Caching and real-time notifications in a fully serverless AWS based web application with long-running workflows

9 July 2021 - 7 min. read

AWS Lambda | Serverless

Serverless infrastructures provide huge advantages with respect to “classic” server infrastructures. One can easily understand this by looking at a basic AWS-based Serverless web application developed using AWS Lambda as the backend tier, DynamoDB on-demand (database), Cognito for Authentication, and S3-CloudFront for the frontend. An application like this will scale automatically, in real-time, to accept any amount of traffic and will cost significantly less than an EC2 hosted counterpart while being much simpler to deploy and maintain. Going serverless is thus a no-brainer in most situations: you’ll pay less for an application that will be able to scale better, will have increased isolation, security, and a much lower maintenance effort.

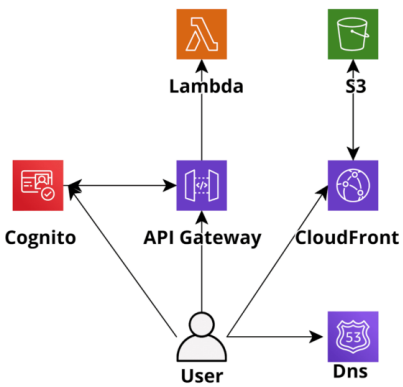

Figure 1: Basic Serverless application example

Nowadays several AWS Lambda-centered development frameworks have been created and they make the development of serverless applications on AWS a breeze (e.g. Chalice for Python, Serverless framework for several languages). Furthermore, they come with batteries included so it is pretty straightforward to create CI/CD pipelines using AWS native DevOps tools (CodeBuild, CodeDeploy, CodePipeline) and to integrate the serverless applications with several other AWS services such as SQS (Lambda based queue consumer architecture), SNS and Kinesis.

However, moving to a serverless architecture involves new problems and solutions with respect to more traditional architectures. Caching of both internal and external resources (e.g. database queries, frontend assets, and third-party APIs) in particular requires different tools and techniques from the ones used in more traditional architecture, and so does the management of long-running processes.

Caching

In a traditional application (e.g. Ruby on Rails or Django) cache is managed with a local or centralized redis/memcached deployment which is often used for everything, from frontend components to DB queries and external API calls responses.

In a serverless environment caching needs to be not only very fast to access (single-digit ms) and straightforward to use but also connectionless and infinitely scalable. These requirements make DynamoDB the most obvious, and usually the better, choice: when configuring a table as on-demand its write and read capacity scales automatically allowing the application to remain responsive even in case of a sudden steep traffic burst and the table access time from Lambda is single-digit millisecond. If an even lower access time is needed it is possible to activate Dynamo DAX Accelerator which can lower the read latency from milliseconds to microseconds!

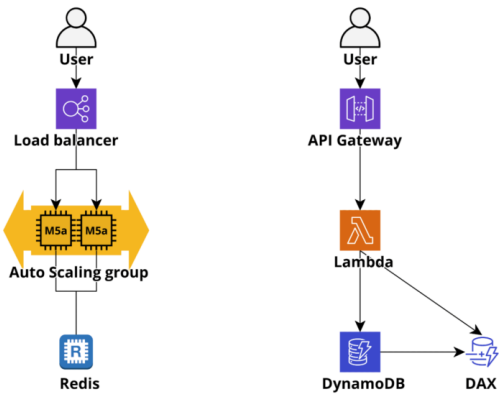

Figure 2: Caching in AWS: EC2 based app on the left, serverless on the right

Long-Running Jobs in a serverless web Application

In a traditional architecture, a web application would usually manage long-running tasks by spawning threads and sending notifications via open WebSocket connections. On the other end, a Lambda-based application would launch a long-running task using AWS SQS or Kinesis as a queuing service and Fargate containers or standalone Lambdas as workers. Particularly complex or contrived tasks can also be carried out by starting serverless workflows using AWS StepFunctions.

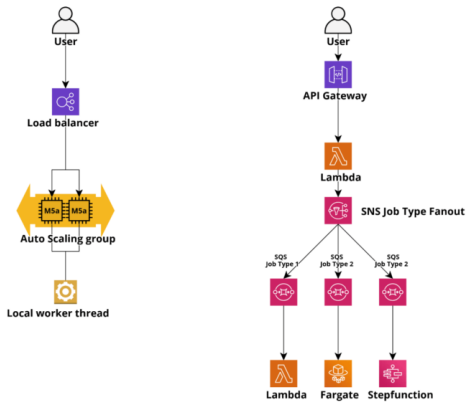

Figure 3: Long-running jobs in an AWS web application: EC2 based app on the left, serverless on the right

Finally, it is often useful to send notifications directly to the client browser in order to update the state of the frontend components in response to events such as the execution status of running tasks, notify the user of action carried out by other users (e.g. in online games), deliver messages and more and more often notify users about status changes of IoT devices.

Real-Time Notifications

While in classical applications it is possible to directly use WebSockets, even through AWS ELBs which supports HTTP/2, for serverless applications we need to leverage the AWS ApiGateway Websocket support which is also natively supported by several serverless frameworks, such as Chalice. When a Web socket connection is established by a client a lambda can be invoked by the *$connect* hook and it is able to register the connection id to a database, usually DynamoDB. When a user disconnects the *$disconnect* endpoint is invoked which allows our application to delete the connection from the connections table. Developing a logic to send notifications is thus pretty straightforward: when a message needs to be delivered from the backend to a user the ApiGateway @connections POST API is invoked using the id/ids of the user's open

WebSocket connections and ApiGateway takes care of forwarding the message in the open WebSocket channel.

