

Integrating Amazon ElastiCache with other AWS services: The serverless way

by Nir Mashkowski | on 11 NOV 2020 | in [Amazon DynamoDB](#), [Amazon ElastiCache](#), [AWS Lambda](#) | [Permalink](#) | [Comments](#) | [Share](#)

Amazon ElastiCache for Redis is a great way to accelerate cloud applications through caching and other use cases such as session storage, leaderboards, and messaging. Developers use Redis to store application data using in-memory data structures like hashes, lists, and sorted sets. This data can be useful downstream for other purposes such as reporting and data analysis. Because Redis stores data in-memory to provide low latency and high throughput, long-term data storage is impractical due to the high cost of RAM. Therefore, it's ideal to transfer data from Redis to other AWS purpose-built databases such as [Amazon DynamoDB](#) or [Amazon Aurora MySQL](#).

In this post, I describe a simple way to transfer data from ElastiCache for Redis clusters to other AWS database services using [AWS Lambda](#) functions.

Use case

Developers use the [hash](#) data type to represent objects or *property bags* that can be mapped to a DynamoDB or relational table row, such as a website shopping cart. For our use case, we use Redis to store customer selections, helping provide a fast and interactive experience. When the customer decides to make a purchase, the selected items are sent to the billing system for processing, where they are persisted. One way to enable business analysis is to have a record of the items the customer saved in their cart during their time on the site. You can later use the data to analyze customer purchasing patterns and product selection trends.

Solution requirements

Redis has a rich API that enables efficient reading and writing of data. For example, adding a new field to a Redis hash is as easy as providing a new field name and value with the [HSET](#) command. Redis also allows for diverse data structures (strings, hashes, sets, lists), which you can map to different entities in other databases. Therefore, our solution must be:

- **Flexible** – Because the incoming data can vary so much, we need to call different backends for different data types. For example, we may want to map the data from Redis hashes that include user information to one table and their product preferences to another.
- **Scalable** – We use Redis to make sure our customer-facing application is highly responsive and can handle fluctuations in traffic. Our solution needs to handle changes in traffic without creating lags or impacting the responsiveness of your Redis-powered front end.
- **Reliable** – Uptime is critical for retaining the responsive user experience we're looking for. Our solution needs to allow for redundancy and limit the "blast radius" in cases of failure.

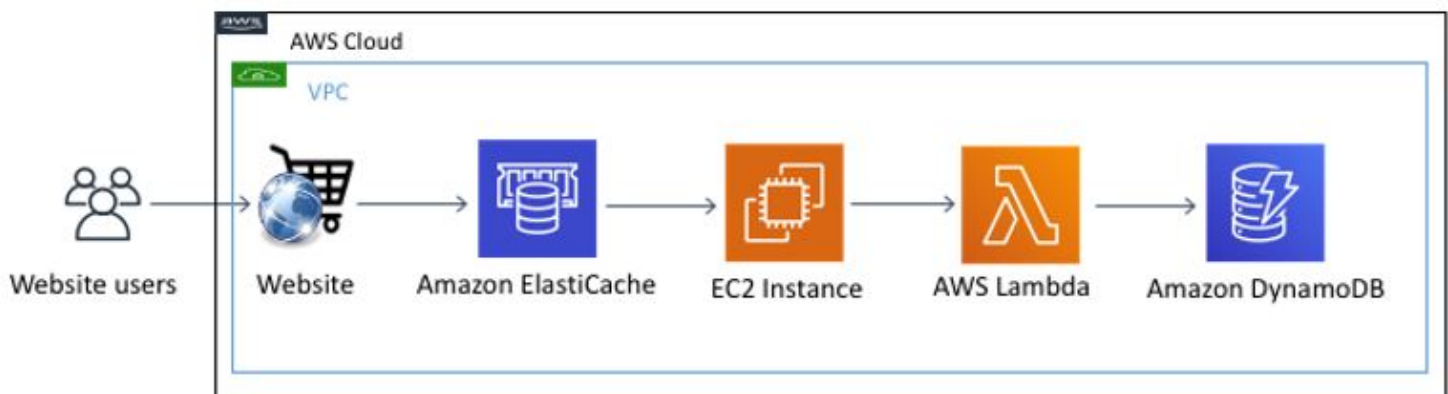
Solution overview

We use several technologies to address our requirements. To keep our solution flexible, we use Lambda functions. You can deploy Lambda functions quickly and easily integrate them with other AWS solutions. To make our data feed scalable, we use Redis [keyspace notifications](#). Keyspace notifications are provided through pub-sub events and can be configured to only emit a subset of commands (for example, only hash-based operations) as well as a subset of keys. With the pub-sub architecture, it's easy to add nodes that act as subscribers for Redis data changes and scale out the solution so we can handle more incoming traffic.

