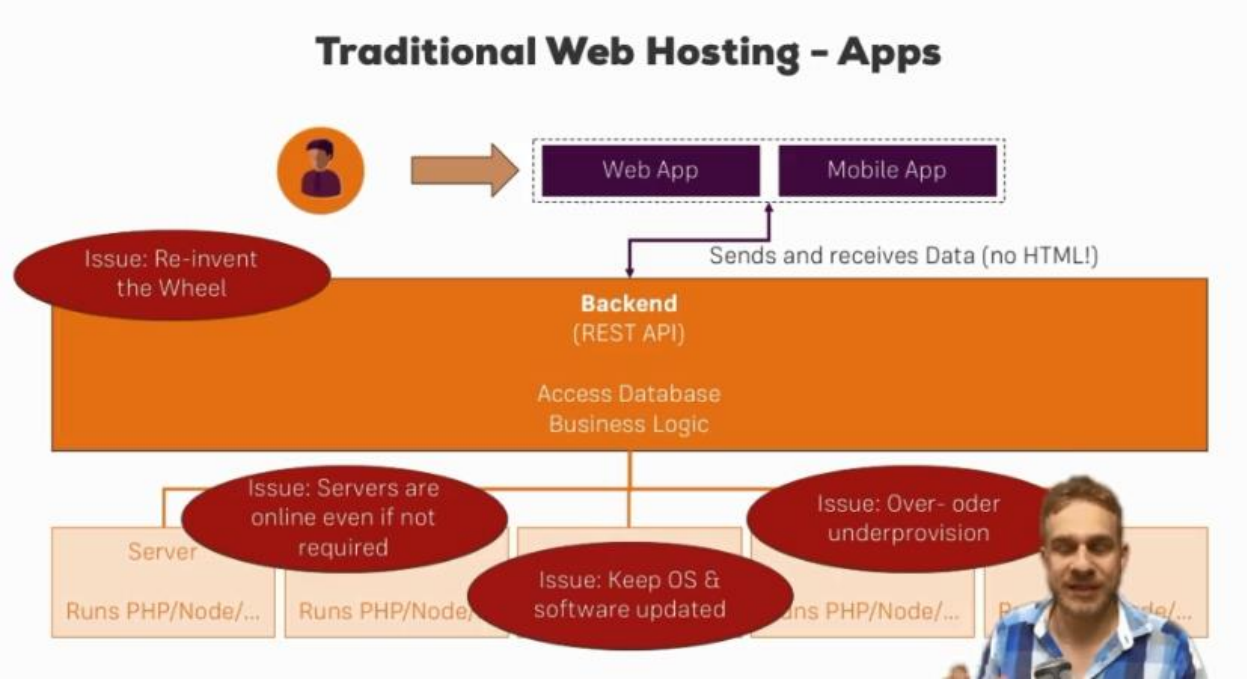
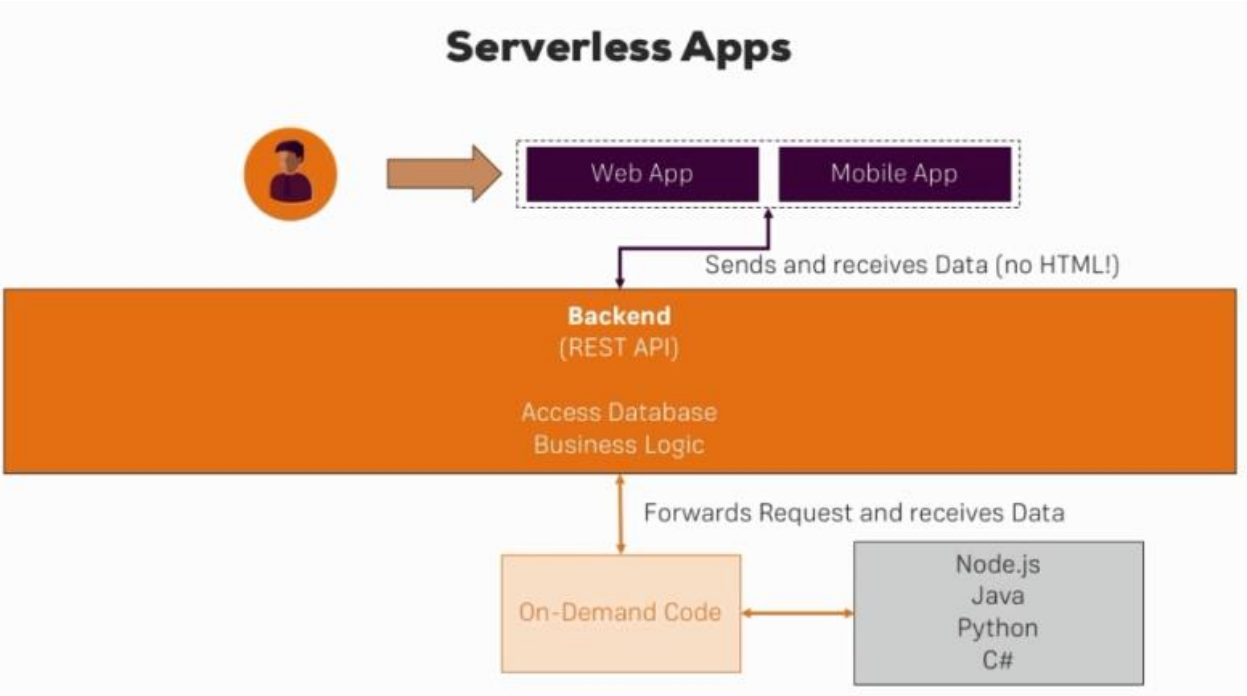
AWS Serverless APIs & Apps - A Complete Introduction