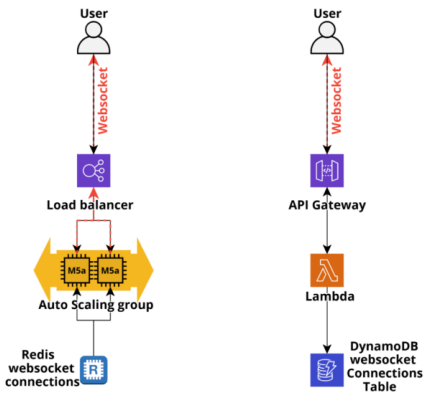

Figure 4: Real-time Websocket notifications in an AWS web application: EC2 based app on the left, serverless on the right

A real-world example

While these techniques are especially useful for high traffic applications, even low traffic apps can take great advantages by implementing them, especially those managing complex workflows.

The following example architecture is a greatly simplified version of the architecture we actually deployed for an application allowing a customer to dynamically manage S3 buckets and IAM users accessing them.

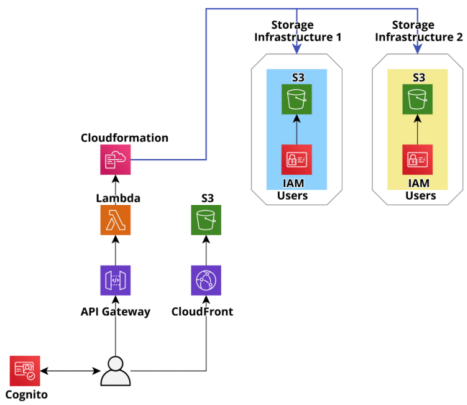

Figure 5: First version of the example application infrastructure

The application allows the customer's project managers to easily create secured S3 buckets in dedicated, project-specific, isolated, and hardened AWS Accounts. Once the buckets are created it is also possible to generate managed IAM users to access

them. The user credentials can then be shared securely with third parties to allow them to download and/or upload files from S3 using legacy on-premise systems, leveraging either the AWS S3 Apis or SFTP (AWS Transfer for SFTP). Users and Bucket can easily be added and removed from each project and User permissions can also be managed through a straightforward user interface.

In order to simplify the backend development and at the same time make the application more resilient, we decided not to set up any database and simply use the backend to create, update and modify Cloudformation templates. In this way the consistency of the AWS infrastructures in the Accounts is strongly enforced, each action performed is automatically logged, and in case of any error during the creation of a resource, the rollbacks are automatically executed by the Cloudformation service.

However, this approach has two main drawbacks, one for the LIST/GET operations and one for the CREATE/UPDATE operations.

In fact, each time an end-user lists the existing resources the backend needs to fetch all the CloudFormation templates, parse them to list the resources, and finally return the response. This flow needs to be executed at least on one template any time a user performs a LIST or GET operation. For accounts with dozens of buckets and users, this operation can take several seconds, making the user experience very poor and potentially, in a few extreme cases, breaking the 30-sec ApiGateway response limit.

The second problem involves update operations and manifests itself when several customers are connected: if one manager updates a bucket or user all the other users cannot modify it until after the Cloudformation execution is over, which in this use case only takes a few seconds. However other connected users have no way to know that a user is being updated and may try to modify it resulting in unexpected errors which, while harmless, make the user experience clumsy.

Beyond these problems there is also a further concern: since the described infrastructure is “GET heavy” towards the Cloudformation Apis if a big enough number of managers log in, the Cloudformation API rate limit could potentially be breached resulting in further slowdowns (AWS SDKs implement exponential backoff).

To resolve all these problems, after the initial MVP, we applied the design pattern described above to make the application production-ready: we used DynamoDB to cache the CloudFormation state and we used the builtin Cloudformation SNS integration to update the state in real-time through ApiGateway managed web sockets

connections and at the same time update the dynamo cache with the latest Cloudformation state, so that the state in dynamo always mirror the Cloudformation state.

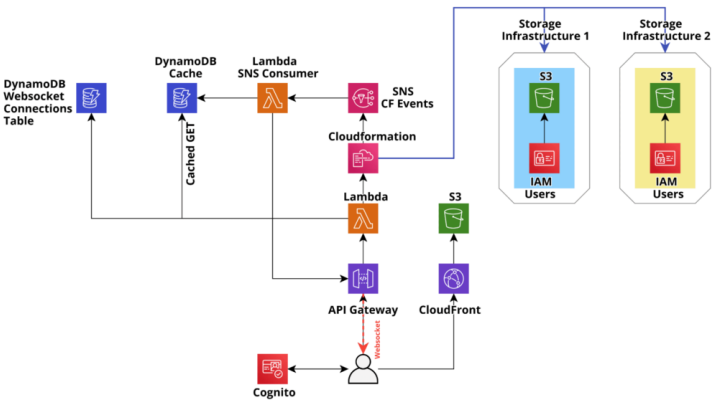

Figure 6: Final version of the example application infrastructure

This resulted in huge speed-ups, which completely changed the user experience for the better, with response times for GET/LIST requests dropping consistently below 200ms. Furthermore, CloudFormation updates also became significantly faster and the resource state is consistent on all the connected clients.

The customer was more than satisfied with the end result and a happy customer is always the best final confirmation of a well-done engineering job!

If you have any questions about DynamoDB design patterns or any other topic concerning (or not concerning) this article do not hesitate to **contact us!** We would love to discuss it and help you :)

See you again in 14 days!

Matteo Moroni

DevOps and Solution Architect at beSharp, I deal with developing Saas, Data Analysis, and HPC solutions, and with the design of unconventional architectures with different complexity. Passionate about computer science and physics, I have always worked in the

first and I have a PhD in the second. Talking about anything technical and nerdy makes me happy!
