

CS-455 Cloud Computing

Project 1

Objective

The objectives of Project 1 are:

- Continue practicing using AWS services. In this project you will combine many of the concepts we have looked at to do a simple cloud application (S3, Lambda, RDS, XML, and JSON).
- Learn how S3 Event Notifications work. S3 can publish events to many destinations (including a Lambda function). Read this [article](#) about S3 Event Notifications. We also did a module where we configured a Lambda with an S3 trigger (see Module 12).
- Learn how to interact with a relational database running in the cloud.
- Learn how several cloud services can be used together.

Scenario

In the spirit of current events, we will do a scenario related to COVID vaccination. You will devise a system where a central authority (e.g., the *Center for Disease Control and Prevention* (CDC)) is tasked with keeping track of the vaccination drive taking place across the US. The collected information is useful to monitor areas of the country that are making progress vs. others that are not, adjust policies accordingly, and keep the public informed about the vaccination progress (e.g., by publishing daily numbers in major newspapers).

Let's start with a few assumptions:

1. The CDC is the central authority that aggregates and keeps track of vaccination data. It does that by collecting **daily** information that covers the following:
 - a. Total number of people who have received their first shot.
 - b. Total number of people who have received their second shot and are considered fully vaccinated.
2. There are thousands of sites across the US administering vaccines (pharmacies, clinics, hospitals, etc.). These sites do not have systems in place to directly communicate with the CDC computer systems. All sites, however, have Internet connections and can upload data to the cloud.

3. The CDC created an AWS S3 bucket and requested all sites to upload at the end of each day a summary of the vaccines administered in each site. The CDC gave sites the option of uploading their data in XML or JSON format. Examples of the expected format are shown in Figure 1.

XML:

```
<data month="4" day="22" year="2021">
  <site id="1000223">
    <name>Bellevue Overlake Hospital</name>
    <zipCode>98004</zipCode>
  </site>
  <vaccines>
    <brand name="Pfizer">
      <total>630</total>
      <firstShot>400</firstShot>
      <secondShot>230</secondShot>
    </brand>
    <brand name="Moderna">
      <total>420</total>
      <firstShot>300</firstShot>
      <secondShot>120</secondShot>
    </brand>
  </vaccines>
</data>
```

JSON:

```
{
  "date" :
  {
    "month" : 4,
    "day" : 22,
    "year" : 2021
  },
  "site" :
  {
    "id" : "1000223",
    "name" : "Bellevue Overlake Hospital",
    "zipCode" : "98004"
  },
  "vaccines" : [
    {
      "brand" : "Pfizer",
      "total" : 630,
      "firstShot" : 400,
      "secondShot" : 230
    },
    {
      "brand" : "Moderna",
      "total" : 420,
      "firstShot" : 300,
      "secondShot" : 120
    }
  ]
}
```

Figure 1: XML and JSON data examples the CDC is expecting from vaccination sites.

- day, month, year: date of vaccination.
- id: a unique ID the CDC gives each participating site.
- name, zipCode: site name and its location zip code.
- brand: vaccine brand name (assume only two allowed: Pfizer and Moderna).
- total, firstShot, secondShot: total number of people who got a shot at the site in a given day, number that it is their first shot, number that it is their second shot (firstShot + secondShot should be equal to total).

- Note that not every site administer both vaccine brands. Some sites offer Pfizer only, some offer Moderna only, and some offer both. Therefore, the XML does not necessarily have to have two <brand> elements under <vaccines>. Same applies to the JSON equivalent: the vaccines array may contain only one item.

System Design

Your task is to implement a cloud application for the CDC that works like this:

- At the end of each day, vaccination sites upload their daily summary (in XML or JSON format) to an S3 bucket that the CDC created. Each site uses a simple program (`UploadData.exe`) to upload the XML or JSON file. The `UploadData.exe` executable is called from the command line of any computer connected to the Internet. It takes two arguments: The first argument is the path of the file to upload. The second argument is a string (xml or json) that indicates whether the file is XML or JSON. Here are two examples of how `UploadData.exe` may be called (text in blue is the first argument, and text in red is the second argument):

```
prompt> UploadData.exe C:\Temp\daily.xml xml
```

```
prompt> UploadData.exe C:\vaccination\data\today.json json
```

`UploadData.exe` uploads the file passed in the first argument to the S3 bucket and sets with the uploaded object a tag named “type” with a value that indicates if the file is XML or JSON (“xml” or “json”).