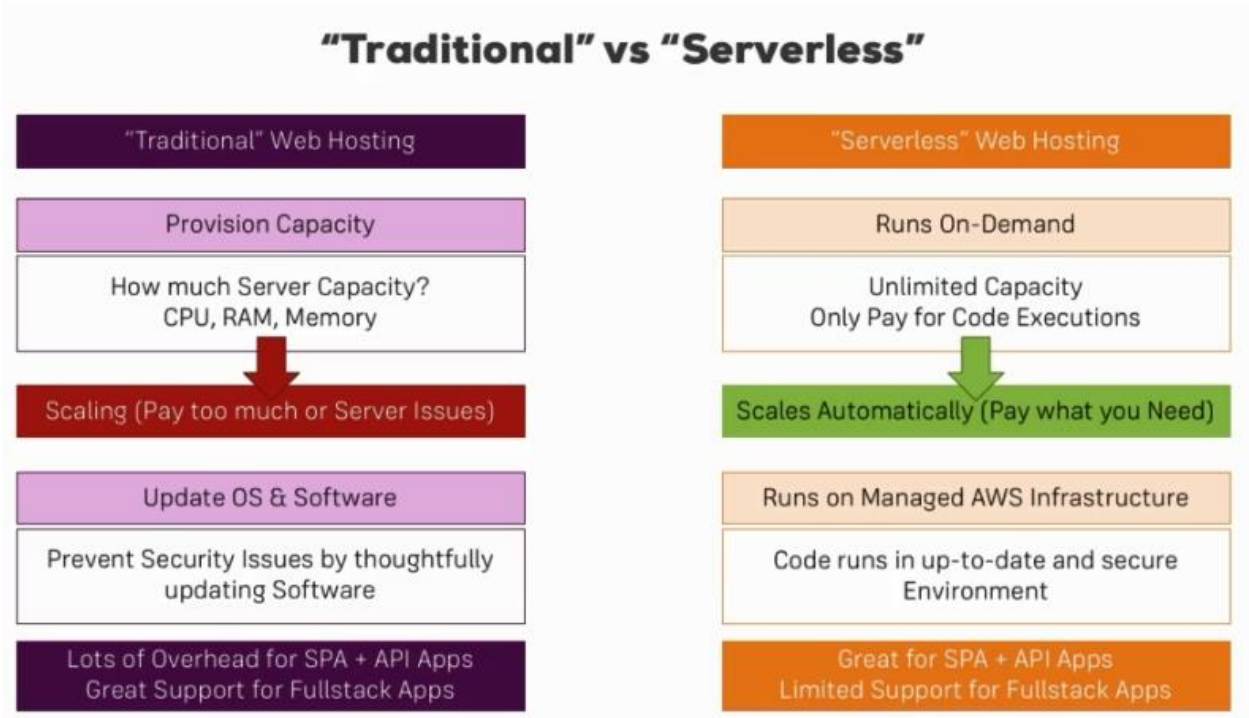
Contents

Getting Started

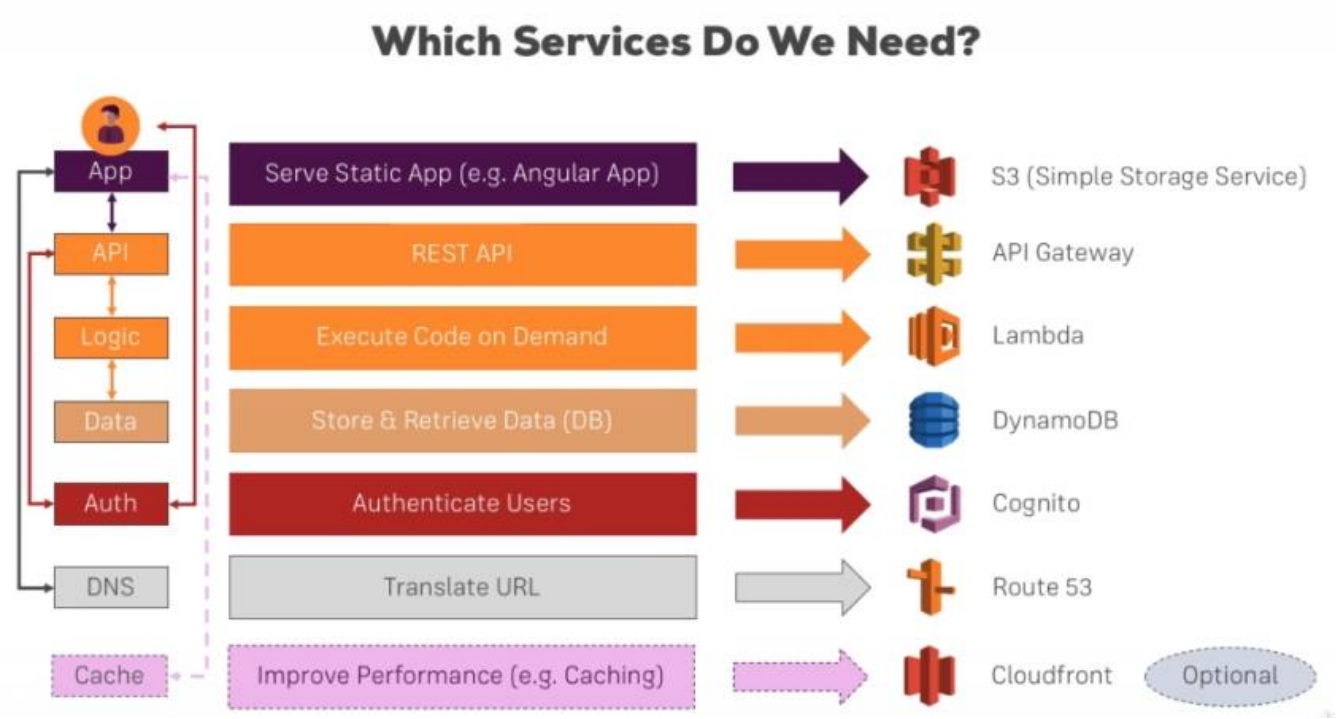
Serverless Development






"Traditional" vs "Serverless"
"Traditional" Web Hosting
Provision Capacity
How much Server Capacity?
CPU, RAM, Memory
Scaling (Pay too much or Server Issues)
Update OS & Software
Prevent Security Issues by thoughtfully updating Software
Lots of Overhead for SPA + API Apps
Great Support for Fullstack Apps
"Serverless" Web Hosting
Runs On-Demand
Unlimited Capacity
Only Pay for Code Executions
Scales Automatically (Pay what you Need)
Runs on Managed AWS Infrastructure
Code runs in up-to-date and secure Environment
Great for SPA + API Apps
Limited Support for Fullstack Apps

