Blueprint for a Serverless App

by Keith O'Connell

Senior Enterprise Architect

Serverless applications are turning the development world on its head; here we give you a basic blueprint for creating a serverless app.

2018

XTIVIA

WHITE PAPER

This article is a primer intended to outline an example blueprint for a serverless application, covering common components present in many serverless applications and the glue needed to stitch them together. We will be referencing services available through Amazon Web Services, though the concepts are common across cloud providers and analogous services on other cloud providers are certainly available.

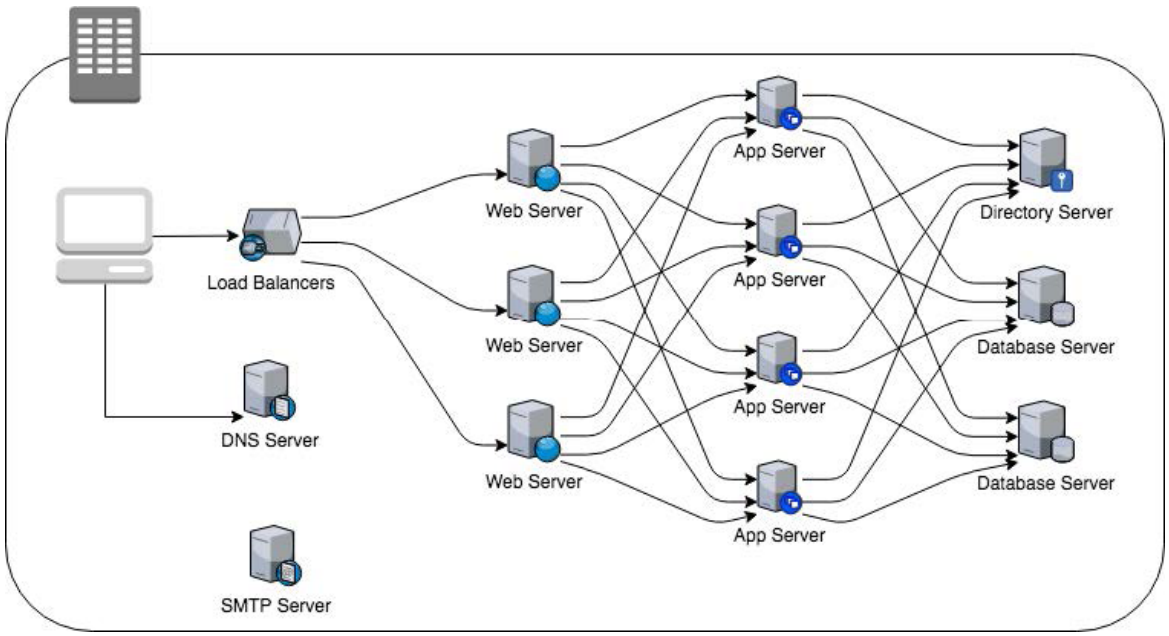
Serverless applications have been taking the development world by storm over the past few years; with the release of AWS Lambda in 2014, Amazon pioneered a new way of developing and delivering applications which has been proven to speed time-to-market while drastically reducing overhead. Each of the major cloud vendors rapidly followed suit, and at this point, serverless development seems to be reaching critical velocity.

What IS Serverless?

So, what IS serverless computing? In a nutshell, it's a platform that allows developers to write and deploy code without having to concern themselves with the infrastructure needed to run their code in a shared environment. As it happens, the word "serverless" is actually a bit of a misnomer; the servers are certainly still there, they're just managed by someone else. It's similar to the difference between owning a car and hiring a rideshare service; with the former, you have to buy the car, set it up so that you can drive it, change the oil, get it repaired, etc...with the latter, you just request a ride, and the car shows up, takes you where you want to go, and drops you off. No muss, no fuss.

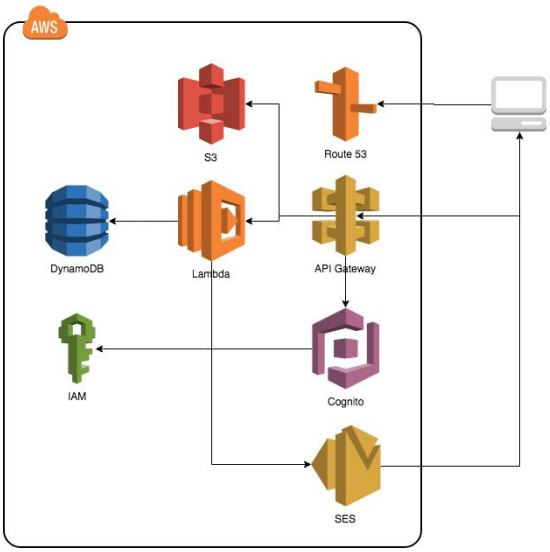
What Does "Serverless" Look Like?