- An AWS Lambda function is listening on file arrival to the S3 bucket (using an S3 event notification mechanism). When a file is uploaded to the bucket, S3 triggers the invocation of the Lambda function. Lambda can then read the file content and the “type” tag, parse the file content, extract vaccination data, and enter the data into a PostgreSQL relational database.
- The PostgreSQL relational database schema should have two tables: **Sites** and **Data**. Since a given site uploads daily data, you should have a one-to-many relationship between the “Sites” and “Data” tables (use a primary-foreign key combination).

- The **Sites** table should have the following fields:

SiteID:	The site ID (an integer - can be used as the primary key of this table).
Name:	A string that represents the site name.
ZipCode:	A string that represents the site zip code.

- The **Data** table should have the following fields:

SiteID:	A foreign key (also of type integer) to the Sites’ table SiteID primary key.
---------	--

Date: The data of vaccination
FirstShot: An integer that represents the first shot.
SecondShot: An integer that represents the second shot.

Note that the SiteID and Date (together) form the primary key of the **Data** table. That is, the combination of SiteID/Date should be unique.

No need to keep track of vaccine brand. So there is no field for it.

- On any given day, it is possible that a site makes mistakes and need to upload a corrected XML or JSON file. Your system should allow this; in this case a row in the **Data** table should get UPDATED (not INSERTED since you already have a row for the SiteID/Date combination).

Figure 2 is a diagram that summarizes the workflow of this system.

