$ mvn package

$ mvn package shade:shade

^ create jar file ^

$ sls deploy –package /out/artifacts/jarFile

Serverless Course

* APP : Serve Static App 🡪 S3
* API : REST API 🡪 API Gateway
* Logic : Execute Code on Demand 🡪 Lambda
* Data : Store & Retrieve Data 🡪 DynamoDB
* Auth : Authenticate Users 🡪 Cognito
* DNS : Translate URL 🡪 Route 53
* Cache : Improve Performance 🡪 CloudFront

API Gateway

* to define API endpoints and HTTP methods
  + via resources
* authorize access
* directly access some AWS Services
* run Lambda Code (and forward Request Data) //trigger function
* “$” REST API 🡪 Build
* Request/ Response Cycle
  + Method Request 🡪 Integration Request 🡪 Integration Response 🡪 Method Response
  + in JSON format
  + integration configures response into proper format
  + can configure meta data or other values in Integration Response
  + configure headers in Method Response
* “$” Create New Resource (creates API endpoint)
  + API Gateway CORS
    - Cross-Origin Resource Sharing
    - having correct headers
    - check for API to automatically configure CORS between the 2 servers
  + create a HTTP method
    - select 1 of 7 methods (ex POST)
    - select Integration type
      * Lambda Function
        + runs code on demand
        + Use Lambda Proxy Integration

selecting will take incoming request and all the accompanying metadata and pass it unfiltered to the Lambda function in JSON format

then, in the Lambda function you will have to extract what you need and send response

this takes logic away from API gateway (done automatically) and brings it to the Lambda server

not recommended

* + - * + Lambda Region (pick same region as Lambda function ideally)
      * HTTP
      * Mock
      * AWS Service

AWS Lambda

* triggered from event source
  + S3 (file gets uploaded)
  + CloudWatch (scheduled)
  + API Gateway (HTTP Request) 🡨 Main
  + others
* Event Source 🡪 Code 🡪 Result
  + interacts with other AWS Services
  + return response
* a lambda function for each API endpoint
* How to Create Lambda Function
  + “$”Create function
  + “Author form scratch”
  + “Create a new role from template(s)”
    - enter a role name
    - can select a policy template (optional)
  + “Create function”
  + Lambda editor UI
    - write code
    - attach triggers
    - edit the general configuration
  + sync preferred over async
* shortcuts
  + Full Screen – cmd + shift + f
  + Cache file locally – cmd + s
  + save (UpdateFunctionCode) cmd + shift + u
  + test – cmd + i
  + configure test events – cmd + j
* Author / Test / Debug cycle
  + Author
    - code
    - application
  + Test
    - challenges
      * reproduce environment that resembles Lambda
        + OS
        + Libraries
        + Runtime
        + configured limits (memory, timeout)
      * mimic response and log outputs
      * test events need to be
        + syntactically accurate
        + different for each trigger
    - SAM Local – helps overcome these challenges
      * CLI tool for local testing of serverless apps
      * leverages Docker images to mimic Lambda’s execution environment
      * emulates Lambda functions and APIs
      * event generator to help you generate event payload for common Lambda triggers
      * response objects and function logs available on local machine
      * supports live debugging
      * open source and accepts pull requests
      * www.github.com/awslabs/aws-sam-local
  + Debug
* Serverless Application Model (SAM)
  + CloudFormation extension optimized for serverless
    - API gateways
  + new serverless resource types: functions, APIs, and tables
  + supports anything CloudFormation supports
  + open specification (Apache 2.0)
  + can define three resource types
    - API Gateway
    - Lambda Functions
    - DynamoDB tables
  + CloudFormation
    - provision and manage a collection of related AWS resources
    - your app = CloudFormation stack
    - input .yaml file and output provisioned AWS resources
    - similar to SAM but optimized for infrastructure, hence the need for SAM
    - resources provisioned by CF :
      * AWS::Lambda::Function
      * AWS::IAM::Role / Policy (2)
      * AWS::ApiGateway::RestApi/Stage/Deployment (3)
      * AWS::Lambda::Permission
  + AWS Cloud9
    - cloud-based dev environment
    - write, test, and debug with just a browser
    - optimized for serverless