With that very, very high level explanation of serverless computing, let's jump in and talk about how a serverless application is designed. An example traditional 3-tier application might look like this:



This is a familiar architecture; you have a browser-based client communicating with a load balancer, which sends requests to a proxy HTTP server tier, which forwards requests to the application server layer that hosts your business logic. The application server layer coordinates identity management through a directory server or identity provider, retrieves persisted data from a database server farm, and potentially integrates with other, internally managed services before returning the result back to the user. Of particular note is the complexity needed to ensure a highly-available solution; a large number of systems must be created and maintained in order to try to avoid a single point of failure and provide scalability, and despite this, there are still a number of ways that this architecture can fail catastrophically.

Taking this application and moving it into a serverless design will result in an application architecture that looks like this:



Looks a lot more manageable! We've now got the client tier leveraging a set of services, provided and maintained by Amazon, to get work done. All of the functions that were contained in the application server are now independently deployed and managed, and we've offloaded the administrative overhead of creating, managing, maintaining, and scaling the infrastructure required to provide those services to the cloud vendor (Amazon). There's a somewhat glib saying in the information technology world that "serverless is really just someone else's servers"; while this is usually meant to poke fun at the concept of serverless architectures, it underscores a critical point: with a serverless architecture, managing the infrastructure needed to provide business solutions becomes someone else's problem. Your team's effort can be focused solely on developing application functionality and thereby addressing business needs.

Breaking It Down

Next up, let's take a look at the AWS services that we included in the diagram above, and dig into why we'd include that service in our application. Note that there are literally hundreds of services provided by AWS; we chose the services that would most closely map to an N-tier application for this example, but a serverless topology could easily leverage many other services depending on the business need.

Here's a breakdown of the components that we're using in our serverless application:

	Lambda: Custom application code execution
	API Gateway: API management and request routing
	Simple Storage Service (S3): static asset storage (including frontend code)
	DynamoDB: Application persistence layer
	Cognito: User authentication and authorization
	Identity and Access Management (IAM): Fine-grained permissions management
	Route 53: Naming integration and failover routing, high availability
	Simple E-mail Service (SES): E-mail integration

Lambda

Lambda is the bread-and-butter serverless service that Amazon provides; it is THE service that founded the concept of "serverless". It is essentially Function-As-A-Service (FAAS); a straightforward service allowing developers to create "functions"; lightweight, atomic pieces of code that are executed in order to respond to specific events. For many serverless applications, all requisite business logic will be contained within Lambda functions, which will automatically execute in response to web requests, scheduled tasks, queue operations, or any number of other trigger events inside or outside the AWS ecosystem. AWS Lambda supports functions written in a number of languages, including Node.js, Python, C#, Java, and Go. Lambda functions are written in the same manner as traditional applications; they can include references to third-party libraries and invoke other services (both AWS-based and external). In an N-tier system, the closest analogue to Lambda would be the code running on the application servers.

API Gateway

The AWS API Gateway is a service that provides powerful API management capabilities, allowing you to create, publish, version, secure, maintain, and monitor application interfaces at scale. In layman's terms, the API gateway acts as the gatekeeper for your Lambda functions and other AWS services, routing requests and traffic as needed based on your application's requirements. Some of the functionality provided by the API Gateway would map to the application server and the load balancer in a traditional N-tier system, though the API management part of the API Gateway's functionality has no peer in a basic N-tier application.

Simple Storage Service (S3)

The Simple Storage Service, or S3, is an AWS service with which you may be familiar. The basic purpose of S3 is to provide object-based storage at massive scale; in a basic serverless application, we commonly use S3 in two ways: to store and deliver assets required for the serverless application to execute (including the packaged code that Lambda functions use), and as persistent storage for files or objects which are processed as a part of the application. S3 fulfills some of the functions provided by both the application server and the web servers in an N-tier application.

DynamoDB

DynamoDB is a non-relational datastore often used in serverless implementations to maintain application state across function invocations. Serverless applications are particularly well-suited to nonrelational persistence, and for AWS-based applications, DynamoDB provides a fully-managed, resilient NoSQL database engine which can operate at massive scale. In an N-tier application, the standard relational or non-relational database servers would serve this function.

