UDC 004.4´2
DOI: 10.18523/2617-3808.2020.3.36-41

I. Morenets, A. Shabinskiy

SERVERLESS EVENT-DRIVEN APPLICATIONS DEVELOPMENT TOOLS AND TECHNIQUES

Serverless, a new cloud-based architecture, brings development and deployment flexibility to a new level by significantly decreasing the size of the deployment units. Nevertheless, it still hasn't been clearly defined for which applications it should be employed and how to use it most effectively, and this is the focus of this research.

The study uses Microsoft Azure Functions – one of the popular mature tools – because of its stateful orchestrators – Durable Functions. The tool is used to present and describe four flexible serverless patterns with code examples.

The first pattern is HTTP nanoservices. The example demonstrates how flexible can be the Function-as-a-Service model, which uses relatively small functions as deployment units.

The second usage scenario described is a small logic layer between a few other cloud services. Thanks to its event-driver nature, serverless is well-suited for such tasks as making an action in one service after a specific event from another one. New functions easily integrate with the API from the first example.

The third scenario – distributed computing – relies on the ability of Durable Functions to launch a myriad of functions in parallel and then aggregate their results. and distributed computing. A custom MapReduce implementation is presented in this section.

The last pattern described in this research significantly simplifies concurrent working with mutable data by implementing the actor model. Durable Entities guarantee that messages are delivered reliably and in order, and also the absence of deadlocks.

The results of this work can be used as a practical guide to serverless main concepts and usage scenarios. Main topic of future research was chosen to be the development of a full-fledged serverless application using typical patterns to study the architecture in more depth.

Keywords: serverless, Function-as-a-Service, Microsoft Azure, event-driven, cloud computing.

Introduction

Serverless, also known as Function-as-a-Service (FaaS) and nanoservices, is a promising new way of building elastically scalable distributed applications. Best engineers are looking for ways to utilize this technology in the most profitable ways, but it's still often looked down upon as not as powerful and mature as other architectures, for example, microservices. The main goal of this work was to study FaaS in detail and prove that in the future it can become a de-facto standard for building many types of applications.

Serverless analysis

FaaS features

Serverless can be similar to Platform-as-a-Service but it has several fundamental differences. One of the main ones is the significantly smaller deployment units. While Platform-as-a-Service typically operates full applications, Function-as-a-Service uses separate functions. This allows scaling each function independently which can be very useful, for example, when a service has read and write operations and the first one is significantly more popular [8].

Another dissimilarity with PaaS is that serverless is inherently event-driven. They react to certain predefined events such as a timer tick, a new message in the queue, or an HTTP request. During the lifecycle, they launch, run, and free allocated resources. The following and previous instances are completely independent.

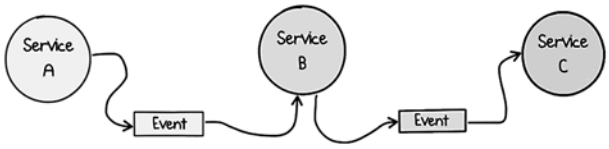
This is connected with another important Serverless feature – developers pay not for some dedicated computation units, but for the time their functions were running. With the ability to scale separate functions this allows for a significant expense optimization [8].

Also, a serverless function cannot just call another one directly but has to do it asynchronously with special tools. Transition to purely asynchronous communication is not simple and can be a substantial drawback depending on the application.

FaaS caveats

Nanoservices have to use message queues for communication which can negatively impact not only its simplicity and clarity but also the speed and reliability.

When the number of such functions grows it becomes hard to comprehend their connections. Any function can react to an event sent by another function which provides great potential for changes and scalability because of loose coupling. At the same time, even a simple sequential flow can be hard to track.