AWS Core Serverless Services



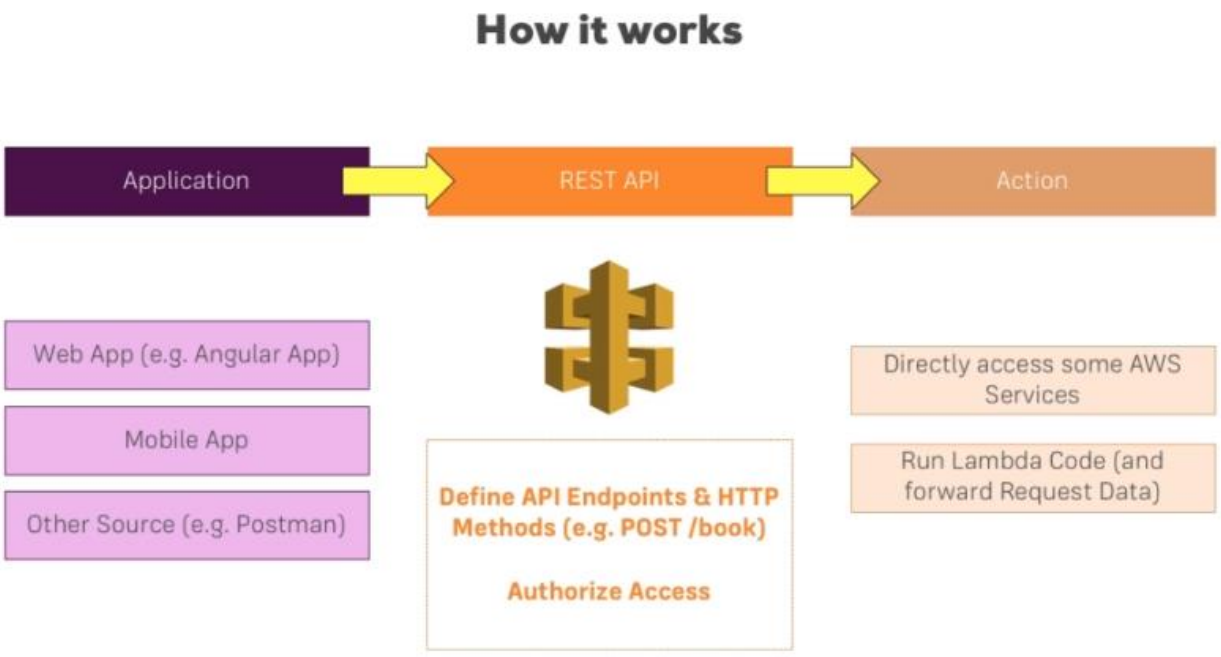
- **S3** is the Simple Storage Service from AWS where we can host our **static** files. Hence we can store our html, css, js files related to SPAs.
- **API Gateway** is a service which makes it easy for us to create an API with different paths and HTTP methods we want to handle.
- AWS **lambda** is a service which allows you to execute code on-demand. This can be used to execute some code when we receive some request at our API endpoint. Apart from responding to API requests, we can also execute code for different events.
- **DynamoDB** is a great solution for database. It's a NoSQL database and it's a database where we don't have to provision any database servers, so we can just store data as we get it.
- AWS **Cognito** is a service where we can easily create user pools to allow users to sign-up and then sign themselves in to use other services. We can protect the rest API for example so that only authenticated users are able to access certain or all of the endpoints we provide there.
- **Route 53** allows us to register and configure our own domain so that whenever we receive a request to this domain, we actually load the web page from our S3 bucket or our S3 file storage.
- **Cloudfront** is useful because it will basically copy your static files which lie in S3 to different destination all over the world so that people accessing your web page will always have the quickest route possible (kind of caching).

Creating an API with API Gateway & AWS Lambda

API Gateway

Introduction

- With API Gateway, we can create Restful APIs.
- You could also easily implement things like authentication with API Gateway, so that you can ensure that only the users of your app which are authenticated are able to fetch certain data or send certain requests.
- With the API Gateway, you can then directly access other AWS services for example or one of the most important services where we can trigger an AWS lambda function.



General API Gateway Features

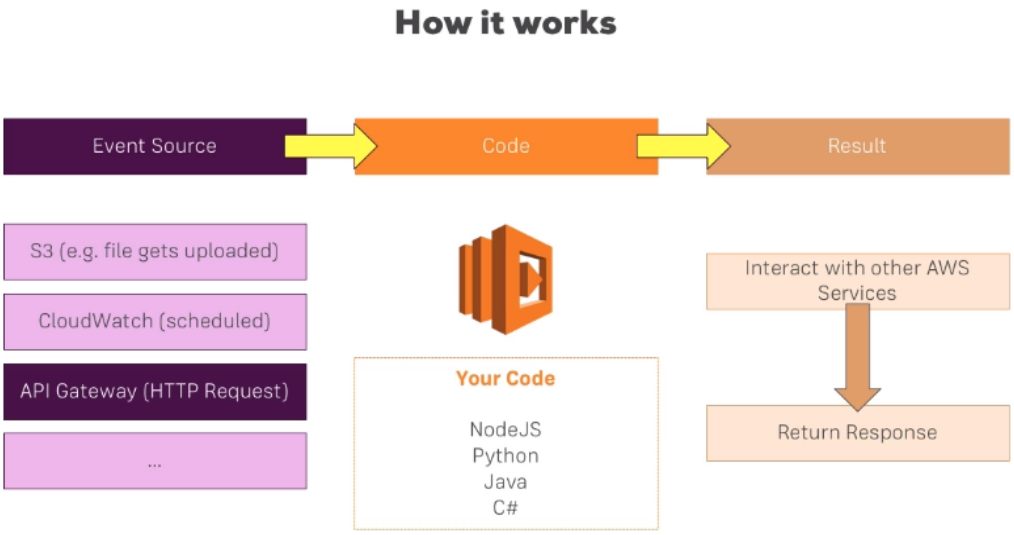
- API keys are interesting if you plan on creating an API which is shared with other developers, so not with users of your app but with other developers creating their own apps. For instance – Think about the Google Maps API. They have an API where you can send coordinates and get back address information. Now if you want to use that, you have to register on Google and you will get an API key by them. It's that key which you then have to pass with any request you send to the API because you identify yourself with that key and Google can also track your usage of the API and possibly limit you if you exceed your limits or anything like that and you can do the same here in API Gateway.

- By default, AWS doesn't give any permissions to any of your services. That means, that no service may interact with other services.

Request-Response Cycle

- **Method Request** - defines how requests reaching this endpoint should actually look like. It's like a gatekeeper, it ensures that incoming requests have a certain shape, have certain data, and fulfill certain requirements. (validation, authorization, model mapping)
- **Integration Request** - is about mapping incoming data or basically transforming incoming data into the shape we want to use it on the action we're about to trigger. The role of integration request is to trigger this endpoint, be that a mock endpoint, the lambda function or something else and possibly if we want that, transform our incoming request data (body data, headers or metadata like authenticated users). So it allows us to extract the data and pass it on to the endpoint.
- **Integration response** is the first thing which gets triggered as soon as our action is done. It allows us to configure the response we're sending back. Just opposite of Integration Request like transformation, etc.
- **Method Response** defines the shape of our response. Here we can simply configure possible responses we're sending back, so we could set up different status codes here too and on this response, we can set up which headers this response should be allowed to have and which type of data it should send back and here we can again use models we might have defined, so that we can be clear about which shape of data we're sending back.
- Method response defines the shape of the response, integration response fills that shape with life.

AWS Lambda



- AWS lambda that is a service which allows you to host your code on it and run it upon certain events.
- We get different events sources, for example
    - In S3 file storage service, a file upload could trigger a lambda function which then will receive info about that file as an input and could possibly transform it. This is useful if you have a service where people upload photos of themselves and you want to compress them, maybe analyze them with machine learning, you could do all that with lambda.
    - Another example is cloud watch, where you can schedule events, so you can basically trigger a new lambda execution every X minutes, hours, days. It's useful for cleanup jobs or anything which runs on a regular basis.
    - You can always use API Gateway to run the code whenever a request hits the API Gateway.
- Languages supported by Lambda are – NodeJS, Python, Java, C#.
- So when an event is triggered, the code in the AWS Lambda gets executed once your defined event occurs and in this code, you can do any calculation, reach out to other AWS services like a database to store or fetch data but any service is possible, you could also start sending mails, etc. and in the end, you will return a response or you will execute a callback to indicate that this function is done, can shut down and possibly if needed, pass any data back to the event source who called it.
- This is how lambda works.
- In conjunction with API Gateway, we can have multiple endpoints in API Gateway, some methods and resources and which trigger different lambda functions for each of these endpoints.

