

All Articles

Faux-SQL or NoSQL? Examining four DynamoDB Patterns in Serverless Applications

During my time at [Serverless, Inc.](#), I've talked with a lot of users about their serverless applications. One of the persistent questions that people ask is around which database to use.

Many people reach for DynamoDB as their database of choice, with good reason. It's the most serverless-friendly database, for a number of reasons:

- Accessible via HTTP;
- Fine-grained access controls with IAM;
- [Usage-based pricing model](#);
- [Stream-based activity log for updating secondary data stores](#)

These features make DynamoDB a great fit for use with Lambda or even non-Lambda-based serverless solutions like [Step Functions](#) or [AppSync](#).

That said, I see a few different patterns on modeling data in DynamoDB with serverless applications. In this post, I want to cover the four patterns I see most often:

- The Simple Use Case
- Faux-SQL
- "All-in NoSQL"
- The True Microservice

Let's review them one-by-one, covering the pros and cons of each.

The Simple Use Case

One of the great things about serverless is how easy it is to build and maintain simple services. In a few hours, you can have a [Slack bot](#) or a [GitHub webhook handler](#).

For these simple applications, DynamoDB is a perfect fit. Your data access patterns are pretty limited, so you won't need to go deep on learning DynamoDB. You can handle all of your needs with a single table, often without the use of [secondary indexes](#).

I find these simple use cases to be one of the “gateway drugs” of serverless usage. These apps can be free to run with the [AWS Free Tier](#). They're often internal tools or fun toys, so you don't need to go deep on Lambda internals for optimizing performance or avoiding cold-starts.

While these simple use cases are a great way to learn and provide quick value, it's the production workloads that are most interesting. Let's move on to the next most common pattern I see.

Faux-SQL

In production, you have more complex applications than the simple use case. You have multiple, interrelated data models. For serverless users, it can be a hard choice between:

1. **Traditional SQL database:** A data model they know and understand, but a scaling and interface model that doesn't fit within the serverless paradigm.
2. **DynamoDB:** A HTTP-based service that scales well but has an unfamiliar data model.

Some people decide to choose parts of both — use multiple DynamoDB tables and normalize their data. For example, if you were building a blog, you might have two DynamoDB tables — `posts` and `comments`. The `comments` table would use a `postId` as the hash key of its primary key to tie the comments to a specific table.

This pattern is not recommended by the DynamoDB team. They have some great [best practices on modeling relational data](#), but they advise strongly against normalizing your data in a faux-SQL approach.

That said, I see this pattern recommended a lot by the [AWS AppSync](#) team. AppSync is a hosted GraphQL service from AWS, and [many of the tutorials](#) use a separate DynamoDB table per entity.

Pros of the Faux-SQL approach:

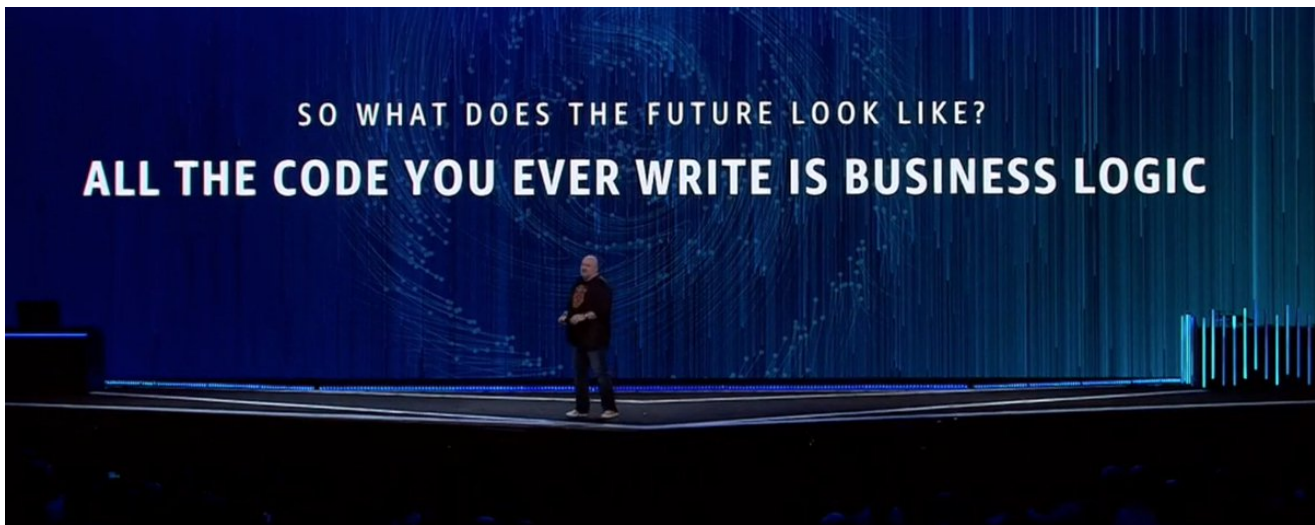
1. **It's familiar to developers.** They know how to normalize data. They don't have to learn a complex new data modeling system.
2. **It's easier to put into an external service.** Often you will replicate DynamoDB data into a secondary datastore, such as Elasticsearch for searching or Redshift for analytics. Since your data is already normalized, they can be streamed directly into those sources without much modification.
3. **It's easier to add new query patterns in the future.** It's not as flexible as a traditional SQL database in this regard as you're still limited by DynamoDB's query patterns, but a normalized structure allows for additional flexibility in the future.