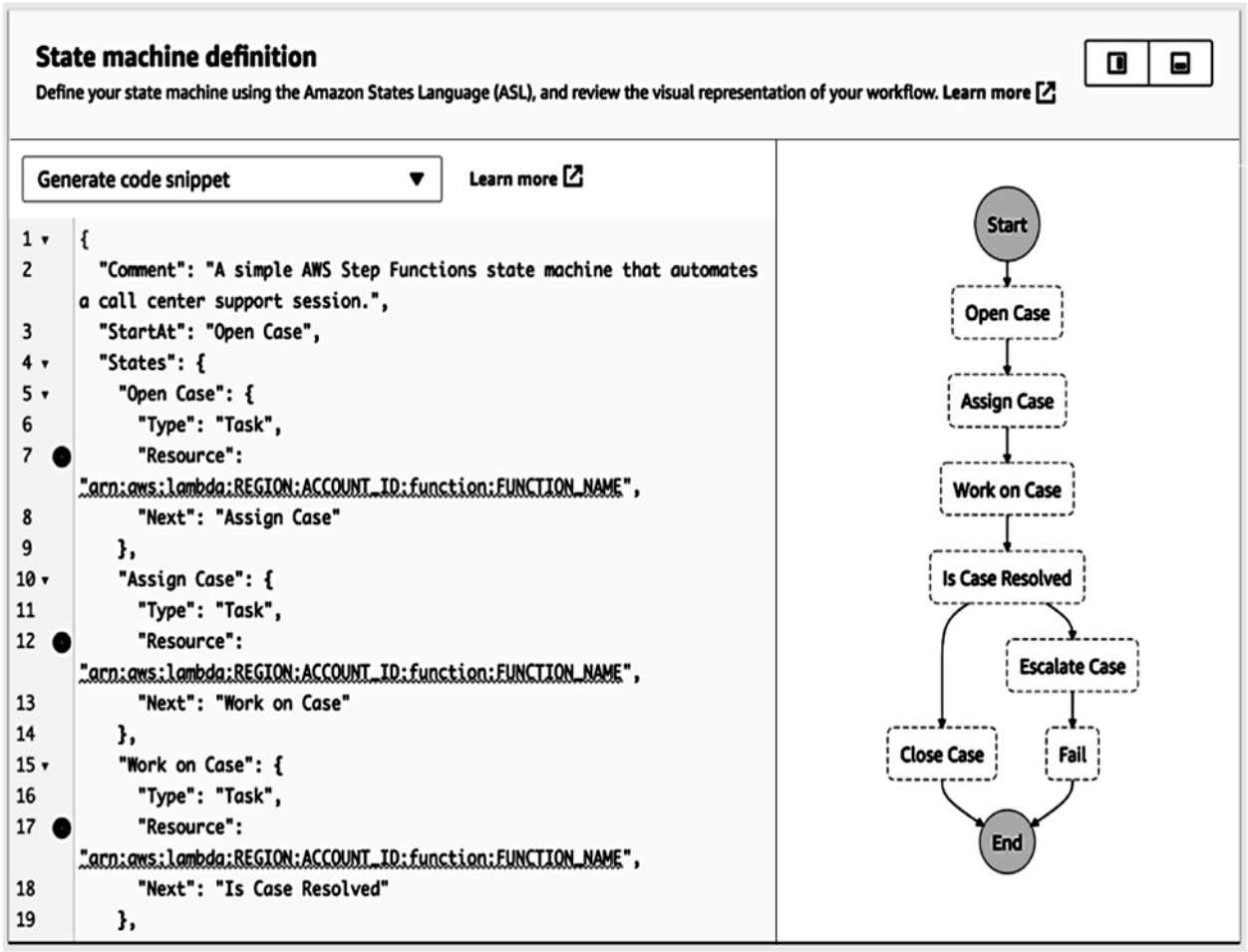
Picture 1. Sequential serverless functions execution [9]

More complex scenarios are even harder to implement with simple FaaS. For example, error-handling and retries are stateful operations (as each try needs to know about the previous ones) but serverless typically is stateless. A FaaS solution might have built-in ways of solving such problems but they are not always flexible enough. For instance, it might be desirable to set an exponential back-off strategy or specify which fallback functions to run in case of errors [9].