Redis pub-sub messages are generated in real time and aren't persisted in Redis. This means messages are lost unless an active client is listening on the right [channel](#). To make a more fault-tolerant version of this solution, you can write data to [Redis streams](#) and consume them in a similar manner to pub-sub. Because Lambda doesn't support the Redis pub-sub trigger type, we use a simple executable hosted on an [Amazon Elastic Compute Cloud](#) (Amazon EC2) instance to subscribe to the Redis keyspace notification events and invoke Lambda functions. We don't cover it in this post, but it's easy to containerize the subscriber piece and have it scale out with services like [Amazon Elastic Container Service](#) (Amazon ECS).

Solution architecture

For our use case, we use two Lambda functions to push data to DynamoDB. A Golang application hosted on an EC2 instance is used as a listener for Redis data changes and to invoke the Lambda functions. The following diagram illustrates the solution data flow and components.



Solution code

The solution contains surprisingly few lines of code, which we discuss further in this section.

Redis listener

The source code for this component is located on the [Redis Listener](#) GitHub repo. It's a Golang executable that uses [Redis pub-sub](#) to subscribe to keyspace notifications. We do that by connecting to Redis and subscribing to the appropriate channel.

The `PubSubListen()` function is called after some initiation work when the Redis listener executable is loaded. See the following code:

```
func PubSubListen() {

    //Connect to Redis
    rdb := getRedisClient()
    //Register to listen to the keyspace notification channel
    pubsub := rdb.PSubscribe(ctx, os.Getenv("REDIS_SUB_CHANNEL"))
    //Error handling
    _, err := pubsub.Receive(ctx)
    if err != nil {
        panic(err)
    }
    //Declare the channel variable
    ch := pubsub.Channel()
    //Iterate over messages in the channel
    for msg := range ch {
        fmt.Println(msg.Channel, msg.Payload)
        //Check whether the key name in the message is a map of Redis data to Lambda
        if strings.HasPrefix(msg.Payload, os.Getenv("META_MAP_SUFFIX")) {
```

`LambdaInvoke` can be called more than one time for the same Redis hash if you want to send data to more than one destination. The data structure used as an interface between the Redis listener component and the Lambda functions is a very simple JSON object. This `iEvent` type is shared between the listener component and the Lambda function:

```
type iEvent struct {
    Id      string `json:"id"`
    Obj_Name string `json:"obj_name"`
    Body    string `json:"body"`
}
```

DynamoDB Lambda function

The Lambda function code is also very simple; it takes the `iEvent` object and sends it to DynamoDB. Because DynamoDB requires a unique ID for each row, we use the name of the hash object from Redis to make sure records are added and updated. Delete is not supported but can be added later on. See the following code:

```
input := &dynamodb.PutItemInput{
    Item:      av,
    TableName: aws.String(os.Getenv("DYNAMO_TABLE_NAME")),
}

_, err = svc.PutItem(input)

if err != nil {
    fmt.Println("Got error calling PutItem:")
    fmt.Println(err.Error())
    os.Exit(1)
}
fmt.Println("saved to Dynamo " + msg.Id)
```

Setting up the solution components

Our solution includes four different components, as well as security and networking configuration. We use an [AWS Serverless Application Model](#) (AWS SAM) template to put it all together.

The template included with this post deploys the DynamoDB table, Lambda function, EC2 instance, and the configuration needed to connect them (VPC, subnets, security group, internet gateway, and route table to allow access to the EC2 instance). When the setup process is complete, you set up the Redis listener component on the EC2 instance.