Cons of the Faux-SQL approach

1. **You are making multiple queries for your calls.** You should aim to satisfy your needs with a single query.
2. **Your application code just became a database.** Rather than doing a JOIN in SQL, you are doing it in code. This means you are responsible for maintaining referential integrity and for optimizing your queries.

You know what's really good at maintaining referential integrity and optimizing JOINS? A relational database.

If the promise of serverless is that the only code you write is your business logic, why are you replicating half of an RDBMS in your application code?



If you want a flexible, normalized data system, use a traditional RDBMS + SQL. If you want to use DynamoDB, lean in to the NoSQL mindset.

This brings us to our next pattern: going “all-in” on NoSQL.

“All-in NoSQL”

The third DynamoDB pattern I see is the “All-in NoSQL” model. In this pattern, people take the time to learn and understand NoSQL generally and DynamoDB specifically.

They don’t fight NoSQL. They embrace it.

The best way to get up to speed on this is to watch the [Advanced Design Patterns for DynamoDB talk from re:Invent 2018](#) by database rockstar Rick Houlihan. Based on the Twitter chatter I saw this year, he was the breakout favorite from re:Invent.



Josh Barratt
@jbarratt





Feel completely schooled (in the best way) by Rick Houlihan's talk about DynamoDB, but also general NoSQL data modeling practices from [#reinvent2018](#). Already online! [youtube.com/watch?v=HaEPXo...](https://www.youtube.com/watch?v=HaEPXo...) Highly recommended if you're open to

AWS re:Invent 2018: Amazon DynamoDB Deep Dive: Adv...
This session is for those who already have some familiarity
with DynamoDB. The patterns and data models discussed

with DynamoDB. The patterns and data models discusse...

 youtube.com

9:53 PM · Nov 28, 2018

 31  See Josh Barratt's other Tweets

One of the more important points Rick makes is that “NoSQL is not flexible. It’s efficient but not flexible.”

“NoSQL is not flexible. It’s efficient but not flexible.” – Rick Houlihan

NoSQL databases are designed to scale infinitely with no performance degradation. They’re able to be efficient because they are perfectly tailored for your queries.

Unlike a SQL database, you won’t design your tables first and then see what questions you need to answer. You map out your query patterns up front, then design your DynamoDB table around it.

You will need to learn how DynamoDB works.

You will need to overload your secondary indexes.

It will be extra work upfront.

If your application needs to scale, **it will be worth it.**

Pros of the “All-in” NoSQL approach

1. **Your queries will be efficient.** You should be able to satisfy any access pattern with a single query. Your database won’t be your application bottleneck.
2. **You are built for scale.** Your application should scale infinitely without any performance degradation. You won’t need to rearchitect because of data access concerns.

Cons of the “All-in” NoSQL approach

1. **It’s unfamiliar to your developers.** Most developers have a good understanding of modeling for SQL databases. Modeling for NoSQL is a different world, and it can take some time to get

up to speed.