AWS Lambda Function

* function code (bottom)
  + “$” Code entry type
    - edit code inline
    - upload .zip file
    - upload a file from Amazon S3
  + “$” Runtime – node.js (or Java)
  + “$” Handler - defines entry point
  + *default –*
  + $ exports.handler = async(event) =>{
    - const response = {
      * statusCode: 200,
      * body: JSON.stringify(‘Hello’),
    - };
    - return response
  + }:
  + *other option*
  + $ exports.handler = (event, context, callback) => {
    - callback(new Error(), {message: ‘Hi’});
  + };
  + “$” Handler – index.handler
* “$” Environment variables – to have variable in code
* “$” Tags – for validates and builds
* “$” Execution role – permissions, who is allowed access
* “$” Basic settings –
  + default timeout 3 seconds
  + memory(MB) – allows for faster running but costs more
* “$” Network – VPC
* Function Code Steps
  + create a root entry + handler method
    - JS ex- index.js $ exports.handler = async (event) => { …
    - set “$” Handler to [FILENAME].[HANDLER-FUNCTION-NAME]
      * ex index.handler
  + you may split your code over multiple files and import them into the root file
    - JS ex – require(‘file-path’)
  + select all files and zip (jar) them into an archive
    - important: don’t put them into a folder and then zip
  + upload the created zip/jar file to Lambda
    - (“Code” => “Code entry type” => “Upload a .ZIP file”)
* Deployment
  + must be staged “$” Stages
    - stages are snapshots
  + “$” Actions
    - Deploy API
      * create deployment stage
  + fix CORS error – need to set Headers for each method individually (even with CORs checked)
    - “$” Resources – “$” method(POST)
    - “$” Method Response
      * “$” Response Headers for 200
        + Access-Control-Allow-Origin
    - “$” Integration Response
      * “$” Mapping Value 🡪 ‘\*’
        + ‘\*’
* Body Mapping Templates
  + passing to Lambda only the information it needs
  + “$” Integration Request
    - “$” When there are no templates defined
      * “$” Content-Type –
        + application/json
        + $ (in-line code)
  + google Body Mapping Template Language for syntax
  + { “body-json” : $input.json(‘$’), } //returns all data
    - $ input = variable provided by AWS giving access to request data- body, params, etc
    - json(‘$’) = extracts complete request body
  + { “age” : $input.json(‘$.personData.age’) } //returns age only
  + or
  + #set($inputRoot = $input.path(‘$’))
  + { “description” : “$inputRoot.desc” } //add “” if a String

#set

$input.json(‘$’)

* Models
  + “$” Models
  + “$” Create
  + $ Model schema
    - JSON format
  + used to validate data
  + { “age”:$input.json(‘$.age’) } // requires that the data contain an age property
  + “$” Model Schema
  + {
    - “$schema”: “http://json-schema.org”,
    - “title”: “CompareData”,
    - “properties”: {
      * “age”: {“type”; “integer”},
      * “height”: {“type: “integer”},
      * “income”: {“type”: ”integer”}
    - },
    - “required”: [“age”, “height”, “income”]
  + }
  + **add model to “$” Request Body**
    - “$” Request Validator : Validate body
* Path Parameters
  + can create flexible GET path (single or multiple)
  + “$” New Resource
    - “$” Resource Path = {type}
  + “$” Configure function
    - Lambda function code
    - $
    - exports.handler = (event, context, callback) => **{**
      * const type = event.type;
      * if (type == ‘all’) {
      * etc
  + Body Mapping Template
    - application/json
    - {
      * “type”: “$input.params(‘type’)”
    - }