This and similar problems are often solved by orchestration – specification of multiple functions' execution order and conditions in one place. It allows not only to easily track and modify a functions chain but also to implement more complicated patterns such as error-handling, retries, task parallelization, and others. Orchestration in one way or another is supported by most main serverless providers [9].

Main providers

AWS Lambda and Microsoft Azure Functions are considered to be the leaders in the field, but Tencent, Google, Cloudflare, and Alibaba are also powerful players [3]. However, Microsoft has an advantage – Durable Functions, stateful orchestrators. Usually, serverless orchestration is done on a high level, such as with a finite state machine in AWS Step Functions.



Picture 2. AWS Step Functions orchestration state machine [1]

Microsoft not only has an alternative for such a high-level approach in the form of Azure Logic Apps but also more low-level Azure Durable Functions which allow describing complex scenarios in code. This approach can be used to implement more flexible and powerful patterns so it was chosen for a detailed study.

Azure Functions introduction

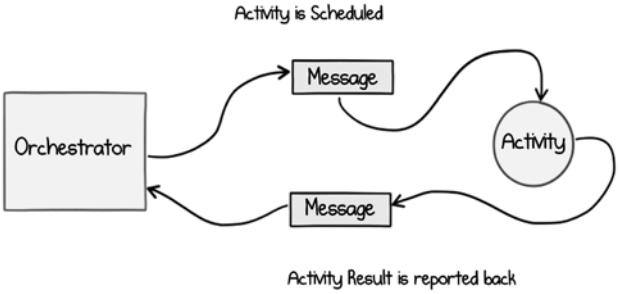
Azure Functions are a part of the Azure cloud. As of the first quarter of 2020, they support multiple mainstream programming languages: C#, F#, JavaScript, TypeScript, Java, Python, and Powershell. At the time it was apparent that bigger priority was given to .NET and JavaScript, so all following patterns were implemented with C#.

Triggers are a way to launch an Azure Function. Examples are a timer, an HTTP request, a new message in a queue, new entry in the database, or any other supported event [4]. To connect a function to another resource for additional inputs or outputs so-called bindings are used. They allow writing to a queue or a DB, returning an HTTP response, and doing many other operations [4].

Durable Functions introduce four new function types:

- Orchestrator – defines the order and the conditions of actions' execution. Abstracts calling activities and sub-orchestrators, signaling entities, and doing other actions.
- Activity – similar to ordinary functions but can only be called by an orchestrator.
- Durable entity – operates on some internal state, can respond to incoming signals by changing it, returning something, or sending signals to other entities.
- Client function – react to triggers and can launch orchestrators and signal entities [5].

Neither orchestrators nor client functions really "call" other functions, but rather abstract working with message queues.



Picture 3. Orchestrator "calls" an activity function [9]

To save intermediate state orchestrators use event-sourcing, so they have to be deterministic. It's prohibited to use random number generation, getting local date and time, generating GUIDs, making HTTP requests, and doing other non-deterministic operations in them [7].

Patterns

HTTP nanoservices

The first pattern is HTTP nanoservices. Azure Functions have a special HTTP request trigger and an output binding allowing to return a response, but other providers also usually support this scenario. A simple blog backend was used for a demonstration, including three operations: getting a post by its identifier, creating a new post, and deleting a post.

Request for a post by an identifier can look like this:

```
public static IActionResult GetPost(
  [HttpTrigger(methods: "get", Route =
  "posts/{author}/{id}")] HttpRequest req,
  [CosmosDB(
        databaseName: "MyBlog",
        collectionName: "Posts",
        PartitionKey = "{author}",
        Id = "{id}"
  )]
  Post? post,
  string id
) =>
  post == null
   ? new NotFoundObjectResult(new {message
   = $"Can't find a post with id {id}"})
   :(IActionResult) new OkObjectResult(post);
```

The function receives an HTTP request, the requested post or null (if not found), and the id from the request path. Additional Cosmos DB bindings allow implicitly making a DB request with attributes metadata.