2. **It's more work upfront.** You have to do the hard work up front to really think about your data access patterns and how you will satisfy them.
3. **It's less flexible.** If you want to add new access patterns in the future, it may be impossible without a significant migration. You should account for this possibility.
4. **It's harder to replicate into a secondary source.** If you want to stream your DynamoDB data into something like Elasticsearch for search or to Redshift for analytics, you will need to do heavy transformations to normalize it before sending it to the downstream systems.

When done right, the “All-in NoSQL” approach can be wonderful. The key is taking the time to do it right.

Finally, let's take a look at a fourth pattern — the True Microservice.

The True Microservice

The previous two patterns — faux-SQL and “All-in NoSQL” — are strategies for when you have a complex data model with multiple, interrelated entities. In this pattern, the True Microservice, we will see how to get all of the benefits of DynamoDB with none of the downsides.

[Microservices](#) are one of the bigger trends in software development in recent years. Many people are using serverless applications to build a microservice architecture.

When building microservices, you should aim that [each microservice aligns with a bounded context](#). Further, you should avoid synchronous communication with other services as much as possible. This is particular true in Lambda where, as Paul Johnston says, [“functions don't call other functions”](#). The added latency and dependency issues will hurt the benefits you want from microservices.

My favorite use case for Lambda + DynamoDB-based microservices is basically implementing the principles from Martin Kleppmann's talk, [Turning the Database Inside Out](#). In it, Martin describes a new way to think about data in an event-centric world. Rather than

using the same, single datastore for all of your reading + writing needs, you use purpose-built views on your data that are tailored to a particular use case.

Let's see how this might fit in a serverless architecture. Your microservice is likely getting its data asynchronously from an upstream source. This could be from an [SQS queue](#), a [Kinesis stream](#), or even the [stream of a DynamoDB table](#) from a different service. The upstream source will trigger a writer function that saves the data to DynamoDB.

All web requests to your service are read-only requests. This service is handling one or maybe two data entities, so your datastore is perfectly tailored to serve these read requests.

Pros of the True Microservice

1. **Simple, straight-forward data modeling.** Because you are working with a limited domain, you don't need to get into advanced data modeling work with DynamoDB.
2. **Fast, scalable queries.** You get all the benefits of DynamoDB's low response times with infinite scaling.
3. **Easy to stream into secondary sources.** The data model is straight-forward, so you don't need to do as much work to normalize when sending to Elasticsearch or Redshift.

Cons of the True Microservice

1. **Not always possible.** It may be difficult to whittle each portion of your application into a small, contained microservice. As Rick Houlihan says in his talk, "All data is relational. If it wasn't, we wouldn't care about it." For the core of your application, you may have multiple related entities that you cannot break into independent services without relying on expensive service-to-service calls.
2. **Eventual consistency issues.** If you are populating your data asynchronously from upstream sources, you can run into consistency issues. This can be confusing to users of your application. Not all True Microservices use the asynchronous writer pattern, but many do.

The True Microservice is the purest serverless application, and I highly recommend using it when you can. The logic is straight-forward due to the limited domain. Scaling is easy and automatic, and it's easy to see how much the service is costing you on your bill.

Conclusion

DynamoDB is a great database and a perfect fit for many serverless applications. In this post, we covered the four main patterns I see with DynamoDB and their pros and cons.

I'd love to hear about how you are using DynamoDB or about which other questions you have on DynamoDB. Hit me up [on Twitter](#) or [via email](#) if you want to chat!

Published 19 Dec 2018

[AWS](#)[Cloud](#)[DynamoDB](#)

AWS Data Hero providing training and consulting with expertise in DynamoDB, serverless applications, and cloud-native technology.

[Alex DeBrie on Twitter](#)

ALSO ON ALEXDEBRIE

My DynamoDB Wish List

a year ago • 13 comments

In this post, I talk about my #awswishlist for DynamoDB, including ...

Three ways to use AWS services from a ...

a year ago • 5 comments

By default, Lambda functions in a VPC cannot access the public ...

DynamoDB Transactions: Use ...