Prerequisites

AWS SAM makes it easy to create and deploy serverless resources. To get started, you need the following prerequisites on your dev machine:

- The [AWS Command Line Interface](#) (AWS CLI) already configured with administrator permission. For instructions, see [Installing, updating, and uninstalling the AWS CLI version 2](#).
- Access to your [AWS access key and secret](#).
- [Docker installed](#).
- [Golang](#)
- The [AWS SAM CLI installed](#).
- [Git](#)
- Create or make sure you have access to an Amazon EC2 [key pair](#). You use it to connect to your EC2 instance. For instructions on obtaining a key pair, see [Creating or importing a key pair](#).

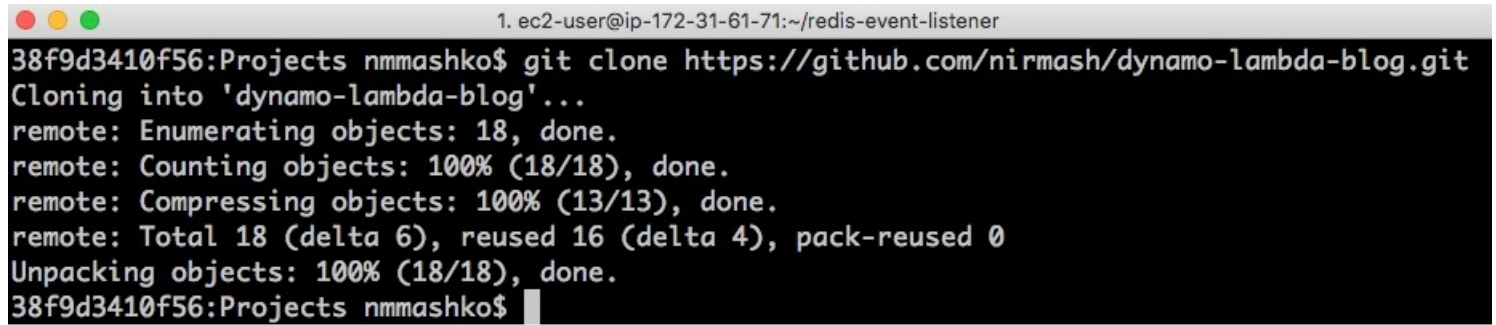
Deploying your template

When everything is installed, open a terminal window to deploy your AWS SAM template.

