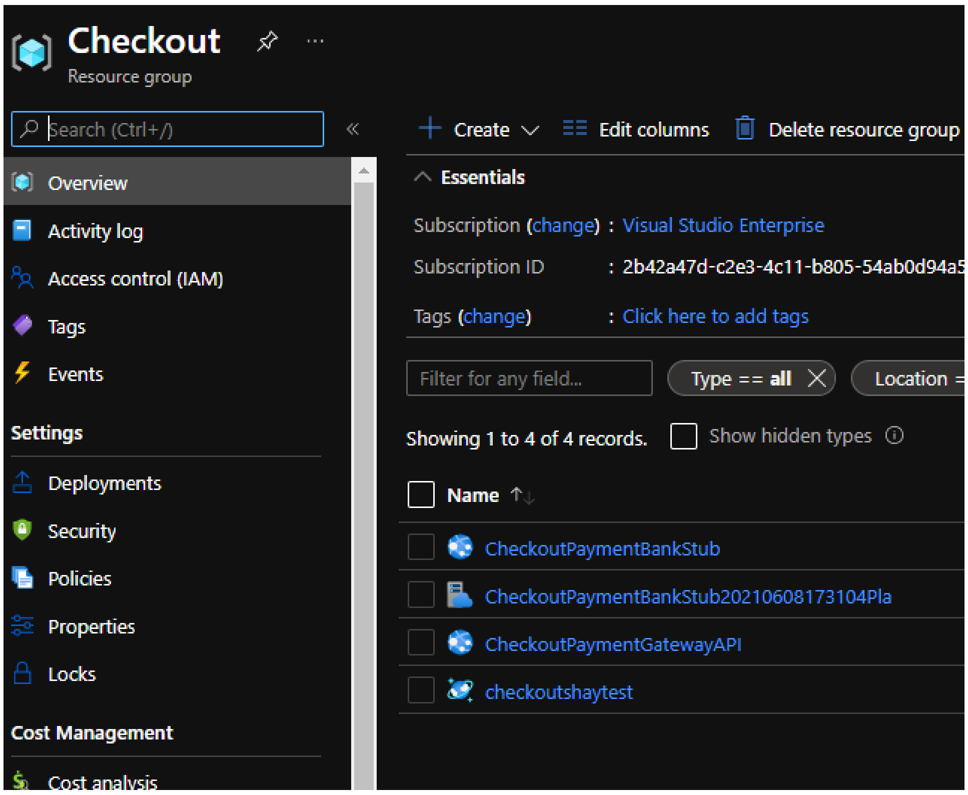
Shahzad Emambaccus Payment Gateway Checkout Tests

Technologies used:

- Cosmos DB
- Azure app services
- Azure hosting plan
- Visual Studio
- Swagger contract documentation

The image shown below is a resource group of the services deployed in Azure. There are two apps' services one that holds the API itself and the second one holds the stub for mimicking bank response.



Subscription (change) : Visual Studio Enterprise
Subscription ID : 2b42a47d-c2e3-4c11-b805-54ab0d94a5
Tags (change) : Click here to add tags

Showing 1 to 4 of 4 records.

Name
CheckoutPaymentBankStub
CheckoutPaymentBankStub20210608173104Pla
CheckoutPaymentGatewayAPI
checkoutshaytest

Example API Call without running locally:

Make Payment Endpoint:

https://checkoutpaymentgatewayapi.azurewebsites.net/swagger/index.html (Swagger contract)

https://checkoutpaymentgatewayapi.azurewebsites.net/Payment/ProcessPayment?CardNumber=11111111111111&NameOnCard=wqeq&Expiry=02%2F21&Amount=213&Currency=gbp&Cvv=123&postcode=n8djs

Get Payment Endpoint:

https://checkoutpaymentgatewayapi.azurewebsites.net/swagger/index.html (Swagger contract)

https://checkoutpaymentgatewayapi.azurewebsites.net/Payment/GetPaymentDetails?identifer=258b4708-289c-4c6a-b885-27ddb188f684

Bank Stub:

Within the bank stub it returns either a successful or unsuccessful response dependant on a random number this is to generate a few unsuccessful bank transactions. The bank stub also returns the unique identifier.

Assumptions made:

- The Gateway API makes a call to the Bank API, if the bank does not respond or throw an error then we do not save the response in the database.
- The fields that are needed for a payment are: name on card, card number, expiry date, cvv, post code, currency, amount and identifier. The identifier is used to identify the purchase in the other end point for the API.
- Card number is 14 digits long.
- There is a maximum transaction amount of 999999.99.
- Post code is 6 characters long.
- Added a health check, this applications will be deployed in the cloud. Health check is needed to check if the application is up and running. A service as Azure application insights would automatically log and display in life graphs the health stay, if the application does go down Azure application insights would alert.
- Since there were no acceptance tests, I created wrappers for dependencies so that classes can be unit tested.
- All object models have been created in a separate project but it within the solution called contracts this could be used to create an SDK and used in other projects.
- The bank creates the unique identifier.
- There is a 1-1 mapping with transactions to the bank. Therefore every transaction is unique and does not know about any other transactions from the same user.
- The partition key used in the CosmosDB is currency. This allows for a finite list of sharding and storing the data.
- The gateway pattern was required for this application.

Other Notes:

- Using appsettings to store the connection string to the bank, this could be swapped out at for a productionized bank URL.
- Using dependency injection for interfaces. By adding singletons at on start of the application.
- Using nunit for unit test.
- Using Polly retry to retry calling the bank API in case there's an error.
- Using the DRY so that code does not get repeated.
-

DockerFile:

- Within the Dockerfile I've started from a base image for Microsoft dotnet core 5 Alpine as this is a common image used for deploying dotnet applications on cloud, also the application is running on dotnet core 5. I've also passed in a few arguments such as version a pat token and a username. This would allow the CI/CD pipeline to pass in different parameters as they could change between environments. There's also the run commands which will run a few credential and updates within the image. I then set out a working directory, next I start copying the solution and all other projects including their folder structures from the repository into the container. Then run a dotnet restore provided the URL to Nuget. Finally for this layer I copy over everything else. Within the next layer I run a dotnet build on the main project, under dotnet publish. In the final layer I pulled down the same image from the start, copy out the main build and expose the application with an entry point. This will allow the application to run on something like Kubernetes.

Improvements:

- Knowing how to mask an integer in a Json response. Instead of having to convert the integer to string then manually replacing values with stars. This will allow for the contract to not have many redundant classes and fields within them.
- Not having to write wrapper classes. These classes that use wrappers should be covered by acceptance tests.