10 months ago • 18 comments

DynamoDB Transactions are powerful ways to operate on multiple items in a single ...

Us Re

2 ye

This cus wal

23 Comments

[alexdebrie](#)

[Disqus' Privacy Policy](#)

[Login](#) ▾

Recommend 12

Tweet

Share

Sort by Best ▾



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS

Zack_Leman • 2 years ago

Great article. I'm interested in how the performance between the same docker image running on fargate vs a c5.large EC2 instance. My initial tests (a few months ago before they started using firecracker for fargate) have showed at least a 200% performance increase. I also noticed fargate performance really depends on the processor AWS allocates to the fargate task. And the processor is not always the same. Would be worth checking if there are significant differences between fargate with different underlying CPU types.

7 ^ | v • Reply • Share ›

Alex D Mod → Zack_Leman • 2 years ago

Thanks, @@Zack_Leman !

Great question, and I'd love to dive deeper into specific performance testing in the container space. There are a lot more knobs to tune there.

When you saw a 200% performance increase, was this with the container already up and running on Fargate? Or did that include startup time?

Fargate startup time wasn't the best, but I waited until all the instances were ready before starting testing.

^ | v • Reply • Share ›

Jack Yu • 2 years ago • edited

Nice article!

I have a question about making checksum.

According to your article, the authorizer of request type can't get the request body.

Is it right?

If I want to make a checksum with request body, I need to make it in my backing lambda function (in my API Gateway endpoint).

Is my thought correct?

1 ^ | v • Reply • Share ›

Felix Bouaket Chanthapanya → Jack Yu • a year ago

correct. just log the authorizer's event to confirm.

^ | v • Reply • Share ›

Sarfaraz Ghulam • 4 months ago • edited

Great article. Is it possible to create service proxy integration for an SQS? I know to do it from console but I am struggling to create a SAM template to achieve it.

^ | v • Reply • Share ›

nistozia • 5 months ago

pistazie • 5 months ago

Thanks for this tutorial, I used it to get a RestApi integration with SNS done in AWS CDK.

I packed this into an AWS CDK library: <https://github.com/pistazie...>

^ | v • Reply • Share ›

Steve • 7 months ago

Excellent and very helpful read. Thank you.

Regarding the context object in the response, and inserting arbitrary data into it...

I've previously used the built in API Gateway Cognito authorizer. This worked nicely, as it added the JWT token claims to the event payload that my lambda received. Essentially:

```
'requestContext': {  
  'authorizer': {  
    'claims': {  
      'sub': '...',  
      'otherthing': '...'
```

However now with my custom authorizer I want to try return the same structure. But if I try to add an object into this context, the API Gateway explodes with:

```
Execution failed due to configuration error: Invalid JSON in response: Can not  
deserialize instance of java.lang.String out of START_OBJECT token
```

I've resorted to simply adding the claims as below, but I prefer the other structure:

```
'requestContext': {  
  'authorizer': {  
    'claim.sub': '...',  
    'claim.otherthing': '...'
```

^ | v • Reply • Share ›

Juan • 9 months ago

Do you know if “JWT Authorizers” of the new **HTTP APIs** (from API Gateway) can be an equivalent solution?

^ | v • Reply • Share ›

Alex D Mod ➔ Juan • 9 months ago • edited

Hey @Juan , good question. I haven't done much with the new HTTP APIs, but check out Jeremy Daly's post on [using JWTs with the new HTTP APIs](#).

2 ^ | v • Reply • Share ›

This comment was marked as spam.

EsteVap ➔ EsteVap • 10 months ago

I found, because I was using CLI AWS from windows, some adaptation with specific caractere was required to adapt settings about http header value and the

arn format in the template values.

^ | v · Reply · Share ›

Thang Nguyenchien · 10 months ago · edited

Great article. Can I run codesearch with local resources ?

If my resources change. Do I rebuild file livegrep.idx ?

^ | v · Reply · Share ›

Sebastian · a year ago

Hi Alex, thanks for your articles, they are very helpful.

I am trying to reuse an existing module hosted on premises that is based in IdentityServer4 to authenticate to it. The users/passwords should stay in the IdentityServer module, not in AWS.