1. Enter the following code:

```
git clone https://github.com/nirmash/dynamo-lambda-blog.git
```

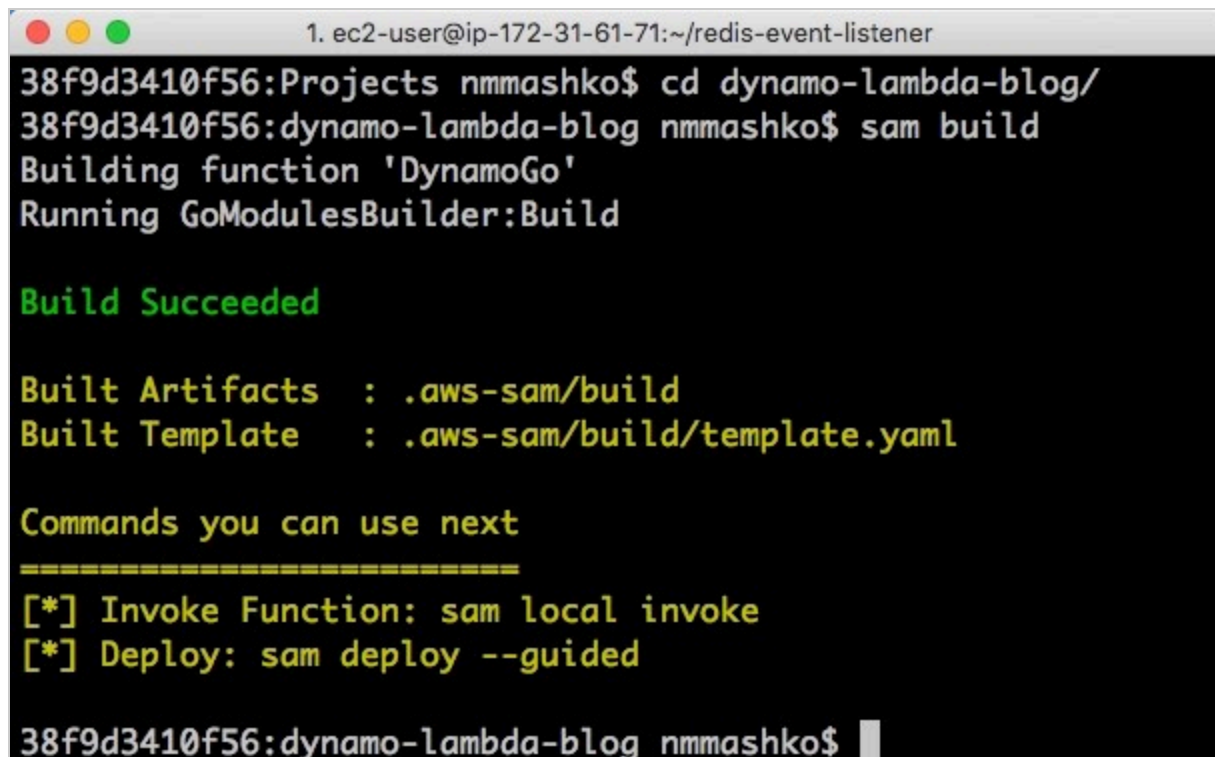
The following screenshot shows your output.



```
1. ec2-user@ip-172-31-61-71:~/redis-event-listener
38f9d3410f56:Projects nmmashko$ git clone https://github.com/nirmash/dynamo-lambda-blog.git
Cloning into 'dynamo-lambda-blog'...
remote: Enumerating objects: 18, done.
remote: Counting objects: 100% (18/18), done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 18 (delta 6), reused 16 (delta 4), pack-reused 0
Unpacking objects: 100% (18/18), done.
38f9d3410f56:Projects nmmashko$
```

2. CD into the new Git folder.
3. Enter `SAM build`.

The following screenshot shows your output.



```
1. ec2-user@ip-172-31-61-71:~/redis-event-listener
38f9d3410f56:Projects nmmashko$ cd dynamo-lambda-blog/
38f9d3410f56:dynamo-lambda-blog nmmashko$ sam build
Building function 'DynamoGo'
Running GoModulesBuilder:Build

Build Succeeded

Built Artifacts   : .aws-sam/build
Built Template    : .aws-sam/build/template.yaml

Commands you can use next
=====
[*] Invoke Function: sam local invoke
[*] Deploy: sam deploy --guided

38f9d3410f56:dynamo-lambda-blog nmmashko$
```

Image: build the SAM template

Now that the template is built, it's time to deploy.

4. Enter `SAM deploy --guided`.
5. Follow the prompts on the screen, using the key pair name you obtained earlier.
6. Write down the stack name (first parameter in AWS SAM deploy process) to use later.

The following screenshot shows your output.

Configuring SAM deploy

```
Looking for samconfig.toml : Not found
```

```
Setting default arguments for 'sam deploy'
```

```
Stack Name [sam-app]: DynamRedisBlog-1
```

```
AWS Region [us-east-1]: us-west-2
```

```
Parameter ProjectName [Redis-Listener-Sample]: dynamo-redis-blog-1
```

```
Parameter LatestAmiId [/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2]:
```

```
Parameter ElastiCacheInstanceClass [cache.t3.small]:
```

```
Parameter EC2KeyName [nmmashko-kp-two]:
```

```
#Shows you resources changes to be deployed and require a 'Y' to initiate deploy
```

```
Confirm changes before deploy [y/N]:
```

7. When the setup process is complete, note the values of the ElastiCache cluster endpoint, ElastiCache instance DNS, and the Lambda function ARN.