Post creation function is a bit more complicated:

```
public static async Task<IActionResult> AddPost(
  [HttpTrigger(methods: "post", Route =
  "posts")] HttpRequest req,
  [CosmosDB("MyBlog", "Posts")] IAsyncCol-
  lector<Post> collector
)
{
  var data = JsonConvert.DeserializeObject<
  RequestBody?>(
    await new StreamReader(req.Body).
    ReadToEndAsync()
  );
  await collector.AddAsync(new Post(data.
  Author, data.Body, data.Title));
  return new OkResult();
}
```

This function receives a collector which allows writing multiple entities into the database. But even if the framework doesn't provide helpful bindings,

it's always possible to use a DB client, as demonstrated in the deletion operation example:

```
public static async Task<IActionResult>
DeletePost(
  HttpTrigger(methods: "delete", Route =
  "posts/{author}/{id}")]  HttpRequest req,
   [CosmosDB("MyBlog", "Posts")]
   DocumentClient client,
   string author,
   string id
)
{
  var collectionUri = UriFactory.CreateDoc
  umentCollectionUri("MyBlog", "Posts");
  var posts = client
      .CreateDocumentQuery<Document>(collec
      tionUri)
      .Where(d => d.Id == id);

  foreach (var doc in posts)
  {
    await client.DeleteDocumentAsync(
     documentLink: doc.SelfLink,
     options: new RequestOptions {Partition-
     Key = new PartitionKey(author)}
    );
  }

  return new OkResult();
}
```

This code looks more like a regular application but with some Cosmos DB peculiarities. A client can be used instead of any previously demonstrated DB bindings to build more complex flows.

Service glue

The most classic serverless usage scenario is a small logic layer between a few other cloud services. Thanks to their event-driver nature FaaS are well-suited for such tasks – after a specific event from one service make an action with another one.

For example, it's possible to send email notifications to an author's subscribers when she makes a new post by using a special mailing service:

```
public static async Task Emailer(
  [CosmosDBTrigger("MyBlog", "Posts")]
  IReadOnlyList<Document> posts,
  [CosmosDB("MyBlog", "Posts")]
  DocumentClient client,
  [SendGrid(ApiKey = "SendGridKey")] IAsyn
  cCollector<SendGridMessage> collector
)
{
  var authorsUri = UriFactory.CreateDocume
  ntCollectionUri("MyBlog", "Posts");

  foreach (var doc in posts)
  {
    var author = doc.GetPropertyValue<string>
    ("Author");
    var messages = client
      .CreateDocumentQuery<Author>(authorsUri)
      .Where(a => a.Name == author)
      .Select(a => a.Subscribers)
      .ToList()
      .FirstOrDefault()
      .Select(
        subscriber => ComposeMessage(
          from: "i.morenets@ukma.edu.ua",
          to: subscriber,
          author: author,
          title: doc.GetPropertyValue<string>
          ("Title")
        )
      );

    foreach (var message in messages)
     await collector.AddAsync(message);
  }
}
```

This example uses a Cosmos DB trigger and two bindings – an output for SendGrid and a universal for Cosmos DB, which is another name for a client.

Distributed computations

Many tasks can be parallelized, from running multiple independent business operations to processing and transforming massive amounts of data. With ordinary serverless, it's usually not a problem to launch many parallel functions, but it's not so trivial to aggregate their results. This problem can be solved using orchestration and Durable Functions.

As a Proof of Concept was implemented a program for text file word counting using own MapReduce implementation. To launch the function a user would need to upload a text file to the cloud storage, which is also the output destination. Only the orchestrator code is shown here:

```
public static async Task WordCountOrchestrator(
  [OrchestrationTrigger] IDurableOrchestrationContext
  ctx
)
{
  var input = ctx.GetInput<WordCountInput>();
  var batches = Batching.ToBatches(ToLines(input.
  Content));

  var mapResults = await Task.WhenAll(
    batches.Select(
     batch => ctx.CallActivityAsync<IList<
     Result<string, int>>>(
       functionName: nameof(WordCountMap),
       input: batch
     )
    )
  );

  var groups = await ctx.CallActivityAsync<
  IList<Group<string, int>>>(
    functionName: nameof(WordCountGroup),
    input: mapResults
  );

  var reduceResults = await Task.WhenAll(
       groups.Select(
         group => ctx.CallActivityAsync<Result<
         string, int>>(
          functionName:nameof(WordCountReduce),
          input: group
         )
       )
  );

  await ctx.CallActivityAsync<string>(nameo
  f(WordCountOutput), reduceResults);
}
```