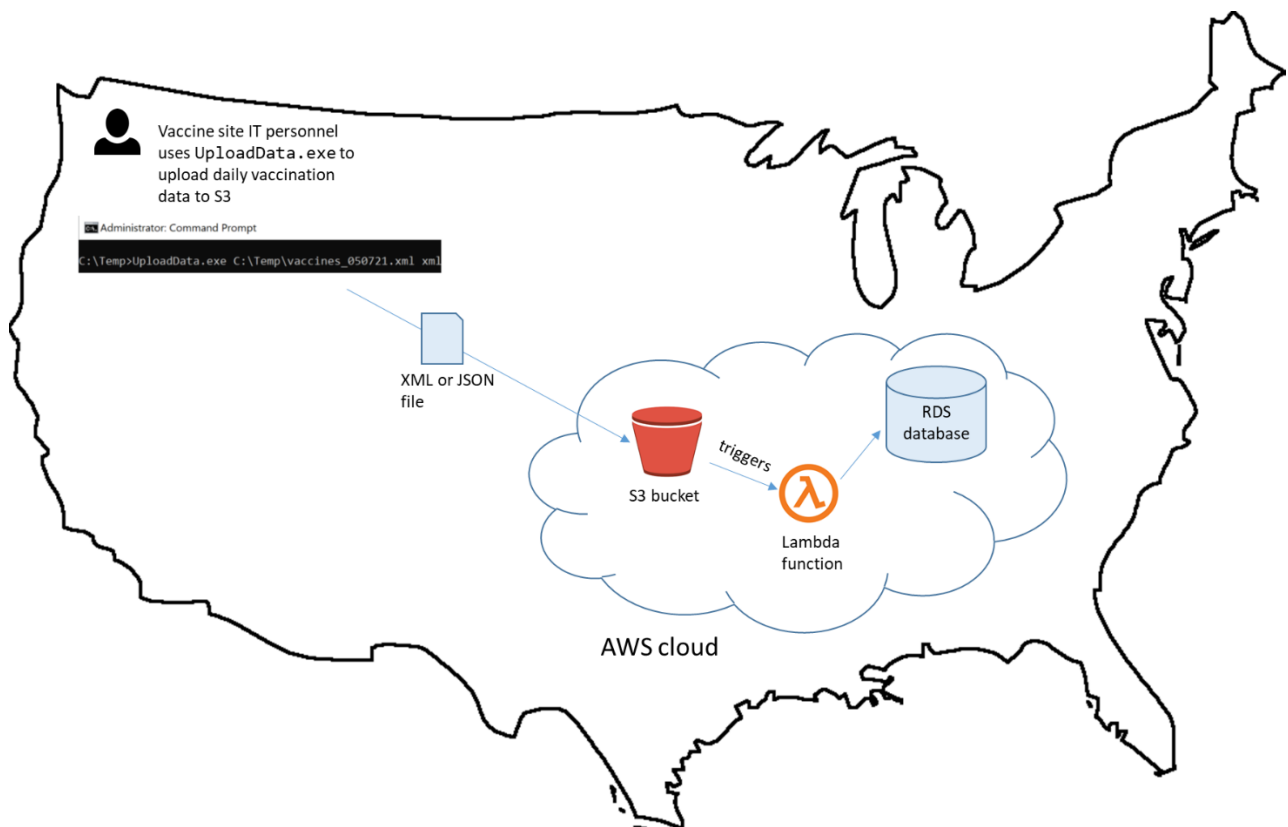


Figure 2: System workflow

Files to Use for Testing


8 files were provided to you for testing. See folder TestFiles.

Implementation Hints

1. In order to query the S3 objects tags you need privileges beyond what the *AWSLambdaExecute* policy gives you. To do that, create a custom role and use that role for your Lambda function.
 - a. Go to the **IAM service**. Click on **Roles**. Then click the **Create role** button.
 - b. Under Choose a use case, select **Lambda** then click the **Next: Permissions** button.
 - c. Search for the following two policies and check each:
 - i. *AWSLambdaExecute*
 - ii. *AmazonS3FullAccess*
 - d. Click the **Next:Tags** button, then the **Next:Review** button.
 - e. Give your role a name (e.g., *CDCVaccinationSystemRole*).
 - f. Click the Create role button.

This creates a role named *CDCVaccinationSystemRole* with two policies attached to it: *AWSLambdaExecute* and *AmazonS3FullAccess*.

Now when you use the publish Lambda function wizard in Visual Studio, choose role *CDCVaccinationSystemRole* for your function (you do this in the second screen - this new role that you created in your account should appear in the dropdown):

 Upload to AWS Lambda

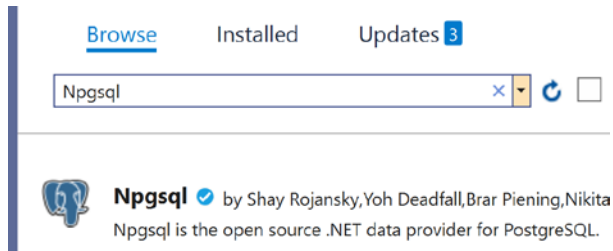


2. You need to read the tag “type” associated with the S3 file that was uploaded to S3. Its value will tell you whether the file is of type XML or JSON. You can read the tags associated with an S3 object using a combination of the following classes/methods:

```
GetObjectTaggingRequest  
GetObjectTaggingResponse  
GetObjectTaggingAsync
```

This [documentation article](#) shows you an example how it is done (expand “Use dot AWS SDKs” then click the .NET tab). Sorry, I wasn’t able to find the English version of this doc page.

3. To query the PostgreSQL database from the Lambda function, use the **Npgsql** library. You need to add this package to your Lambda function in Visual Studio.



Cheating Policy

This is an individual project and you should present your own code and work. If I notice unusual resemblance between the codes of two students, both students will get a 0 (the student who gave help and the student who received help).

What to Submit?

There are no files to submit.

I will grade your project by having you give me a quick demo that shows your system is working start to end. I will also observe your UploadData and AWS Lambda code and ask you questions about the overall system. I can do that one-on-one via a 5-minutes Teams meeting that I will do with each student.

Grading Rubric

1. Workflow works successfully (that is, data gets inserted into the relational database after files are uploaded to S3 using utility UploadData.exe) using a variety of upload possibilities. **60 points.**
2. You are parsing XML and JSON using industry standards (e.g., XPath for XML). **10 points.**
3. UploadData is well commented with all steps clearly explained. **5 points.**
4. AWS Lambda function is well commented with all steps clearly explained. **10 points.**
5. You can explain how the overall system works and how the different pieces connect. **15 points.**

Late Submissions Policy

1. Up to 2 days late: 25% penalty.
2. More than 2 days late: not accepted.