DynamoDB

* fully managed, cloud-based NoSQL db
* don’t have to spin up db servers
* just a couple Solid State Drives that store data
* Data format – Key/Value pairs, JSON
* always required – partition key
* partition key must be unique, and present on each item
  + the key is partitioned to different drives, hence the name
  + equivalent to primary key
  + Optional : partition key + sort key = primary key
* Partition Key, Dynamo supports 2 types
  + partition key: a simple primary key composed of one attribute
  + partition key + sort key: a composite primary key composed of two attributes
* DynamoDB stores and retrieves each item based on the primary key value
  + items are distributed across 10-GB storage units called partitions (physical storage internal to DynamoDB).
  + each table has one or more partitions
  + DDB uses the partition key’s value as an input to an internal hash function
  + the output from the hash function determines the partition in which the item is stored
  + all items with the same partition key are stored together, and for composite partition keys, are ordered by the sort key value
    - DDB splits partitions by sort key if collection outgrows 10GB
* Global Secondary Index – forces Dynamo to specially manage attribute
* Local Secondary Index
* NoSQL characteristics
  + no relations
  + high flexibility (no/weak schemas)
  + data repetition
  + no integrity checks
  + easy scalability
* DynamoDB + Lambda
  + DDB can be an event source to trigger Lambda
  + Lambda can store and retrieve data form DDB
* Creating a table
  + from main console in AWS- search Dynamo
  + “$” Create table
    - Table Name
    - Primary key
      * ex UserId – String
      * Sort Key (optional)
    - Secondary Index
      * advanced setting. TBL
* Table- 5 x 5 in free tier
  + all tables across all regions share the free tier
  + cannot exceed capacity
* Dashboard
  + Overview
    - stream details
    - TTL (time to live)
    - table details
  + Items
    - view and create items in a table
    - “$” Create item – requires Primary Key
    - “+” – variable type ex String
  + also Metrics, Indexes, Access Control, Tags
* Using Dynamo from Lambda
  + choose lambda function from AWS
  + AWS SDK (java)
    - npm install aws-sdk
      * only need to install on own machine if using from it (and not lambda)]
    - **read docs!**
    - need permissions
    - “$” Run –
      * need to import lambda function
      * can instantiate services to access methods
      * “$” API documentation
  + data will always be passed to DyanamoDB as a string
    - so you must specify the data type
  + $ putItem(), getItem(),
  + $ query() takes conditions into account while searching
    - scan() AFTER reading all items
  + by default, no service has any permissions
    - search IAM
    - view policy
      * default is just logging permissions so we need to attach another policy
      * AmazonDynamoDBFullAccessRole
  + Javascript example
    - dynamodb.scan(params, function(err, data){
      * if (err) etc etc
      * }else {
        + const items = data.Items.map(

(dataField) => {

return {age: parseInt(dataField.Age.N}

* Secondary Indexes
  + to perform queries on attributes that are not part of the table’s primary key
  + can define up to 5 each of the two types- local and global
  + Local
    - index is local to a partition key
    - allows you to query items with the same partition key
    - all items with a particular partition key in the table and the items in the corresponding local secondary index (aka item collection) are stored on same partition
    - can only be created when a table is created and can’t be deleted
  + Global
    - index is across all partition keys
    - allows you to query over the entire table, across partitions
    - can have a partition key and optional sort key that are different from the partition key and sort key of the original table
    - key values do not need to be unique
    - can be created/deleted at any time
    - only supports eventual consistency
* DynamoDB Streams
  + an ordered flow of information about changes to a table
  + the records are ordered by when the changes occurred
  + each change contains 1 stream record
  + available for 24 hours
* DynamoDB Accelerator (DAX)
  + DAX is a caching service that enables you to benefit from fast in-memory performance for demanding applications
  + reduces response times of eventually-consistent read workloads
  + reduce opertaoin and application complexity through a managed service that is API-compatible with Amaon DynamoDb
  + increase throughput for read-heavy or bursty workloads
* Monitoring and Troubleshooting Best Practices
  + check the AWS error code returned from your operations and include in application logs
  + enabled CloudTrail so that DynamoDB control operations are available for analysis
  + use the CloudWatch metrics provided to monitor table performance
  + set alarms for pertinent metrics out of acceptable range
* Basic Item Requests
  + Write
    - PutItem – write item to specified primary key (new/update)
    - UpdateItem – change attributes for item with spk (specified primary key)
    - BatchWriteItem – write bunch of items to the spks
    - DeleteItem – remote item associates with spk
  + Read
    - GetItem – retrieve item associate with specified primary key
    - BatchGetItem – retrieve items with this bunch of spks
    - Query – for specified partition key, retrieve items matching sort key expression
    - Scan – get all items in the table

