**Rule Processing Architectural Style and Consideration**

The “RuleEngine” application is broken down in several loosely coupled, async behavior driven component which perfectly fits for cloud based solution. The given problem statement was very straightforward where the whole logic has been put into single component and mostly running in local system. The solution has been created in Azure “PaaS” platform and inherently highly scalable with in built support of load balancing, data replication and long running task handling.

REST API

“Apply R6”

Table entries of job information

Client Component

POST file content for rule to be applied



Blob storage

RuleEngine (WebJob)

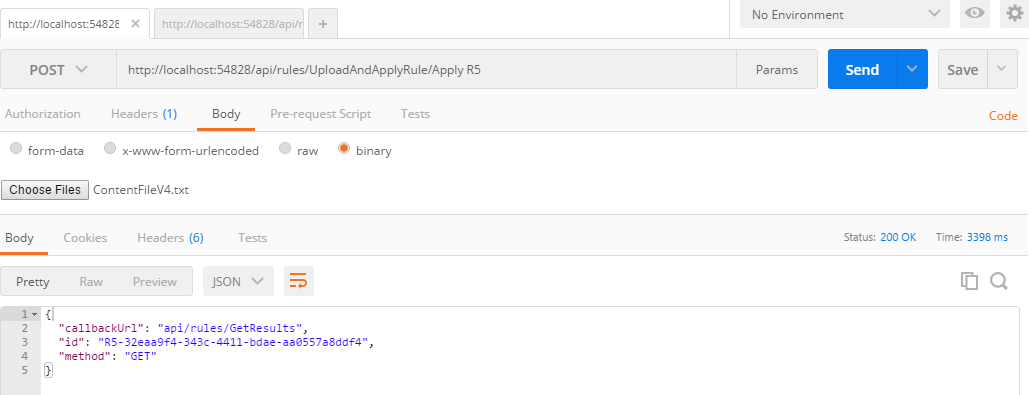
There are primarily 3 components,

1. One REST driven Azure API service (**ASP.NET Core 1.1**)
   1. This one can be both horizontally (based on CPI performance or custome config.) and vertically (based on service plan tiers) scaled
   2. Client application can consume the service and post the content file with rule command e.g. “Apply R5” to the service
   3. The rules and rules text are easily preconfigured in a json file “appsettings.json” instead of text file. This makes application both highly machine and human readable content to configure
   4. The service supports runtime rule change or adding new rule using built in configuration notification changes without any service downtime
   5. The service loads data asynchronously the file to Azure Blob storage, create job entry in Table storage with rule information and returns with “Callback” information for retrieval of processed data output
   6. Followed Async-Await pattern to minimize resource contention and latency

**Performance Consideration** – Usage of ASP.NET Core MVC was purposefully done as it has performance benchmark quite higher compared to other backend server framework. We have specifically considered here HTTP pipeline based approach, not TCP .