Understanding Body Mapping Templates

- Body Mapping Templates can be confusing, but in the end, they're not that difficult.
- You use the Apache Velocity Language when working with these templates, that will only matter once you create more complex transformations though. Basic explanations (see below) should be relatively self-explanatory
- First, you need to understand that they simply allow you to control which data your action receives (e.g. Lambda).
- You can for example transform incoming data of the following shape:

```
{
    person: {
        name: "Max",
        age: 28
    },
    order: {
        id: "6asdf821ssa"
    }
}
```

To

```
{
    personName: "Max",
    orderId: "6asdf821ssa"
}
```

with the following template:

```
{
    "personName": "$input.json('$.person.name')",
    "orderId": "$input.json('$.order.id')"
}
```

- $input is a reserved variable, provided by AWS which gives you access to the input payload (request body, params, headers) of your request.
- json() is a method on $input which will retrieve a JSON representation of the data you access. $ here stands for the request body as a whole, $.person.name therefore accesses the name property on the person property which is part of the whole requests body object.
- The $input.json(...) code is wrapped into quotation marks (") because it should be transformed into a string in this example (since both name and id are strings). If you were to access a number, boolean or object here, you would NOT wrap the expression in quotation marks.
- That's is the power of body mapping templates. You can clearly separate lambda from API Gateway and you can control what goes into lambda and what comes out of lambda.