**Lambda**

* permissions
  + requires IAM Resource Policy/Function Policy
    - permission to invoke the lambda function (triggering event)
    - Policy associated with a “push” event source
    - created when you add a trigger to a Lambda function
    - Allows the event source to take the lambda:InvokeFunction action
    - easy to modify within Lambda console
    - there is a size limit, must use IAM roles if too many
  + requires IAM Execution Role
    - permission to execute function (ex make change to DB)
    - IAM role is selected or created when you create Lambda function
    - *IAM policy* includes actions that can be taken with the resource
    - *Trust policy* allows Lambda to AssumeRole
    - Creator must have permission for iam:PassRole
    - requires additional configuration to use with VPC
  + IAM Resource Policy/Function Policy
* event sources
  + Push Events
    - synchronous
    - asynchronous
  + Polling Events
    - stream-based
    - not stream-based
    - Data delivered in batches
  + synchronous Push
    - must manage retry strategy within application code (in case of failure)
    - RequestResponse – to invoke Lambda synchronously
    - examples that invoke Lambda via sync
      * API Gateway, Cognito, CloudFormation, CloudFront
  + asynchronous Push
    - events are queued
    - should be used for batch processes
    - Lambda automatically retries the invoke twice (total of 3 tries)
    - dead-letter queue (DLQ) option
    - Error Handling
      * Maximum Event Age
      * Maximum Retry Attempts
    - Event – to invoke async
    - examples that invoke Lambda via async
      * S3, SNS, SES, CloudWatch Logs and Events, CodeCommit, Config
  + Notes
    - when invoking a Lambda function programmatically, you must specify the invocation type
    - when AWS services are the source, the invocation type if predetermined
    - Lambda console shows all event sources for a given function
    - when you select Test from the Lambda console, it invokes the function synchronously
  + Polling Events
    - used by DynamoDB Streams, SQS, and Kinesis Data
    - services put items into the stream or queue
    - AWS Lambda polls the stream or queue, and if it finds records, it will deliver the payload and invoke the Lambda function
    - the Lambda service pulls data from the stream/queue itself for processing (in this model)
    - Streams Error Handling
      * errors in a shard block further processing
      * a failure in this modle blocks Lambda from reading any new records from the stream until the failed batch of records expires or is processed successfully
      * the events in each shard of the stream need to be processed in order
    - Queue Error Handling
      * errors in a batch are returned to queue
      * Lambda will keep retrying a failed message until it is processed successfully or the retries or retention period are exceeded
      * if the message filas all retries, it will go to the DLQ (if configured) or be discarded
      * does NOT stop processing of the batch, but it may change the order in which messages are processed
* LifeCycle
  + 1. When a function is first invoked, an execution environment is launched and bootstrapped. once the environment is bootstrapped, your function code executes. then Lambda freezes the execution environment, expecting additional invocations
  + 2. If another invocation request for the function is made while the environment is in this state, that request goes through a warm start. Warm start- the available frozen container is “thawed” and immediately begins code execution (skips bootstrap)
  + 3. This thaw and freeze cycle continues as long as requests continue to come in consistently. If environment idle for too long, then it is recycled
  + 4. A subsequent request starts the lifecycle over- a Cold Start
  + Best Practices
    - 1. store and reference dependencies locally
    - 2. limit re-initialization of variables
    - 3. add code to check for and reuse existing conditions
    - 4. use tmp space as transient cache
    - 5. check that background processes have completed

Authoring Lambda Functions

* Handler Method –
  + entry point that AWS Lambda calls to start executing your Lambda function
  + always takes 2 objects – Event Object and Context Object
  + event object
    - provides information about the event that triggered the Lambda function
    - could be pre-defined object generated by an AWS Service, or a custom user-defined object in the form of a serializable string (ex pojo or json stream)
  + context object
    - generated by AWS
    - provides metadata about the execution
    - can be used to interact with Lambda
    - includes –
      * awsRequestId
        + identifyawsRequestID
      * logStreamName
      * getRemainingTimeInMillis()
      * invokedFunctionArn
      * clientContext
* Best Practices (design)
  + Separate Business Logic
    - separate your core business logic form the handler method
      * makes code more portable
      * better for unit tests
  + Write Modular Functions
    - create single purpose functions
      * same principles of microservices
  + Treat Functions as Stateless
    - no information about state should be saved within the context of the function itself
      * because your function only exists when there is work to be done
    - Options:
      * DynamoDB is serverless and scales horizontally to handle Lambda invocations
        + has single-millisecond latency which makes it great for storing state information
      * Amazon ElastiCache may be better if using a VPC (may be less expensive)
      * Amazon S3 can be used as an inexpensive way to store state data if throughput isn’t critical and the type of state data won’t change rapidly
  + Only Include What You Need
    - minimize both your deployment package’s dependencies and its size
      * can have significant impact on startup time
      * only choose the modules you need, not entire AWS SDK
    - reduce the time it takes Lambda to unpack deployment packages authored in Java
      * put your dependency .jar files in a separate /lib directory
    - opt for simpler Java dependency injection frameworks
* Best Practices (writing code)
  + Include Logging Statements
    - lambda functions should include logging statements which are written to CloudWatch
  + Include Results Info
    - functions must give Lambda information about the results of their execution
  + Use Environmental Variables
    - take advantage of environment variables for operational parameters
    - they allow you to pass updated configuration settings without changes to the code itself
    - can be used to store sensitive information required by the function
  + Avoid Recursive Code
    - avoid a situation where a function calls itself

