mediavalet

TECHNICAL INTERVIEW EXERCISE

2019/09

1 Overview

Thank you for applying to MediaValet and we really appreciate the time and effort you have been investing in the interview process and in this exercise. We believe that a developer's code says a lot about the coder and this step in the recruitment & selection process helps us know every candidate better. This exercise also serves as a springboard for discussion for the technical interview with members of the team.

Although the Coding Exercise below has a very specific deliverable, the completion of the exercise is not required. To us at the MediaValet Product Engineering Team, we consider coding as our "craft". We would like to make sure that the people we work with treat their "craft" with the same level of thoughtfulness and enthusiasm as we do.

This exercise is designed to help us understand how you develop, the level of quality you put in, how you solve problems, and how quickly you understand new technologies. We expect from you that you act in this exercise in the same way you would act during actual assignments: follow general development best practices, generate understandable code, avoid anti-patterns, and be consistent with the code style and naming conventions you use (these don't have to be MediaValet's, just generally accepted conventions).

This exercise will include the use of the Microsoft Azure Storage and development in C#.

All code must be in C#. Don't spend too much time on this exercise, if it's not fully functional but still developed in a fashion that you consider to be nice and proper, this is enough for us to get the information we need. It normally takes .NET developers without prior knowledge of Azure 1 ½ to 2 ½ hours to complete this exercise.

2 Technical Requirements

- Visual Studio or Visual Studio Code (https://visualstudio.microsoft.com/)
- Microsoft Azure SDK including the Azure Storage Emulator (see https://docs.microsoft.com/en-us/azure/storage/common/storage-use-emulator)

3 Coding Exercise

3.1 Deliverable

Your assignment is to create a solution which is composed of one **Console Application** and one **Web API** that addresses the user story described in 3.2. You can create projects and structure them within the solution folder as you see fit. This solution folder should be zipped and emailed back to the person who sent you this exercise. This will be unzipped and ran in a dev environment that has the current version of the Azure SDK running with the Azure Storage Emulator. So the connection strings must point to the emulator.

3.2 User Story

As a 3rd party application, I should be able to send orders to an Order Supervisor and receive a response for the processed order including the Agent App ID that processed the order. There is one Order Supervisor and there could be one or more Agent Apps processing the order.

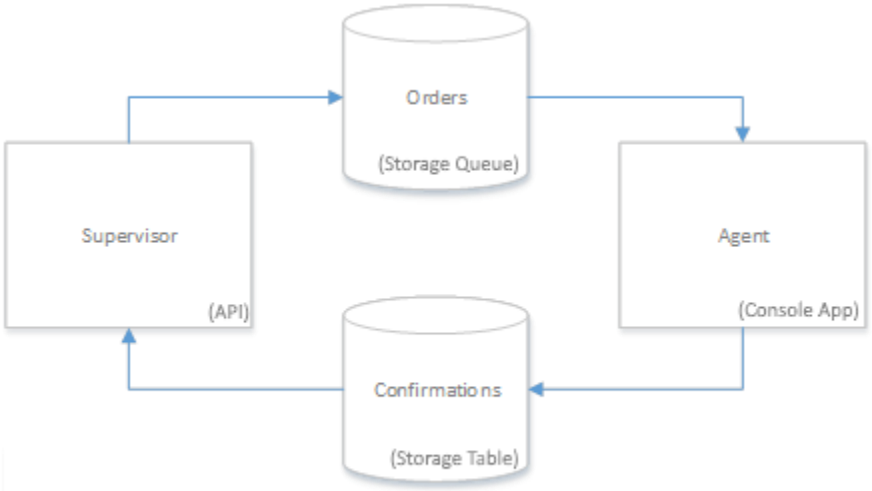
- The Order Supervisor will receive order requests and sends it to a queue.
- Agent Apps get orders from the queue, display a message corresponding to the order, then write a confirmation to be read by the Supervisor.
- Supervisor's response to the order request must contain the Agent App ID that processed the message.

3.3 Technical Specifications

The Order Supervisor is implemented as a Web API and the Agent App is implemented as a console app. For demo purposes, developers can use whatever tool they would like to send the requests to the Order Supervisor.

- **Supervisor** will be the Web API receiving and routing orders to a storage queue
- **Agent App** will be a Console Application executing orders and sending confirmations back to supervisor.
- There will be only one instance of Supervisor. There may be several instances of Agent. Each of these instances is identified by a Guid, generated during startup of the Agent App.
- These applications will communicate through one Azure Storage Queue named **orders** that will contain order commands for the agent to execute.
- The Agent should write to an Azure Storage Table named **confirmations** that the Supervisor will read from.

The cycle is depicted below:



Orders are entities containing the following fields:

- OrderId, an incremental number
- RandomNumber, a random number between 1 and 10
- OrderText, a string

Confirmations are entities containing three fields

- OrderId, corresponding to the order that was served
- AgentId, corresponding to the Guid of the agent having processed the message
- OrderStatus, a string

3.3.1 Order Supervisor

The Order Supervisor is responsible for keeping the order counter up to date (initial value must be 0). It can either use an internal counter or store it externally. The Supervisor also connects to the queue ("*orders*") and creates it if it doesn't already exist.

It then creates the command to be processed by the App Agent.

When sending the message to the "orders" queue, the Order Supervisor also displays a message on the console saying, "Send order #*OrderId* with random number *RandomNumber*".

3.3.2 Agent App

The Agent App starts, connects to the "orders" queues and creates it if doesn't already exist. It generates a unique *AgentId* which is a Guid, and a *MagicNumber*, which is a random number between 1 and 10. It then displays a message in the console window saying "I'm agent *AgentId*, my magic number is *MagicNumber*".

Then it goes into an infinite loop: agent polls from the "orders" queue. When a message is received:

- Deserializes it
- Displays in the console "Received order *OrderId"*
- If the RandomNumber in the message is equal to the MagicNumber selected during startup, then the application displays in the console: "Oh no, my magic number was found", and it exits the loop (and wait for a key to be entered)
- Else the application processes the message:
    - Display the Order.OrderText
    - Store a confirmation, with OrderStatus set to "Processed", to the "confirmations" table
    - Delete the message from the queue using DeleteMessage
- Resume polling messages.

3.4 Additional requirements

- Access to local resources (e.g. console display) must be made through proper locking mechanism
- Messages must be deleted only after they are processed
- Use a nice project and class structure
- All code must be C# (.NET or .NET Core are acceptable)