Using Models & Validating Requests

- We define models via JSON Schema Language.
- E.g.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "CompareData",
  "type": "object",
  "properties": {
    "age": {"type": "integer"},
    "height": {"type": "integer"},
    "income": {"type": "integer"}
  },
  "required": ["age", "height", "income"]
}
```

- Models are defined using JSON schema.
- JSON Schema: http://json-schema.org/
- JSON Schema: https://json-schema.org/understanding-json-schema/index.html
- You can add the same validation in Lambda as well. However the idea of using a model for that and using this built-in validator, API Gateway offers you simply is to give you an easier way of adding such implementation. You get an out-of-the-box working implementation of validation which returns an error if it fails and that can be very convenient but again, it's optional.
- You can use models for validation, mapping, etc.

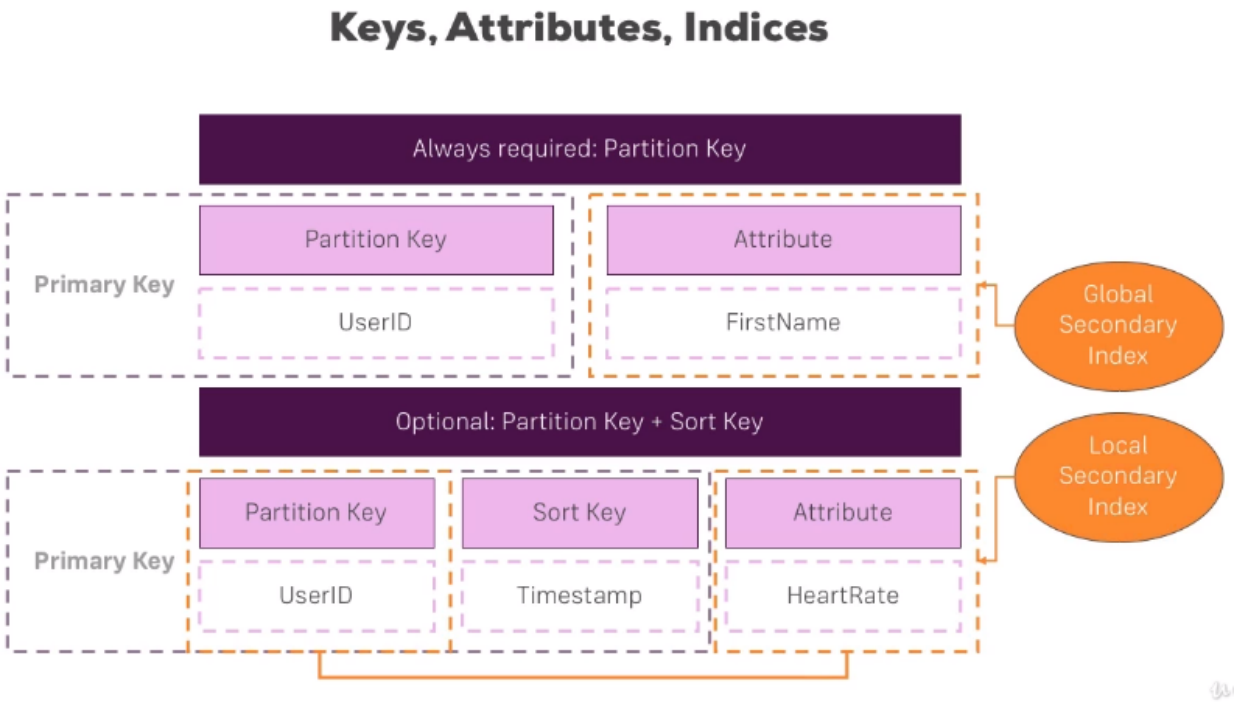
Data Storage with DynamoDB

DynamoDB

Introduction

- DynamoDB is a fully managed NoSQL database provided by AWS. This means you don't have to take care about provisioning any database services and with NoSQL, means there are no relations in that database.
- Why is DynamoDB a great fit for serverless architectures? Because it's fully managed and you don't have to spin up any database servers.
- DynamoDB stores data in key-value pair.

How DynamoDB Organizes Data

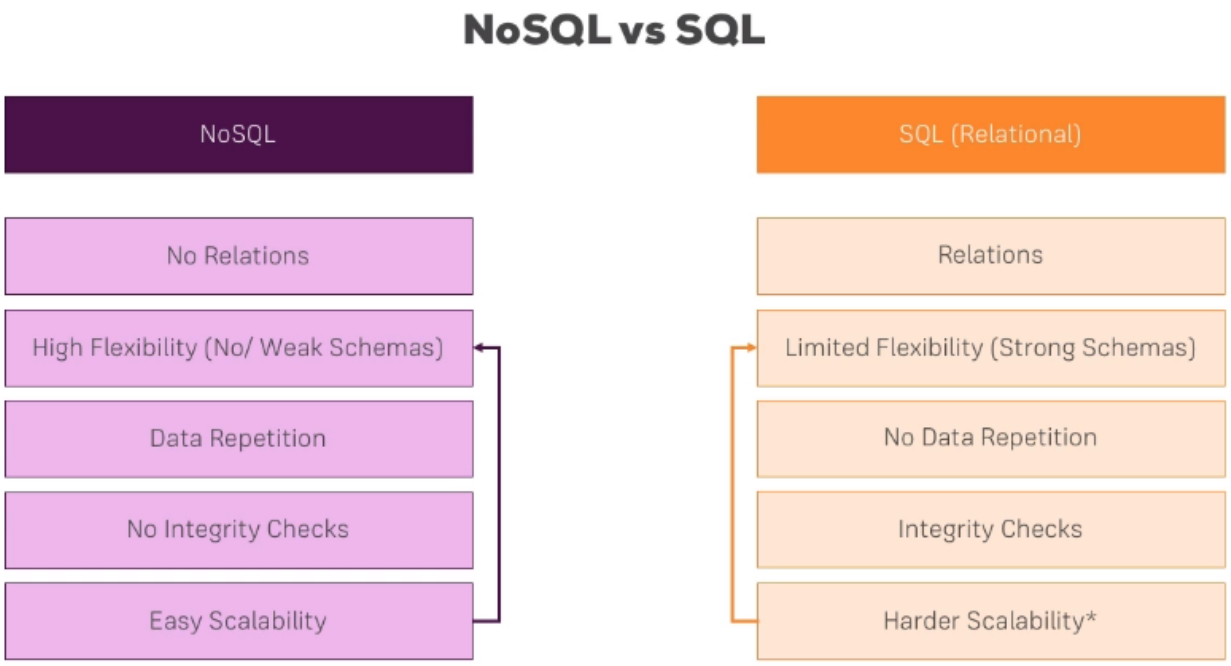


- In DynamoDB table, we're always required to have a **partition key**. Then you may have as many **attributes** as you need but that partition key is a must.
- Partition key must be unique.
- Why is it named partition key? That's related to how DynamoDB stores your data behind the scenes.
- It uses like a fleet of solid state drives and there, it stores your data and it tries to do that efficiently by partitioning these state drives. To put it in a very simple example, you could say it stores them by letters A to Z, of course it is much more complex than that but if

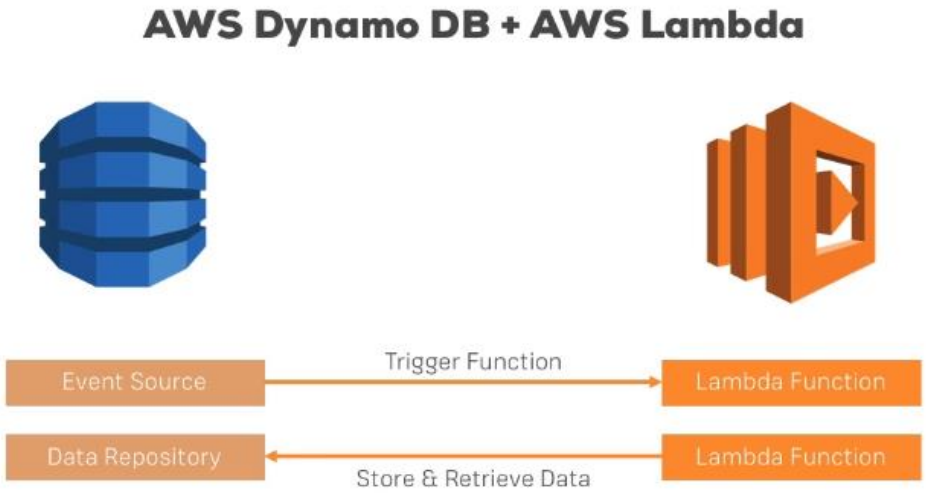
your partition key's value starts with A, it's going to be in the A partition, if it starts with C, it could be in the C partition.

- For a very efficient distribution which makes accessing it as fast as possible, you should aim to split your data amongst many partitions though. So it would be better to have random partition keys.
- Instead of just single unique partition key as a primary key, you can use a partition and a sort key as a primary key. As of combination, **partition key + sort key** must be unique as it will be used as primary key.
- We can then setup **global secondary index** on attributes, at max 5 indices per table. This forces AWS DynamoDB to kind of manage this attribute in a special way, in a more optimal way so that it's aware of this, it automatically makes querying this much more efficient and then therefore also allows you to query this secondary index.

NoSQL vs SQL



DynamoDB with AWS Lambda



Accessing DynamoDB from Lambda

- The great thing is AWS gives us a package, a tool we can use to use all its services, the many services it offers, not just from within AWS Management console but programmatically via **AWS SDK**.
- There are AWS SDKs for different languages.