Reference - <https://github.com/aspnet/benchmarks>

Example -



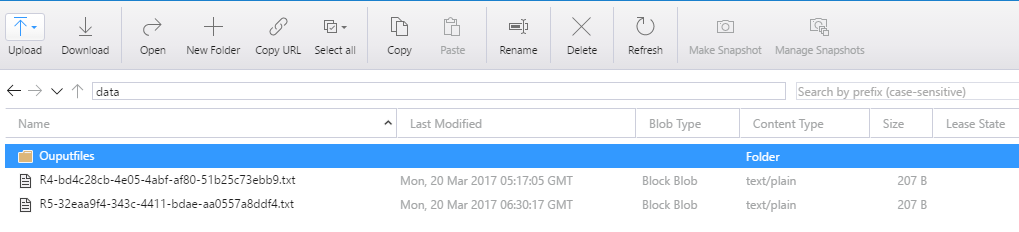
1. **Azure Cloud Storage**

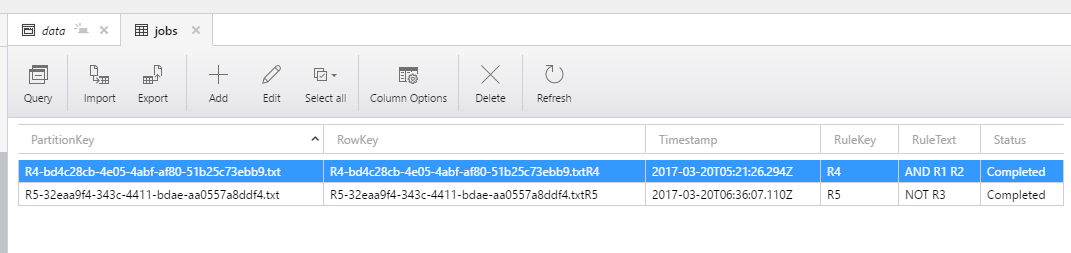
To support large data file processing we have used Azure Cloud “**Blob**” storage, with encryption enabled, “hot” mode for smooth read/write operation and replication strategy as LRS. As business context is not very clear here, but we can scale application like GRS or LA-GRS. There would be trade off between pricing and performance (reference - <https://azure.microsoft.com/en-in/pricing/calculator/>)

We have used “**Table**” storage which is very cheap and highly scalable (till TB) perfectly fits for simple data entry operations e.g. here we have job information

The job goes through several stages like “Created”, “Running”, “Completed”. This allows end user to monitor the job status at the same time help application to filter out unnecessary processes.

Ex –





1. **Azure Web Job**

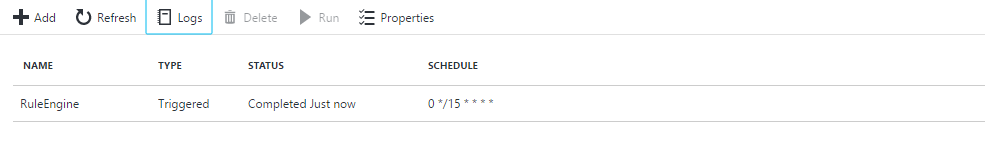
Azure web job acts as schedules asynchronous batch processing service and can leverage both the “Scaleup” and “Scaleout” feature of Azure web apps where it has been hosted. We have used a simple console application as a “Job” which triggeres every 15 min (configurable) and laso can be triggered manually from Azure web portal console. The cron process fetches the job information from Azure table and start processing for newly “Created” jobs.

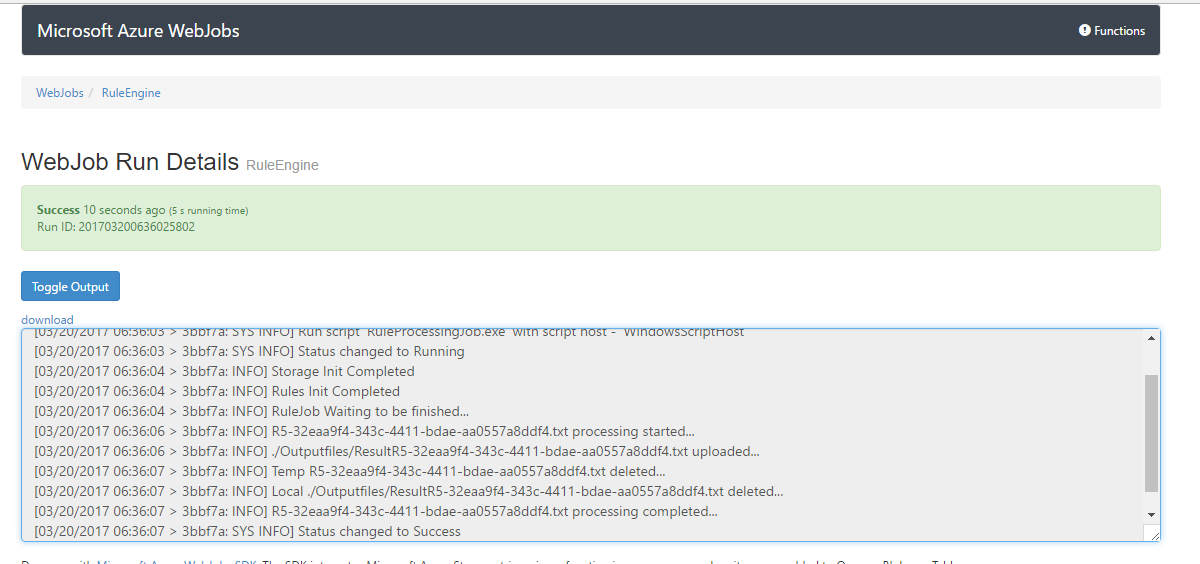
**Multithreading** - The application itself uses Task parallel Library(TPL) to leverage multi cores in Azure cloud platform.