Configuring Lambda Functions

* 3 core components
  + memory
    - how powerful/fast
    - allocates CPU power proportional to memory
  + timeout
    - maximum duration of a function
    - interdependent on memory
    - default – 3 seconds
  + concurrency
    - unit at which scaling is measured
    - default – 1000 concurrent executions
    - can set specific concurrent execution limits or reserves at the function level
    - can limit in order to handle costs
    - or ensure you can handle peak expected volume of a critical function

Troubleshooting with Lambda

* CloudWatch – default monitorer
  + most of the following are available in lambda dashboard
  + Invocations
    - number of times a function is called
  + Errors
  + Duration
    - elapsed during function execution
  + Throttles
    - due to invocation rates exceeding the customer’s concurrent limits
  + IteratorAge
    - for stream-based invocations
    - measures the age of the last record for each batch of records
      * age is difference between the time Lambda received the batch and the time the last record in the batch was written to the stream
  + ConcurrentExecutions
    - measures the sum of concurrent executions for a given function
    - is averaged if aggregated across a period of time
  + UnreservedConcurrentExecutions
* AWS X-Ray
  + use for performance tuning
  + identify call flow of Lambda functions and API calls
* AWS CloudTrail
  + logs API calls made by or on behalf of a function
  + audit actions made against application
  + integrate with a CloudWatch rule to respond to audit findings
  + export for additional analysis
* Dead Letter Queues
  + help capture app errors that you must respond to (and cannot discard)
  + use DLQs to analyze failures for follow-up or code corrections
  + available for asynchronous and non-stream polling events
  + can be an Amazon SNS topic or Amazon SQS queue
  + can be configured from Lambda console

SAM Template

* deployment in a serverless environment
  + the dev must provide everything needed to deploy a function- the code, dependencies, and the blue print for setting up the infrastructure
* use a serverless application framework to ease deployment and reduce work
* SAM is one popular example
* first 2 lines of following example tells AWS CloudFormation that this is a SAM template it needs to “transform”

AWSTemplateFormatVersion: ‘2020-10-24’

Transform: AWS::Serverless-2016-10-31

Resources: //\*creates a lambda function\*

GetHtmlFunction:

Type: AWS::Serverless::Function

Properties: //\*…with the code at the referenced location using the specified

CodeUri: s3://sam-demo-bucket/todo\_list.jar //\* … handler/runtime

Handler: handler.java

Runtime: java11

Policies: AmazonDynamoDBReadOnlyAccess //\* sets IAM policy

Events: //\* creates API Gateway endpoint and takes care of all mapping

GetHtml: //\* …and permissions necessary

Type: Api

Properties:

Path: /{proxy+}

Method: ANY

ListTable:

Type: AWS::Serverless::SimpleTable //\*creates a DynamoDB table

(with 5 read and write units)

Return IAM Policy

{

“Effect” : “Allow”,

“Action”: “execute-api”