- AWS SDK for JavaScript: https://aws.amazon.com/sdk-for-javascript/
- With AWS Lambda, the full SDK is provided in each function by default. So we don't need to install the aws-sdk package. You can simply access the AWS SDK in any of your lambda function no matter which language you're using.

Policies and Roles

- It is good practice to allow only required execution roles for our Lambda functions.
- We can create new policies and roles, attach those policies to roles, add that role to our lambda function. This will ensure that our lambda function has only least required roles and not ALLOW ALL kind of roles.

Authenticating Users with Cognito and API Gateway Authorizers

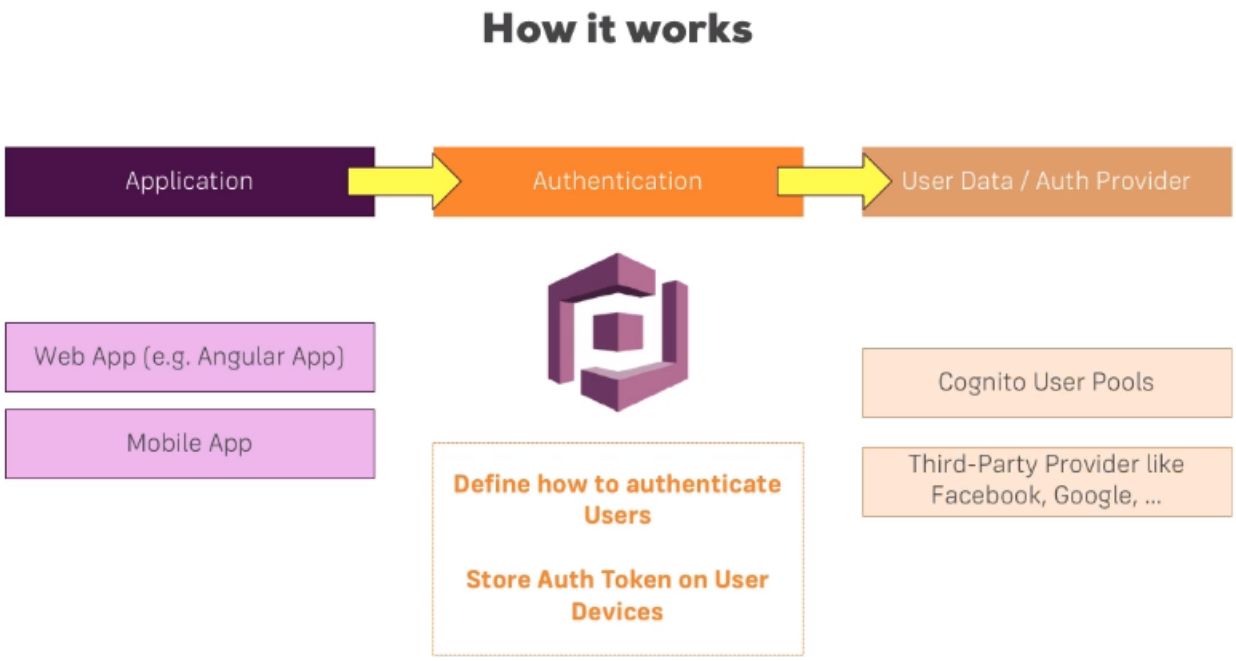
Custom Authorizers

- Refer – https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-use-lambda-authorizer.html
- A custom authorizer uses lambda behind the scenes.
- It tells API Gateway to call a specific lambda function, pass some information from the incoming request to that function and that function then has to run some code to validate or to identify that user.
- Building custom authorizer is a big task if we want to cover all cases. For simple cases, we can build it but it may not be 100% secure.
- The lambda function has to return an **IAM policy** which allows or denies access and an ID of the user (**Principal ID**), that's especially useful if access was granted then our API Gateway knows who is this person accessing our API. And additionally we may return additional piece of data (**Context** object). Refer to see what is expected in the output - https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-lambda-authorizer-output.html
- Now in order to be able to do that, our lambda function also gets some information. Refer to see which information it gets – https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-lambda-authorizer-input.html
- We should use Custom Authorizers if we have existing authorization logic in place.

AWS Cognito

- Custom Authorizer is not so great if you don't have an existing authentication logic, if you don't have an existing user base or way of signing users up and so on.
- And this is why AWS gives you an out-of-the-box working service which will allow you to easily add users, sign users up, confirm users, reset passwords, authenticate users, generate these tokens which we then can use and add verification with some easy set up steps.
- Besides that, it even offers more than that, you can easily integrate it with other providers like Facebook or Google.

How AWS Cognito Works

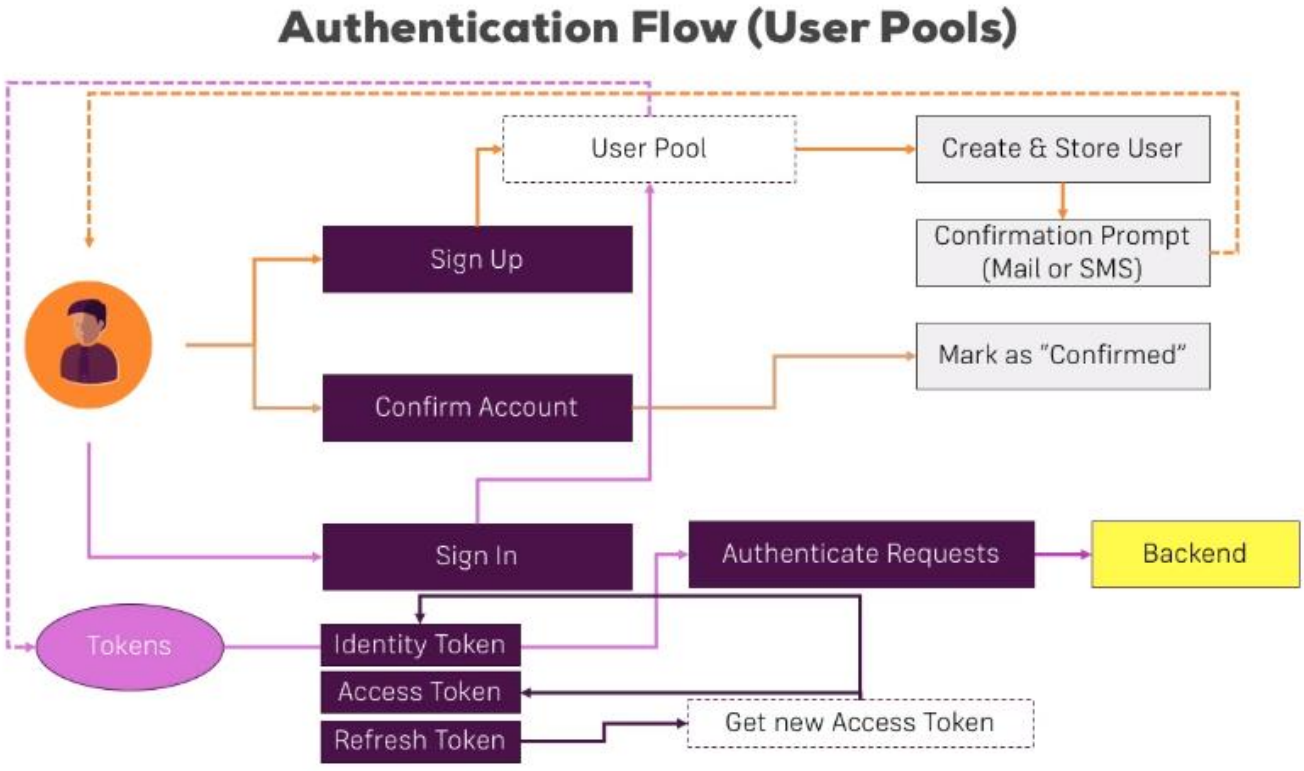


- So Amazon Cognito is useful if we have an application, be that a web app or mobile app where we want to add authentication.
- Amazon Cognito allows us to define authentication logic and then finally issue tokens to these applications, which these applications may use.
- Cognito can do even a bit more than just issue tokens, it can also issue temporary IAM credentials, so that you get full access or full access depending on certain roles you assign.

Options

- Amazon Cognito offers user pools and identity pools.
- **User pools** is a complete solution for people with no authentication process at all and don't want to use third-party providers like Facebook or Google. There you get, as the name suggests, user pools where users can sign-up, sign-in and it's really easy to integrate.
