**AWS**

**CloudFormation stack Configuration:**

1. Create a AWS KeyPair for SSH access in web console and download the data.
2. Create a new user then you will get private keys, use that credentials for configuring AWS CLI.
3. Then create a profile with that user credentials to continue.

**aws configure –profile demo**

Now you need to add all necessary credentials (Access keys)

1. Create a project and the keys downloaded, then add stack.yml with all required configurations like definition of EC2 instance and then security group definition for that EC2 instance.

* If want docker, need to add that specific definition.
* If want to use RDS, then need to add that specific definition.

1. Then create stack using the stack.yml created.

**aws cloudformation create-stack –stack-name sample –template-body** [file://$PWD/stack.yml –profile demo –region us-west-2](file://$PWD/stack.yml%20–profile%20demo%20–region%20us-west-2)

1. Then run the application required using Docker-compose.

**Elastic Beanstalk(EBS):**

1. For maven or gradle, make sure the application name is specified
2. Then build & run the project, you will get a jar file.
3. Next step is upload application to S3

aws s3 cp build/libs/beanstalk-deployment-1.0-SNAPSHOT.jar s3://{you bucket name}/beanstalk-deployment-1.0-SNAPSHOT.jar

1. Using EBS client, Try to get list machine images running and select one among them.

aws elasticbeanstalk list-available-solution-stacks |grep Java

1. Now start configuring cloudFormation script:

* **First basic info:**

"AWSTemplateFormatVersion": "2010-09-09",

"Description": "Spring Boot Cloudformation demo stack.",

* Now add parameters:

**Specify bucket containing application code:**

"Parameters" : {

  "SourceCodeBucket" : {

    "Type" : "String"

  }

}

**specify name of application:**

"SpringBootApplication": {

  "Type": "AWS::ElasticBeanstalk::Application",

  "Properties": {

    "Description":"Spring boot and elastic beanstalk"

  } }

**specify application version:**

"SpringBootApplicationVersion": {

  "Type": "AWS::ElasticBeanstalk::ApplicationVersion",

  "Properties": {

    "ApplicationName":{"Ref":"SpringBootApplication"},

    "SourceBundle": {

              "S3Bucket": {"Ref":"SourceCodeBucket"},

              "S3Key": "beanstalk-deployment-1.0-SNAPSHOT.jar"

    }

  }

}

**specify configuration template:**

"SpringBootBeanStalkConfigurationTemplate": {

  "Type": "AWS::ElasticBeanstalk::ConfigurationTemplate",

  "Properties": {

    "ApplicationName": {"Ref":"SpringBootApplication"},

    "Description":"A display of speed boot application",

    "OptionSettings": [

      {

        "Namespace": "aws:autoscaling:asg",

        "OptionName": "MinSize",

        "Value": "2"

      },

      {

        "Namespace": "aws:autoscaling:asg",

        "OptionName": "MaxSize",

        "Value": "2"

      },

      {

        "Namespace": "aws:elasticbeanstalk:environment",

        "OptionName": "EnvironmentType",

        "Value": "LoadBalanced"

      }

    ],

    "SolutionStackName": "64bit Amazon Linux 2016.09 v2.3.0 running Java 8"

  }

}

**Configure Environment properties:**

"SpringBootBeanstalkEnvironment": {

  "Type": "AWS::ElasticBeanstalk::Environment",

  "Properties": {

    "ApplicationName": {"Ref":"SpringBootApplication"},

    "EnvironmentName":"JavaBeanstalkEnvironment",

    "TemplateName": {"Ref":"SpringBootBeanStalkConfigurationTemplate"},

    "VersionLabel": {"Ref": "SpringBootApplicationVersion"}

  }

}

1. Now deploy the cloudFormation template:

**aws s3 cp beanstalkspring.template s3://{bucket with templates}/beanstalkspring.template**

1. Create stack using the above template from s3

**aws cloudformation create-stack --stack-name SpringBeanStalk --parameters ParameterKey=SourceCodeBucket,ParameterValue={bucket with code} --template-url https://s3.amazonaws.com/{bucket with templates}/beanstalkspring.template**

**AWS Fargate:**

[**AWS Fargate**](https://epsagon.com/blog/aws-fargate-the-future-of-serverless-containers/)**i**s an AWS managed service that is responsible for provisioning and orchestrating your containerized application. This means that you can deploy hundreds of containers without having to define any compute resources because the service will do it for you.

**Steps:**

* Build the application and build docker image.
* Then push the image to[**AWS ECR**](https://aws.amazon.com/ecr/)or Dockerhub.

Create login for AWS ECR, then push

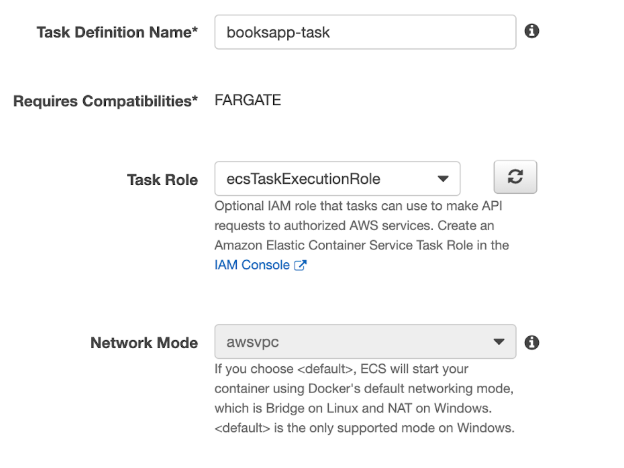
**$(aws ecr get-login --no-include-email --region us-east-1)**