Cognito

One AWS-based service that provides exceptional value to the serverless space is Cognito; Cognito provides a comprehensive solution for user management, authentication, authorization, and access control which is easily integrated with the API Gateway. It provides a secure user directory as well as the ability to federate users through external identity providers such as Google, Facebook, or your own Microsoft Active Directory instance. It also provides fine-grained access control for AWS resources, so that you can easily control what specific services and resources users can access in your application. Cognito user pools take the place of a Directory Server in an N-tier architecture, and the authentication, authorization, and access control take the place of significant amounts of custom code and configuration that would need to be deployed within your application.

Identity and Access Management (IAM)

The Identity and Access Management (IAM) service is used by Cognito to set up role-based access control for resources in your application; IAM is used pervasively in the AWS ecosystem to create, manage, and enforce permissions. There is no direct equivalent in an N-tier application other than the custom code that your team creates to limit user access to parts of the application.

Route 53

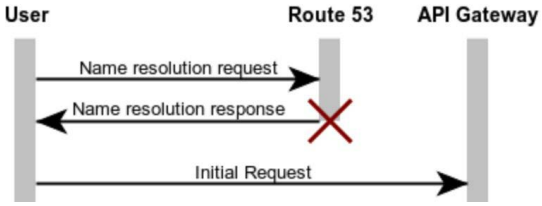
Route 53 is Amazon's Domain Name Service (DNS) implementation. It provides name resolution and routing capabilities to users and services alike; unlike a traditional DNS system, though, Route 53 was built specifically to provide routing and failover at a global level. In a serverless application, Route 53 is most commonly used to provide scalability at a global level, providing routing and failover across multiple AWS regions. In an N-tier application, some of the functionality provided by Route 53 would be provided by internal DNS servers, while other parts might be provided by your networking infrastructure.

Simple E-mail Service (SES)

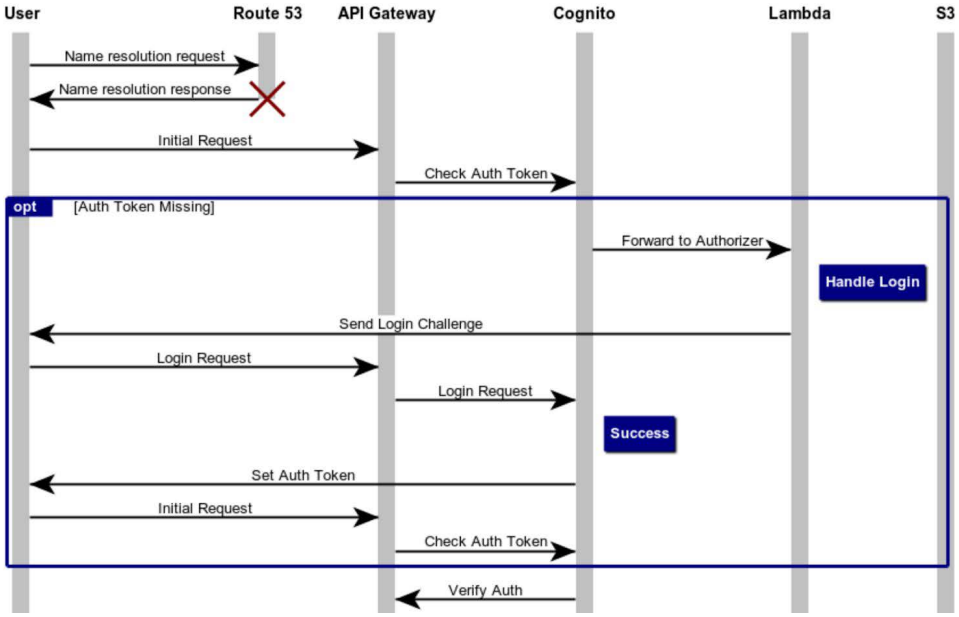
The Simple E-mail Service, or SES, is essentially E-mail-as-a-service; it provides a way to send and receive large amounts of e-mail via SMTP. It is included as an example of a consumable service that is often used in N-tier applications; in traditional applications, it would be necessary to rely on an internally-managed SMTP server to provide this functionality.

Stitching AWS Services Together

To help clarify how a serverless application would work in practice, let's look at the sequence of events that happens in the course of serving up a typical application request.