You need the name of the DynamoDB table to see the data coming in from your Redis cluster. The following screenshot shows your output.

```

1. ec2-user@ip-10-0-4-234:~/redis-listener-blog
ec2-user@ip-10-0-4-234:~/redis-listener-blog (bash)

CloudFormation outputs from deployed stack
-----
Outputs
-----
Key          RedisCluster
Description  Redis Cluster Primary Endpoint
Value        redis-listener-sample-cluster.0001.usw2.cache.amazonaws.com

Key          EC2Instance
Description  ElastiCache Instance URL
Value        ec2-1-1-1-1.us-west-2.compute.amazonaws.com

Key          DynamoGo
Description  ARN of Lambda Function
Value        arn:aws:lambda:us-west-2:1-1-1-1:function:DynamRedisBlog-1-DynamoGo-1AQYZMVEFQTV9

Key          RedisDataTable
Description  Name of the DynamoDB destination table
Value        DynamRedisBlog-1-RedisDataTable-1DDHHCCQ072M0
-----

Successfully created/updated stack - DynamRedisBlog-1 in us-west-2

```

Image: SAM template output

8. When your instance is launched, [SSH into it](#).
9. To set up the listener service, we need to install a few components on the new EC2 instance:
 - **Git** – In the SSH window, enter `sudo yum install -y git`.
 - **Golang** – In the SSH window, enter `sudo yum install -y.`
10. Clone the code repository by entering

```
git clone https://github.com/nirmash/redis-listener-blog.
```

Now you update the service configuration files to point to the Redis cluster and Lambda function.

11. For the environment variables, use an editor to edit the env.sh file in the `redis-listener-blog` folder. Add the following values per instructions:
 1. **REDIS_MASTER_HOST** – Your Redis cluster primary endpoint
 2. **AWS_ACCESS_KEY_ID** – Your AWS access key ID
 3. **AWS_SECRET_ACCESS_KEY** – Your AWS secret access key
 4. **AWS_DEFAULT_REGION** – The AWS Region where your Lambda function is installed.


```
1. ec2-user@ip-172-31-2-129:~/redis-listener-blog
#!/bin/sh
export REDIS_MASTER_HOST=redis1.172.31.2.129.us-west-2.elb.amazonaws.com
export REDIS_MASTER_PORT=6379
export AWS_ACCESS_KEY_ID=AKIAI4478WF4DDA4GK3Q2
export AWS_SECRET_ACCESS_KEY=ng3k02p...XK522
export AWS_DEFAULT_REGION=us-west-2
export REDIS_SUB_CHANNEL=__key*__:*
export META_MAP_SUFFIX=__meta_record__
export SUPPORTED_COMMANDS=hset
```

For function mapping, the `functionCfg` file contains the mapping between Redis keys and the Lambda function. The data from the file is read into a hash key and used every time a new hash field is added or changed to call a Lambda function.

12. Replace `{lambda_arn}` with the ARN of the Lambda function you created.

```
Default (vim)
__meta_record__Dynamo|name, __meta_record__Dynamo, pattern, *, lambda, {lambda_arn}
```

13. Apply the new environment variables from the `source env.sh` file by entering `source env.sh` in the terminal window.

Building and running the service

Now it's time to build and run the service.

1. Enter `go run main.go` in the terminal.

The following screenshot shows your output.


```
1. ec2-user@ip-172-31-2-129:~/redis-listener-blog
ec2-user@ip-172-31-2-129:~/redis-listener-blog (ssh)
[ec2-user@ip-172-31-2-129 redis-listener-blog]$ go run main.go
Enter init...
Load map...
Listen...
```

You now add data to Redis and see it copied to DynamoDB.

2. [SSH into your EC2 instance](#) again from a new terminal window.
3. Enter `cd redis-listener-blog`.
4. Connect to Redis by entering `./redis-cli -h <ElastiCache_primary_endpoint_address>`. Provide the Redis primary endpoint you saved earlier.

You should now be connected to Redis.

5. Add some hash records to Redis by entering the following code:

```
` hset person1 firstName John lastName Smith`
` hset person2 firstName Jane lastName Doe`
```

The following screenshot shows the output.

```
ecdemo.vbaj3gmg.0001.usw2.cache.amazonaws.com:6379> hset person1 firstName John lastName Smith
(integer) 2
ecdemo.vbaj3gmg.0001.usw2.cache.amazonaws.com:6379> hset person2 firstName Jane lastName Doe
(integer) 2
ecdemo.vbaj3gmg.0001.usw2.cache.amazonaws.com:6379> █
```

6. On the DynamoDB console, select the table that starts with `dynamo-go-RedisData`.

On the **Items** tab, you can see the hash records data from Redis.

