository:
<https://github.com/PeterWaher/IoTGateway>

³ <https://neuro-foundation.io/E2E.md#postQuantumCryptographyPqc>

⁴ ML-KEM standardized by NIST in FIPS 203.

⁵ ML-DSA standardized by NIST in FIPS 204.

⁶ Supported module lattice algorithm models include ML-KEM-512, ML-KEM-768, ML-KEM-1024, ML-DSA-44, ML-DSA-65 and ML-DSA-87.

⁷ Lil'Sis' secure decentralized social network: <https://lils.is/>

⁸ On the TAG Neuron and Neuro-Ledger with associated technologies:
<https://www.neuro-tech.io/>

⁹ Download Lil'Sis': <https://lils.is/Downloads/LilSisSetup.exe>

¹⁰ XMPP is an open, flexible and secure protocol originally defined for instant messaging. It is standardized by the IETF in RFCs 6120, 6121 and 6122.

¹¹ Download and Install the Tag Neuron XMPP broker:
<https://lab.tagroot.io/Documentation/Neuron/InstallBroker.md>

¹² <https://neuro-foundation.io/E2E.md>

¹³ The default instance of IoT Gateway stores program data in the folder C:\ProgramData\IoT Gateway\. Multiple instances can be installed on the same machine. Each instance is identified by an instance name. The instance named ABC, stores its program data in C:\ProgramData\IoT Gateway ABC\.

¹⁴ The event log view can typically be accessed via the default URL:
<http://localhost/Sniffers/EventLog.md>.

¹⁵ Markdown reference syntax: <https://abc4.io/Markdown.md>

¹⁶ Script reference syntax: <https://abc4.io/Script.md>

¹⁷ On XPATH in script: <https://abc4.io/Script.md#selectFromXml>

¹⁸ On XML in script: <https://abc4.io/Script.md#xml>