Input data is evenly split into batches and they are sent to map functions. In turn, they return (key, value) pairs lists which are merged and grouped by keys. These groups are sent to reducers each of which returns one (key, value) pair. The resulting list of pairs is written into a text file to the storage.

Implementing such architecture with Azure Functions might not be the most optimal solution, as the platform is not optimized for such tasks and there are strict execution time and input data size restrictions. At the same time, this experiment proves that if the platform is capable of executing such algorithms, it's most likely suited for typical, not so computation-intensive, business logic.

Actor model

Some applications need a state to work with and some of them would use multiple threads to do so. Alas, it's hard to work with mutable data in a multi-threaded environment, because it requires using various synchronization mechanisms to keep data consistent and not allow deadlocks and race conditions. Using such mechanisms gave birth to a new wave of problems to solve and one such solution is the actor model. It introduces a new abstraction – actors – entities which react to messages from other entities and can make some actions in response:

- Change their internal state.
- Send messages to other actors.
- Create a new actor.

Actors use queues for message passing, obviating typical lock-based synchronization [2].

Azure Durable Entities are an implementation of this model. They guarantee that messages are delivered reliably and in order and that blocking an entity doesn't block functions sending it signals [6].

A good example is bank accounts simulation. Users can replenish their accounts, get their balance, withdraw money, and make transactions between two accounts:

```
public class Account : IAccount
{
  public decimal Balance{get; set;}= decimal.
  Zero;

  public Task Replenish(decimal amount)
  {
    Balance += amount;
    return Task.CompletedTask;
  }

  public Task<bool> Withdraw(decimal amount)
  {
    var canWithdraw = amount <= Balance;

    if (canWithdraw)
      Balance -= amount;

    return Task.FromResult(canWithdraw);
  }

  public Task<decimal> GetBalance() =>
  Task.FromResult(Balance);

  public static Task Run([EntityTrigger]
  IDurableEntityContext ctx) =>
    ctx.DispatchAsync<Account>();
}
```

Durable Entities obey the same rules as regular actors but with a few simplifications:

- Entity is implicitly created after it receives the first signal.
- Client functions and orchestrators can also message entities.

Function for replenishing an account may look like this:

```
public static async Task<IActionResult>
Replenish(
  [HttpTrigger(methods: "post", Route = "rep-
  lenish")] HttpRequest req,
  [DurableClient] IDurableEntityClient client
)
{
  var data = JsonConvert.DeserializeObject<
  RequestBody?>(
    await new StreamReader(req.Body).ReadTo-
    EndAsync()
  );

  await client.SignalEntityAsync<IAccount>(
    entityKey: data.Account,
    operation: account => account.Replenish
    (data.Amount.Value)
  );

  return new OkObjectResult(new {message = "Suc-
  cessfully replenished"});
}
```

This function uses a simple HTTP trigger from which it gets an entity key to signal an entity. If there is a need to send a message and receive a response an orchestrator must be used, as only they can await a response. Withdrawal function:

```
public static async Task<bool> Withdraw(
   [OrchestrationTrigger] IDurableOrchest-
   rationContext ctx
)
{
   var input = ctx.GetInput<WithdrawArgs>();
   var account = ctx.CreateEntityProxy<IAc-
   count>(input.Account);
   var wasSuccessful = await account.Withdraw
   (input.Amount);
   return wasSuccessful;
}
```

This time the function awaits a response from an entity and returns whether the withdrawal was successful.