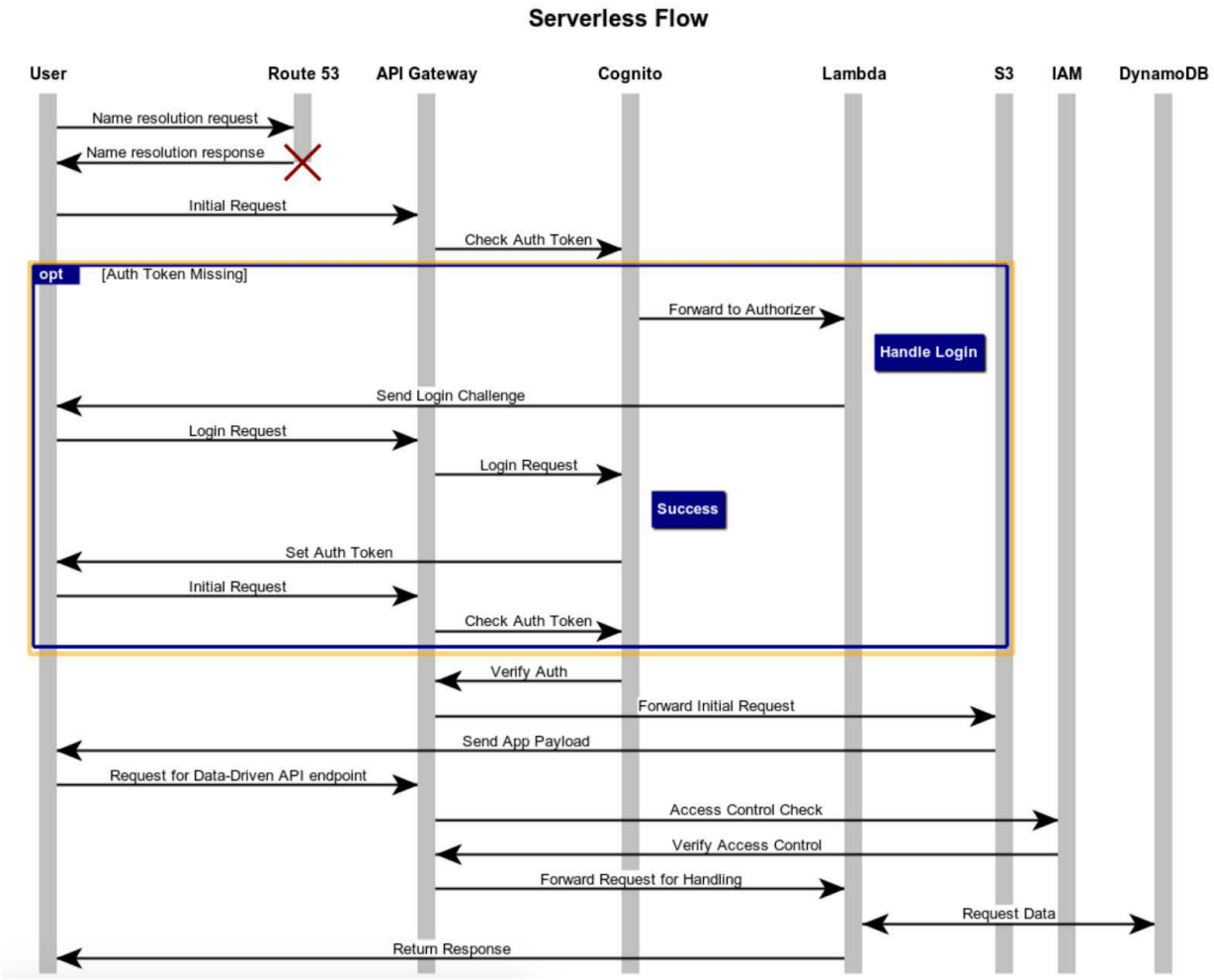


The lifecycle starts off with a user typing in an address, clicking on a link, or opening an app to go to the application in their browser or mobile device. This triggers a DNS request to resolve the application's address via Route 53. Route 53 returns an IP address which points to our AWS API Gateway, and the client sends an HTTPS request over to the API Gateway.



If the client has previously been authenticated, the authentication token from the client is sent over to Cognito for validation; otherwise, the request is forwarded to Cognito for authentication, presenting the user with a login dialog. The user sends their authentication response back to the API Gateway, which forwards to Cognito for authentication validation. If successful, Cognito informs the API Gateway of the success, and the request is forwarded on. If authentication fails, the user is presented with the login dialog again with a contextual error message.