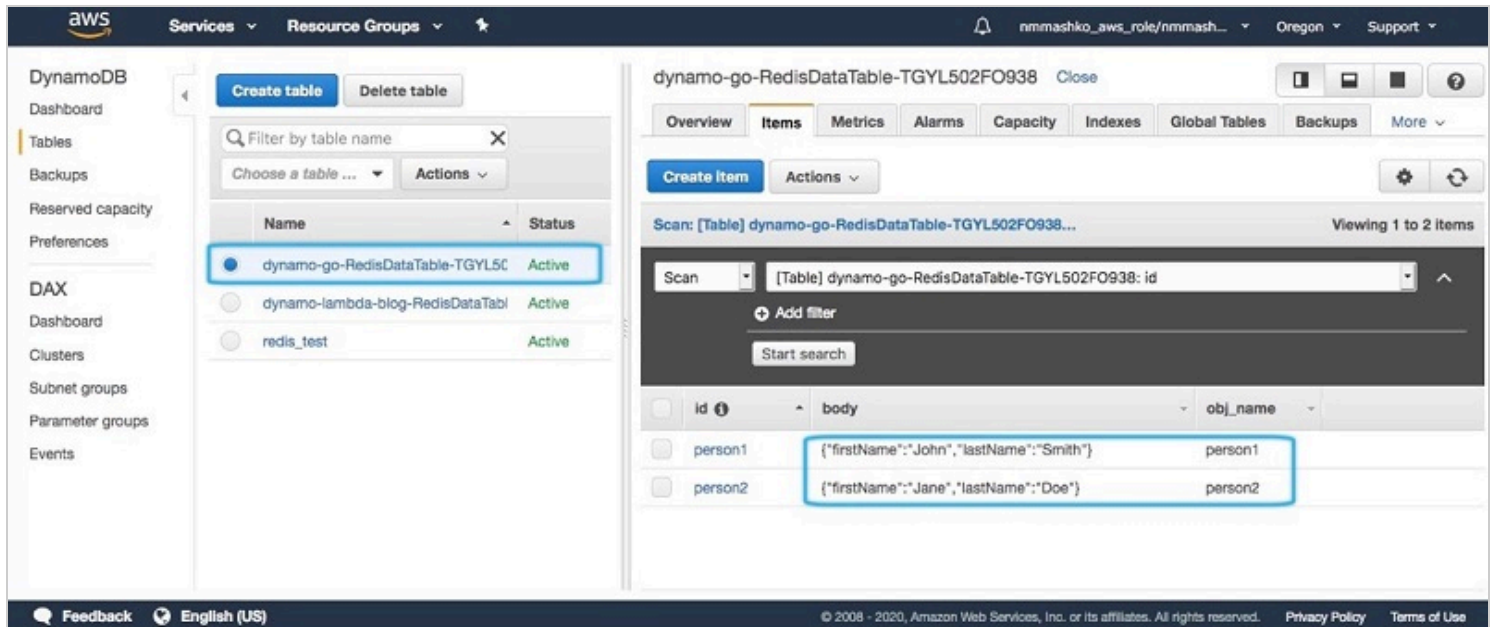


Image: Redis data in DynamoDB

Cleaning up

You now use the [AWS CloudFormation](#) CLI to remove the stack:

```
'aws cloudformation delete-stack --stack-name <Stack Name you entered earlier>
```

This call is asynchronous, so don't be alarmed when it completes immediately. You can check the AWS CloudFormation console for the stack deletion status.

Summary

You can unlock a lot of business value by storing your ElastiCache for Redis data in another AWS data store for long-term analysis or reporting. With AWS, it's easy to put together a flexible, reliable, and scalable solution that you can extend later to support more destinations for Redis data and more Redis data types.

About the author



Nir Mashkowski is a Principal Product Manager in AWS. Nir has been building software products for the last 25 years and is feeling fortunate to be involved in cloud computing for the last 10.

TAGS: [Amazon ElastiCache](#)

Comments

0 Comments

 4 Prashanth ▼



Start the discussion...



Share

Best Newest Oldest

Be the first to comment.

Subscribe

Privacy

Do Not Sell My Data