For a consistent change in multiple entities, one should use so-called critical sections. They block

the entities but not other functions working with them [6]:

```
public static async Task<bool> Transfer(
   [OrchestrationTrigger] IDurableOrchestration-
   Context ctx
)
{
   var input = ctx.GetInput<TransferArgs>();
   var fromEntity = new EntityId(nameof(Ac-
   count), input.FromAccount);
   var toEntity = new EntityId(nameof(Account),
   input.ToAccount);

   using (await ctx.LockAsync(fromEntity,
   toEntity))
   {
      var fromAccount = ctx.CreateEntityProxy
      <IAccount>(fromEntity);
      var toAccount = ctx.CreateEntityProxy
      <IAccount>(toEntity);

      var hasEnoughFunds = await
      fromAccount.Withdraw(input.Amount);
      if (!hasEnoughFunds)
         return false;

      await toAccount.Replenish(input.Amount);
   }

   return true;
}
```

All messages and blocks end up in queues and will be processed as soon as receiving entities are unblocked. Also, critical sections cannot have deadlocks. This model allows for reliable communications in a distributed concurrent environment while being very transparent for developers.

Conclusion

This work presented an analysis of serverless as an architecture and as a tool for building distributed event-driven applications. As a potent FaaS implementation, Microsoft Azure Functions, and their extension Azure Durable Functions, were also presented and demonstrated. Four different flexible patterns were developed, exemplified, and explained to understand important ideas by example.

Research results can be used as a practical guide to serverless main concepts and usage scenarios. More research should be done on the topic, including building a real serverless application using typical patterns to study the architecture in more depth.

References

1. Amazon Web Services. (n.d.). Create a Serverless Workflow with AWS Step Functions and AWS Lambda. In *Amazon Web Services documentation*. Retrieved from https://aws.amazon.com/getting-started/hands-on/create-a-serverless-workflow-step-functions-lambda.
2. Charles. (2012). Hewitt, Meijer and Szyperski: The Actor Model (everything you wanted to know, but were afraid to ask) [Video]. Channel 9. Retrieved from https://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask.
3. Hammond, J., Mines, C., Livingston, A., & Hartig, K. (2020). The Forrester New Wave™: Function-As-A-Service Platforms, Q1 2020. Forrester. Retrieved from https://reprints.forrester.com/#/assets/2/108/RES155938/reports.
4. Microsoft. (n.d.a). Azure Functions triggers and bindings concepts. In *Microsoft Azure documentation*. Retrieved from https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings.
5. Microsoft. (n.d.b). Durable Functions types and features. In *Microsoft Azure documentation*. Retrieved from https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-types-features-overview.
6. Microsoft. (n.d.c). Entity functions. In *Microsoft Azure documentation*. Retrieved from https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities.
7. Microsoft. (n.d.d). Orchestrator function code constraints. In *Microsoft Azure documentation*. Retrieved from https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-code-constraints.
8. Roberts, M. (2018). Serverless Architectures. Retrieved from MartinFowler.com. https://martinfowler.com/articles/serverless.html.
9. Shilkov, M. (2018). Making Sense of Azure Durable Functions. Retrieved from https://mikhail.io/2018/12/making-sense-of-azure-durable-functions.

Моренець І. Е., Шабінський А. С.

МЕТОДИ І ЗАСОБИ РОЗРОБКИ ПОДІЄ-КЕРОВАНИХ ЗАСТОСУНКІВ НА SERVERLESS АРХІТЕКТУРІ

Новий підхід до розробки застосувань – serverless – підіймає гнучкість розробки та розміщення на новий рівень, значно зменшуючи одиницю розгортання. Однак у цьому напрямі досі не було точно визначено, як і які застосування варто будувати, використовуючи його, чому і присвячено цю роботу. В дослідженні використано Microsoft Azure Functions для демонстрування чотирьох гнучких патернів із прикладами коду, таких як HTTP наносервіси та розподілені обчислення. Результати роботи можуть бути використані як прикладний практичний посібник з основних понять і патернів serverless.

Ключові слова: serverless, Function-as-a-Service, Microsoft Azure, подіє-керовані застосування, хмарні технології.

Матеріал надійшов 08.06.2020