- **Federated identities** would allow you to connect third-party providers like Facebook, like Google to Cognito to then create temporary IAM credentials, to issue these users with the rights of performing certain actions depending on which credentials you provided.
- Managing your own user pools and therefore integrating users sign-up and sign-in into an integrating SPA is much more on that side, is much more something you will probably need when creating a serverless application.

Cognito User Pool Auth Flow



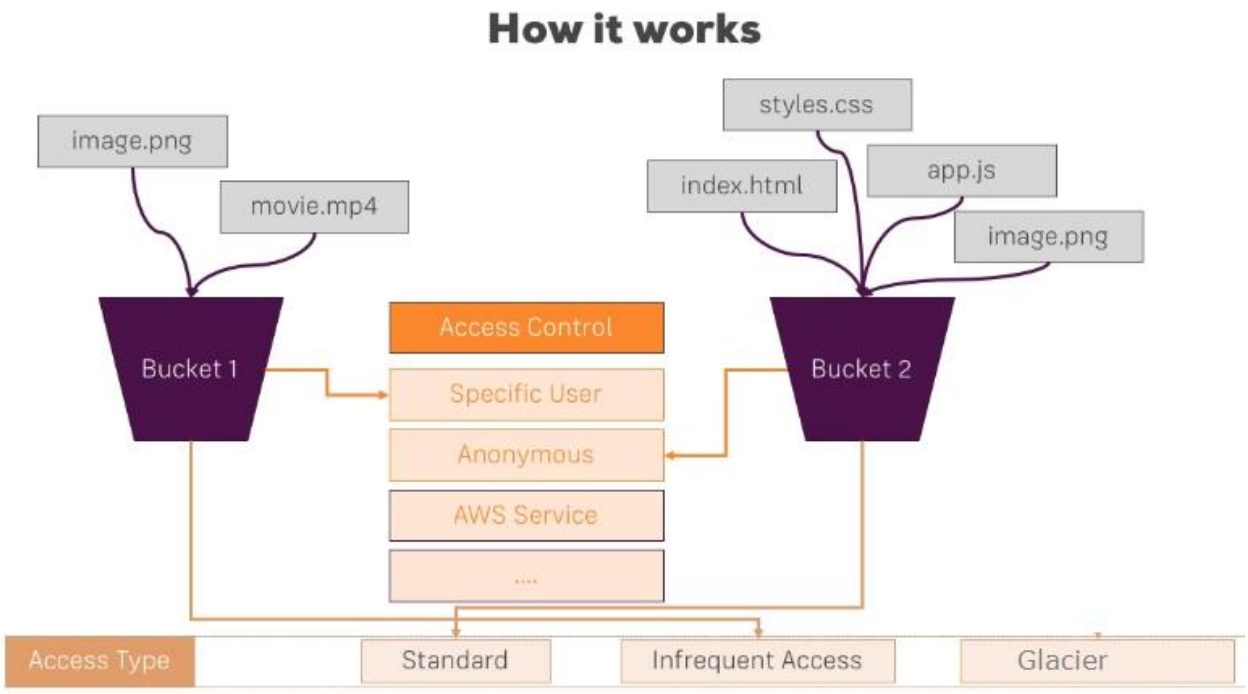
- Once we Sign In after Signup and Email Verification, Cognito issues a couple of tokens to our front-end application. We get three tokens to be precise - the identity, access and refresh token.

- The **identity token** is the one we're sending to our back-end to authenticate requests (in the Authorization header). It holds some information the back-end can use to check this. The backend will be API Gateway in our case and of course, we will need an authorizer for this.
- The **access token** can be used to get for example or to send it to Cognito if we want to change some user attributes for example.
- The **refresh token** required to get new identity and access tokens because these tokens are not living that long, only one hour for security reasons. If they were stolen, then someone else could send authenticated requests and we don't want it. Of course as long as we use HTTPS and we protect ourselves against cross-site scripting attacks because the tokens are typically stored in local storage which could be accessed through Javascript, as long as we do that, the token should be safe, still it is important that it's not living that long.

Hosting a Serverless SPA

Amazon S3

- Amazon Simple Storage Service
- It's a object storage service and it's there for you to store all kinds of files, that can be web app related files like HTML, CSS or image files but it can also be any other files.

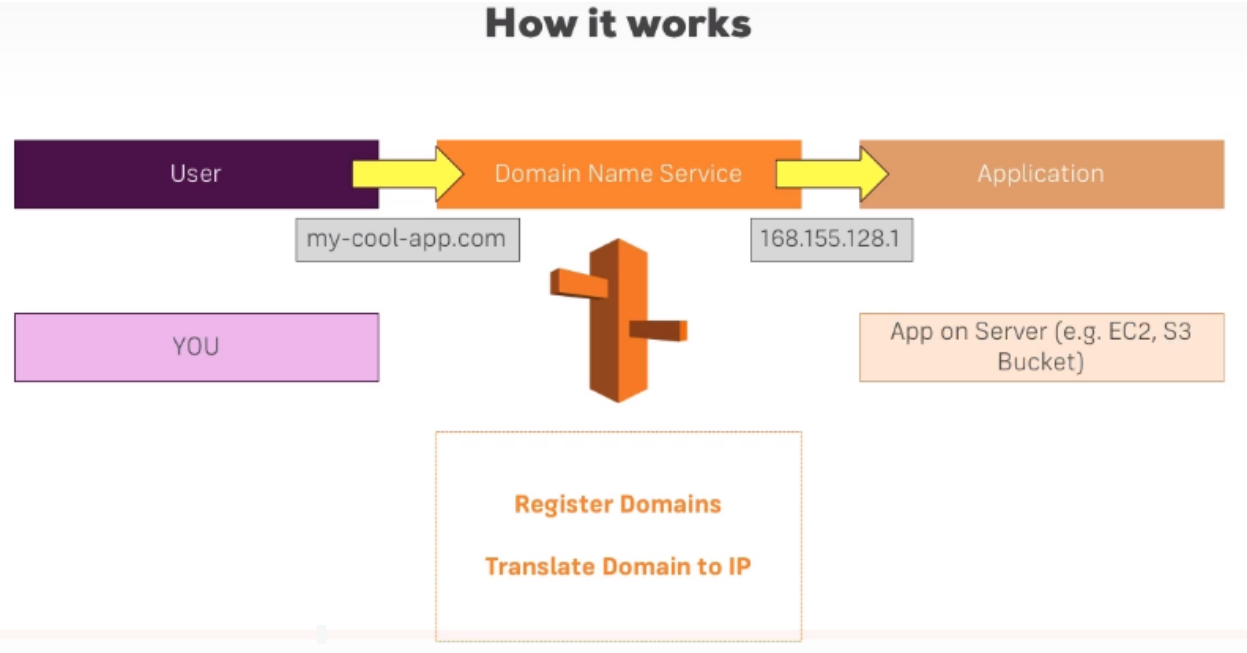


CloudFront

- Optimizing Content Delivery
- CloudFront use the CDN network (edge locations) to serve our page faster and more efficiently.
- With CloudFront, you will get a new domain name/URL which should be used instead of the direct S3 url.

Route 53

- It's AWS domain name service. There you can buy domains or use ones you already own and manage them.
- In the end, what you do on Route 53 is you want to enable that when your user visits mycoolapp.com, you in the end register domain, translate that domain to an IP and behind the scenes, use that ID to actually use your CloudFront or S3 distribution there.



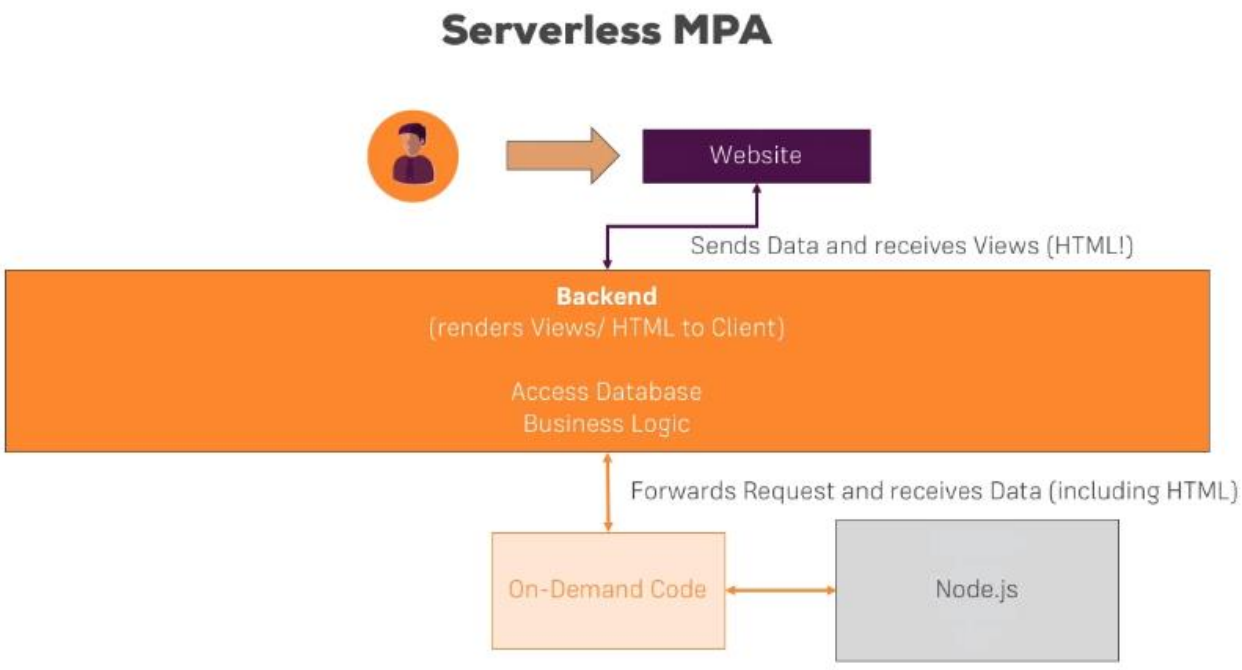
Beyond the Basics – An Outlook

AWS Lambda Triggers

- You create a function and you explicitly state which trigger will execute that function.
- There are many supported triggers. Now API Gateway is amongst them. Other like cloud watch events which would allow you to schedule events so that you execute the lambda function every x minutes for example or DynamoDB to react to changes in the database with lambda and then execute some code whenever something changes. Kinesis to listen to streams of incoming data which is very useful for big data analytics or S3 to listen to file changes, file uploads, file deletions in a specific S3 bucket.