For this I am trying to add our IdentityServer module as a provider to Cognito.

After doing this I would expect that to get a response from our API Gateway resource I would have to somehow send a valid token in the request.

With our current setup I am getting 401 Unauthorized both when testing the authorizer from the API Gateway and when doing a GET request from Postman.

I attached some screenshots of the current setup to this question I posted in the AWS forum:

<https://forums.aws.amazon.c...>

Thanks!

^ | v · Reply · Share ›

Vicenç García-Altés · a year ago

Hi Alex,

Great post. Just one question, if I return an error from my custom authorizer, I got a 500 in the client with no message. What should I do to return a 403 with an "non authorized" message?

Thanks!

^ | v · Reply · Share ›

Vicenç García-Altés ➔ Vicenç García-Altés · a year ago · edited

Oh, I got it (Thanks to <https://stackoverflow.com/a....>

If you want to return a 401 you need to do

`context.fail('Unauthorized')`

If you want to return a 403, you need to return the same policy that you return when you want to allow access but using Deny instead of Allow

And instead of just returning the response, you can do `context.success({the policy})`

^ | v • Reply • Share ›

Dilantha Prasanjith • a year ago

Great article thank you !!!!

^ | v • Reply • Share ›

Arved Sandstrom • a year ago

While this article in a way is not about AWS SNS and SMS per se (and nice article by the way), I thought I'd point out that the process described is easily modified to send a text message to a phone number directly, not using a topic or a subscription at all. Leave out the bits about the topic and the subscription, the permission on `Sns.Publish` is not restricted to a topic resource, the integration uses `PhoneNumber` instead of `TopicArn`, and you'll probably want to have the phone number value be dynamic in your mapping template as well.

^ | v • Reply • Share ›

BoneSpike • a year ago

You mentioned whether the * approach can be a security concern - yes it can - if your cached authorizer approved you to view a record, and another function allows write (or even administrative) functionality - they've gotten through. Possible solutions could be:

- * setting cache to 0 as you mention

- * separate authorizers for different access levels - guard them by 'lowest level' that allows access (that does tie a bit more business logic back into the authorizer layer)

- * use *'s as _part_ of the arn - if you follow a specific naming convention - you can possibly use that (feels really hack-ish to me)

- * use custom authorizers as application-specific identity mappers instead of really authorization - take the base authentication and look up what that means for the app - this google user is a member of app group x, y, z. Shove that in context, and let the underlying app make decisions based on the user's role where possible. You still may have some 'per-record' decisions to make - the authorizer is there for who can 'try' to do it...

Just some thoughts....

^ | v • Reply • Share ›

Simon Ouellet • a year ago

100/100

Thank you !

^ | v • Reply • Share ›

Abd Elhamied Raslan • 2 years ago

This is really a great article which explains every single step, but I have a question since I am a beginner. When I was trying to create the SNS topic I couldn't use the command

that you wrote which is:

```
TOPIC_ARN=$(aws sns create-topic \  
--name service-proxy-topic \  
--output text \  
--query 'TopicArn')
```

The error was that TOPIC_ARN is an unknown command. But it runs without errors when I run the code like this:

```
aws sns create-topic --name service-proxy-topic --output text --query 'TopicArn'
```

But now I don't have the ARN saved in the variable TOPIC_ARN. So do you have any idea about how to solve such problem?

Thank you very much

^ | v · Reply · Share ›

Jonathon Belotti · 2 years ago

Great article. Not having to figure this all out from scratch probably saved me at least 2 hours.

^ | v · Reply · Share ›

John Lim · 2 years ago

In the APIG -> SNS example, how do you control access to your API i.e. without lambda authorizers?

^ | v · Reply · Share ›

Alex D Mod ➔ **John Lim** · 2 years ago

Hey **@John Lim**, great question.

You'll need to use [one of the methods for controlling access to API Gateway](#).

The most common are:

- * Lambda authorizers (as you mentioned)
- * Resource policies (to allow certain IP address ranges or VPCs)
- * Cognito user pools
- * API Gateway API keys

^ | v · Reply · Share ›

John Lim ➔ **Alex D** · 2 years ago

Thanks Alex!

^ | v · Reply · Share ›