Once authentication has succeeded, the initial request from the client is forwarded over to an S3 bucket which contains the frontend payload for the mobile application or single-page web application (SPA). This payload is delivered back to the client.



Subsequent requests are sent by the SPA or by the mobile application to the API Gateway; these requests are forwarded to appropriate Lambda functions based on request routing rules. For example, if a request from the client requires data from a DynamoDB table to complete, the request would be sent to the API Gateway, which would check the request's authentication token against access control information stored in AWS IAM roles. If the request has permission for the underlying route, the request is then forwarded to a Lambda which retrieves the requisite data from DynamoDB. The response containing the data will then be sent back to the client.

This provides a basic example of how requests would flow between services in an AWS-based serverless application; while the service topology and overall flow will vary from application to application, the general concepts will be the same.

Delivering a Serverless Application

A successful rollout of any application is rooted in the process that you use to manage the development and deployment process. While the serverless architecture approach simplifies and abstracts a number of the common problems that plague application development, it is still critical to carefully plan and execute the process used to create and deploy your code. At XTIVIA, we have developed a set of standard practices that help us maintain consistent quality when developing serverless applications.

These practices are as follows:

1. Manage all assets, both custom code and service configuration, in a version control system.
2. Use continuous integration to build and test each and every commit to version control in a controlled environment.
3. Use distinct AWS accounts for development, QA, and production.
4. Service provisioning and configuration must be automated.
5. All deployment activities must be automated.
6. Rely on a single source of truth for pipeline and deployment activities.

Delivery in AWS

Applying these principles to our AWS-based serverless example is straightforward; by adhering to the following pipeline:

1. Code is developed and tested locally by developers using local containers that mock AWS services (via sam-cli).
2. Code is then committed to a version control system. Upon commit, the code is built, and unit tests and automated functional tests are run by an internal continuous integration engine.
3. If all tests pass, the code is packaged and uploaded by the CI engine to the development artifact repository, a versioned, private S3 bucket.
4. The updated Lambda function is deployed by the deployment system and a notification goes out that a new version is available for testing on the QA AWS account.
5. Once the code has been successfully tested (potentially using a fully automated test suite), a tag is created on the commit indicating that it is ready for promotion to production.
6. The deployment service runs a separate deployment job on tag creation which promotes the tagged commit into the production AWS account.

Note that this is a very high-level summary of the steps in our sample delivery pipeline; a separate post covering concrete implementation details for automating delivery of serverless applications in much greater detail will be published soon.

Summary

Serverless architectures are becoming more and more popular as a way to speed up development, minimize overhead, and drastically improve the process of developing custom applications. The proliferation of cloud-based serverless services by providers such as Amazon, Microsoft, and Google, along with the availability of in-house serverless frameworks such as Fission, has brought serverless application development within reach of any organization. While serverless architectures are no silver bullet (and we will be covering some serverless "gotchas" in a separate publication), the benefits that they provide from both an implementation and delivery standpoint are undeniable.

Are you interested in exploring how to start implementing serverless applications in your organization? XTIVIA can help! With a long history of enterprise application implementation coupled with cutting-edge cloud-native architectural and development experience, XTIVIA is the perfect partner to help you bring your enterprise to the cloud. Contact us today!

About the Author

Keith O'Connell

Senior Enterprise Architect

Keith O'Connell is a Senior Enterprise Architect with more than 20 years of experience designing, developing and optimizing large-scale enterprise software applications. He has successfully managed both development and delivery initiatives for a number of large-scale software implementations spanning multiple vertical markets, from public-facing entertainment and e-commerce sites to internal portal and data analysis sites for large international corporations. He has a keen interest and deep expertise in Cloud Architecture and DevOps, and is always looking for ways to innovate and optimize their delivery. Keith lives in Austin, TX, with his wife, daughter and two dogs.

XTIVIA

If you can imagine a business outcome, XTIVIA can create it through technology.

XTIVIA does what it takes to ensure customer success through adaptive technology solutions. Our earned reputation is for delivering the right IT solutions and support that meet our customers' specific requirements, regardless of project complexity. Our passion, combined with a dedicated leadership team and unparalleled technical staff, creates customer relationships that stand the test of time.


xtivia.com/blog


linkedin.com/company/XTIVIA


twitter.com/XTIVIA


youtube.com/c/XTIVIA


(888) 685-3101 ext.2