Going Serverless with a Node/ Express App (Non-API!) – Multi Page App

- We can actually have a website where we don't just send data and get data through Ajax requests but where we send requests on each click basically and get back different HTML pages, different views.

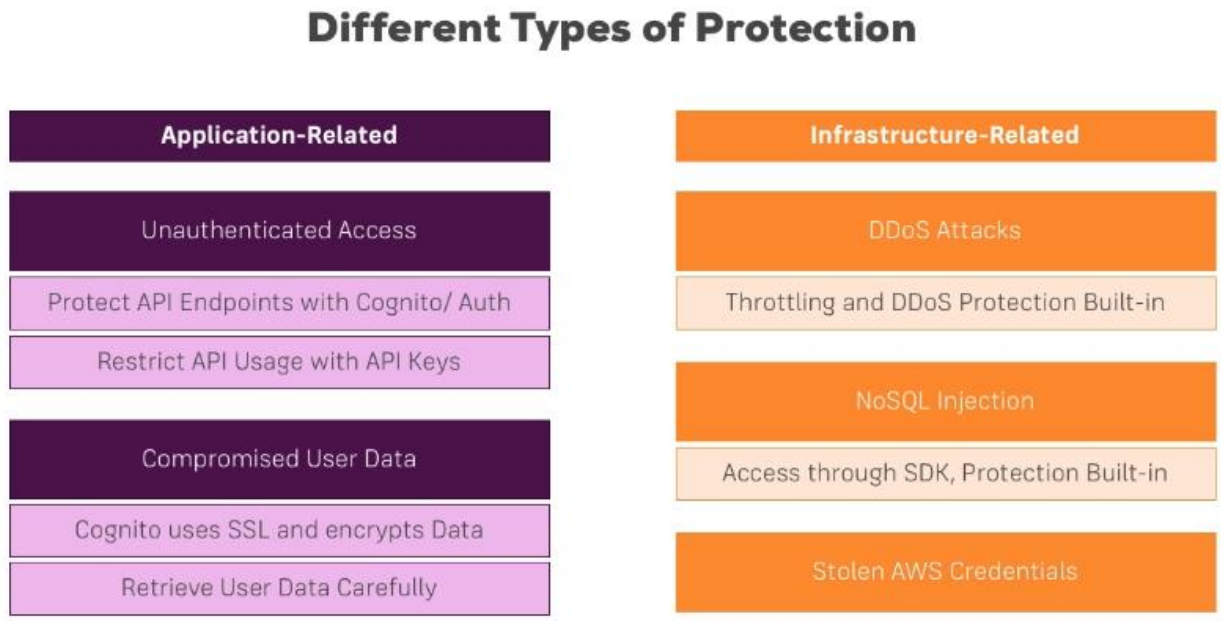


- aws-serverless-express is an npm package. This is a helper package which wraps your Express application which you can then serve on lambda to get your normal Express application handled by lambda.
- https://www.npmjs.com/package/aws-serverless-express

Disadvantages

- Difficult to import any files e.g. styles files used in your html because the browser would request it and would not request it through Express.js but directly from the server which doesn't work because remember, all incoming requests are caught and forwarded to Express. Solution is to inline the styles in the html file.
- Difficult to work with file related operations because your lambda function only runs in a context which is available for a couple of minutes and then is shut down by AWS again. It only runs on-demand and it isn't persistent, it's stateless, that makes working with files pretty much impossible.
- The same is true for sessions, you can only have stateless applications here because a session which persists for multiple minutes or hours is not possible here. Your lambda function is only executing for a maximum of five minutes and in general after it executed, the environment in which it ran will be shut down by AWS and a new one will be started for the next incoming request but there is no persistence between the two requests. So it is stateless, it doesn't behave the same as it does on a normal server where you run it and you have to be aware of that.
- In short, NOT RECOMMENDED!

Serverless Apps and Security



Serverless Framework

- https://www.serverless.com/
- Serverless Framework provides you a better development workflow where you don't have to manage it on the AWS console with clicking around, instead everything is managed via few commands from your local.
- Serverless Framework allows you to focus on your code rather than doing all that AWS managing.
- Important Links –
    - Serverless Framework Website: https://serverless.com/
    - AWS Getting Started Guide (with Serverless Framework): https://serverless.com/framework/docs/providers/aws/guide/quick-start/
    - Managing AWS Credentials (for using the Serverless Framework): https://serverless.com/framework/docs/providers/aws/guide/credentials/
    - Serverless Framework on Github: https://github.com/serverless/serverless

Serverless Application Model (SAM)

- If you don't want to use an extra framework like Serverless Framework or if you are a big fan of writing those config YAML files, there is an alternative to it – the official serverless application model published by AWS.
- This is a popular alternative to the serverless framework which gives you more configuration options and allows you to stay closely to the AWS world without using any other third-party tools.
- Official – https://github.com/aws/serverless-application-model

Testing Serverless Apps with localstack

- https://github.com/localstack/localstack
- https://localstack.cloud/

Other Useful AWS Services to use with Lambda