**Note** – Applicability of TPL is also dependent on “Sevice plan” selec ted for the web apps where web job hosted. If this is not a multicore machine, then the TPL usage will not be beneficial though it will work as multithreaded application.

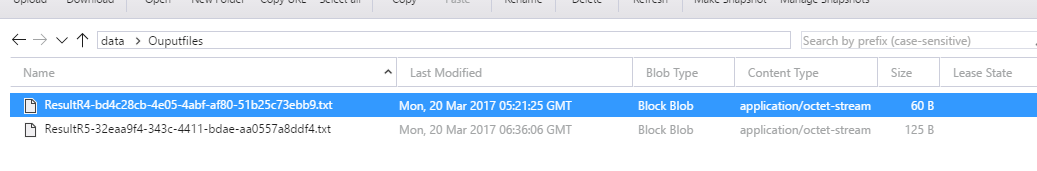
The status of the webjob can be monitored with inbuilt Kudu logger support. Any issue can be tracked and fixed.

Ex –

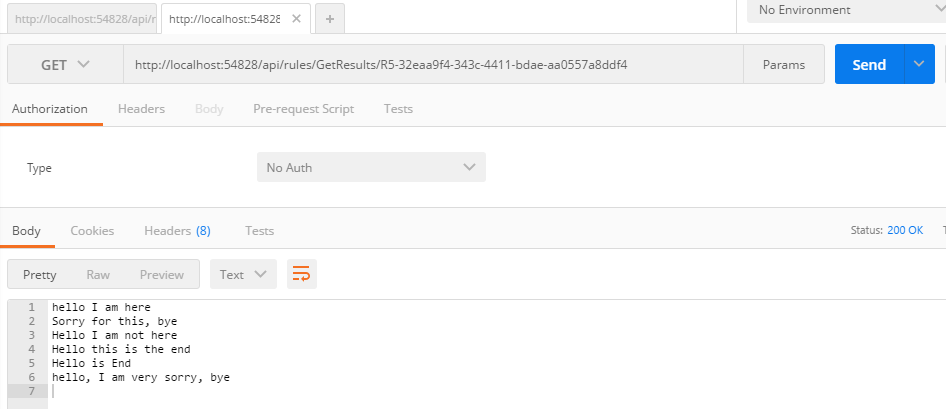




Once web job successfully process the content data based on rule, it will push the output processed files to Blob Conatiner “Outputfiles” location as below.



Any client system now can use the callback url to get the output processed data from the same REST based interface. **Note** – if client system tries to retrieve the data using the callback API url and job not finished, then REST will return empty string



1. **.NET MEF Framework Implementation (Extension and Pluggability)**

MEF framework makes the pluggability of new rules or changing new rules business logic very simple. When Web job starts it loads all the rules components from “bin\Debug\RulesCatalog” and make it ready for rule execution. So to add a new rule or modify an existing rule the following 2 steps required,

1. Change the “appsettings.json” in API service without any downtime
2. Package the new or modified dll in zip and push as web job. This will not impact any existing data processing or any downtime of overall application due to it’s highly decentralized, loosely coupled architecture

**Consideration –** We have not written any explicit client code like console/windows or web to consume REST API. Testing has been done with “Postman” chrome plugin. Here we wanted to focus on the design and approach of the architecture and even it is flexible enough to be used from simple mobile app or javascript.

**Constraint (partial) –** One constraint of this architecture isto upload very large file like in GBs. As we are using REST based services there might be a limitation of file size based on request timeout, cloud connection timeout, underlying virtual machine settings etc. We can improve the file size limit till certain MBs.

From the processing point of view, the web job can process very large file size (in GBs) due to it’s multithreaded, TPL model and chunk wise file processing. Also it leverages Azure webapps backend for scaling.

**Alternative Approach (DevOps style)**

**Alternative Approach 1** - We can bypass the filesize limitation during file upload, if we follow a simple client application (like console or windows) and use Azure Cloud storage SDK directly to chunk the file and push chunk wise to blob. One sample client library has been created “LargeFileBlobClient” to handle this kind of scenario. We have taken a MSDN community page reference to write this client component.

**Alternative Approach 2 –** We can follow manual steps to bypass the filesize upload limit.

1. Upload large data file to Azure cloud blob with application rule naming convention using Azure storage explorer (client app)
2. Add a job entry to table storage with similar convention

That’s all