}

**Authorization**

* Use with Lambda
  + left hand side (inside lambda function) – Authorizers
  + Custom Authorizer
    - return IAM Policy to grant/deny access
    - return Principle ID (User Id)
    - return Context Object
      * optional
      * in JSON format
  + Cognito User Pool
* Identity Token
  + allows access
  + defines where to get it from
    - Identity token source\*
    - ~~method.request.header.Authorization~~
    - UPDATE – ^ is now just - **Authorization**
  + Node example – can go to Policies and check {}JSON to see format of others
    - const token = event.authorizationToken
    - if (token === ‘allow’){
      * const policy = genPolicy(‘allow’, event.methodArn);
      * const principal-Id = ‘asdfqwerty’; //tbd
      * const context = {
        + simpleAuth: true
      * };
      * const response = {
        + principal-Id: principal-Id,
        + policyDocument: policy,
        + context : context
      * };
      * callback(null, response);
    - } else { etc}
    - function genPolicy(effect, resource){
      * const policy = {}
      * policy.Version = ‘2020-10-28’
      * policy.Statement = [];
      * const stmt = {}
      * stmt.Action = ‘execute-api:Invoke’;
      * stmt.Effect = effect;
      * stmt.Resource = resource;
      * policy.Statement.push(stmt);
      * return policy;
    - }
  + Expected input data when using custom authorization functions
* {
* "type":"TOKEN",
* "authorizationToken":"<caller-supplied-token>",
* "methodArn":"arn:aws:execute-api:<regionId>:<accountId>:<apiId>/<stage>/<method>/<resourcePath>"
* }
  + **<caller-supplied-token>** is the token you actually receive. you configure how to extra the token from the incoming request in API gateway
  + **methodArn** refers to the endpoint on which this authorizer was triggered
  + the following **output** data has to be provided by your function-
* {
* "principalId": "yyyyyyyy", // The principal user identification associated with the token sent by the client.
* "policyDocument": {
* "Version": "2012-10-17",
* "Statement": [
* {
* "Action": "execute-api:Invoke",
* "Effect": "Allow|Deny",
* "Resource": "arn:aws:execute-api:<regionId>:<accountId>:<appId>/<stage>/<httpVerb>/[<resource>/<httpVerb>/[...]]"
* }
* ]
* },
* "context": {
* "stringKey": "value",
* "numberKey": "1",
* "booleanKey": "true"
* }
* }
  + **principalId** is the user identifier
  + **policyDocument** is a JS object which uses the IAM policy structure
  + **context** is an optional object of key-value pairs
* “$” New Custom Authorizer (in API Gateway)
  + connect Lambda function\*
  + Execution Role – can assign role or API Gateway will automatically configure access for itself to execute the function if left blank
  + Identity token source- identifies how authorization information is extracted from the incoming request
    - method.request.header.Authorization
    - **Authorization** (now just the last word, I think)
    - API Gateway mapping expression
  + Token validation expression – a regular expression to automatically return requests which don’t fulfill the basic requirements, can be left blank
  + Result TTL in seconds – 30 (a fairly short time)
  + Identity token – *allow* – *deny* - to test
* Get User Id (principal Id) from Integration Request
  + Body Mapping Template
    - search API Gateway Body Mapping Template
    - $context.
    - $context.authorizer.principalId
* Full code-

#set($inputRoot = $input.path(‘$’))

{

“age”: $inputRoot.age,

“name”: “$inputRoot.name”,

“userId”: “$context.authorizer.principalId”

}

**AWS Cognito**

* Authentication
  + defines how to authenticate Users
  + stores Auth Token on User Devices
* User Data/ Auth Provider
  + Cognito User Pools
    - a **complete solution for those with no authentication process**
  + Third-Party Provider (like Google)
    - federated identities would allow you to connect third-party providers
* User Pools
  + user directories that provide sign-up and sign-in options for apps
  + manages the overhead of handling the tokens that are return from social sign-in
  + all members of the user pool have a directory profile that can be accessed
* Identity Pools
  + provide AWS credentials to grant users access to other AWS services
  + users can obtain temporary AWS credentials to access AWS services
  + supports anonymous guests and federation through third-party IdPs
* Creating a Cognito User Pool
  + Review defaults
    - gives a readily configured pool
  + Step through settings
    - to customize settings
    - add standard attributes desired
      * can add custom attributes
    - password strength
    - all options very straight-forward, just read carefully
    - add an app client
      * to connect it to a front end application
      * needs further configuration to create connection
      * “$” Generate client secret
        + must be disabled if your app is running in the browser
        + keep enabled if running from the backend
      * leave other 2 unchecked
* Authentication Flow (User Pools)
  + Sign Up, user is added to User Pool
    - in a database that is managed by Cognito
  + Confirmation Prompt
    - need to provide form to enter info
  + Sign In
    - user validated
    - gives 3 tokens (live for 1 hour)
      * identity token
      * access token
      * refresh token

AWS Lecture

* 10.0 (N) .0.0 (H) /16
  + Web – 10.0.1.0/24
  + App – 10.0.2.0/24
  + DB – 10.0.3.0/24
  + 10.0.\*.\* = 65.5K unique IPs
  + WR – 10.0.11.0/24
  + AR – 10.0.12.0/24
* Elastic Load balancer sends to the left or right (web or WR)
* IGW
* Amazon Virtual Private Cloud (VPC)

sonarqbe

other static and active analysis