**docker build -t jorlugaqui/booksapp . (already performed in previous steps)**

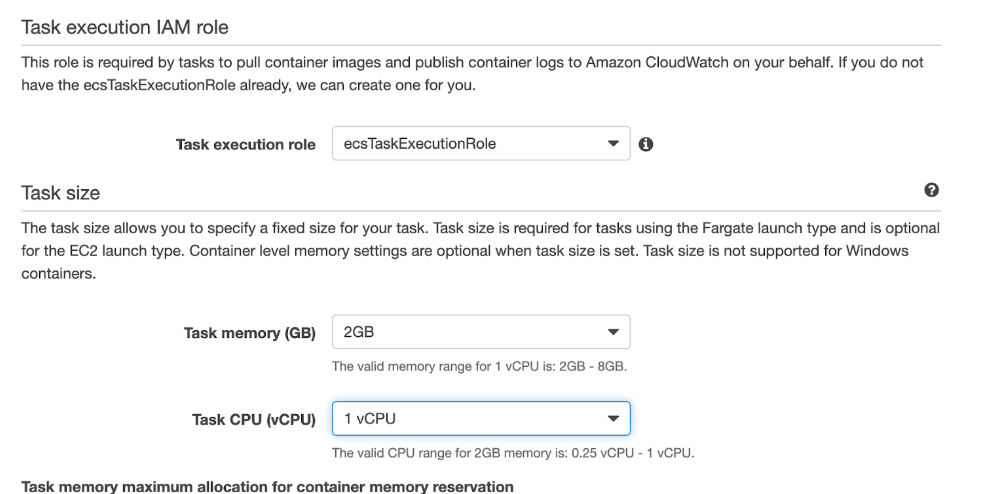
**docker tag booksapp:latest xxx.dkr.ecr.us-east-1.amazonaws.com/booksapp:latest**

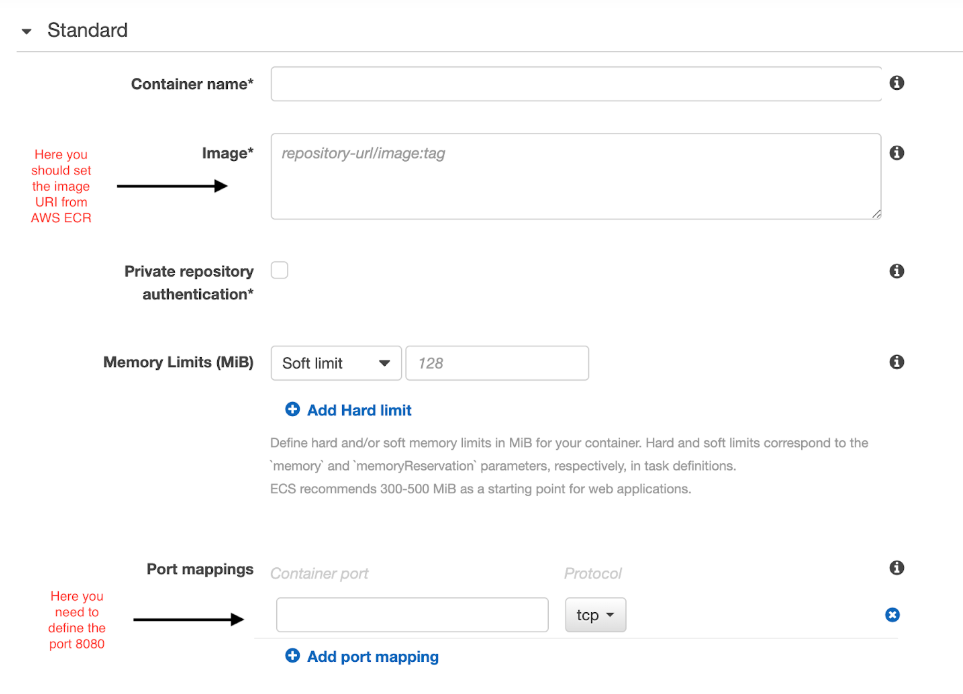
**docker push xxx.dkr.ecr.us-east-1.amazonaws.com/booksapp:latest**

* Then select compatability launch type as Fargate.
* Define a task in AWS ECS for defining a container.



Similarly, add Task EAM role, size and also container definition.





* Run the task on the default cluster or user-created cluster.

Add Task definition, version of platform, specific cluster, No.of Tasks & Task groups.

* Check if the application is working.

Use REST API

AWS CloudWatch

**Sample Templates:**

<https://raw.githubusercontent.com/jasonumiker/nginx-codebuild/master/fargate-cloudformation.template>

<https://github.com/1Strategy/fargate-cloudformation-example/blob/master/fargate.yaml>

**AMAZON DATABASE SERVICES**

**For AWS Database usage:**

1. First create AWS account
2. Then take necessary subscription to specific DBs.
3. Then choose way of usage either CLI or local or in AWS
4. Then create users and save the keys.
5. Add maven depencies and add necessary config classes.
6. Then add necessary spring annotations which DB specific.
7. Then run the application and test the service.

**DynamoDB**

Amazon DynamoDB is a fast and flexible NoSQL database service for all applications that need consistent, single-digit millisecond latency at any scale. It is a fully managed cloud database and supports both document and key-value store models.

**DynamoDB with Springboot :**

We can use DynamoDB locally by:

* Direct local installation
* Docker image
* Apache Maven Dependency

**Maven Dependencies**

<dependency>

  <groupId>org.springframework.boot</groupId>

  <artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

  <groupId>com.amazonaws</groupId>

  <artifactId>aws-java-sdk-dynamodb</artifactId>

  <version>1.11.34</version>

</dependency>

<dependency>

  <groupId>com.github.derjust</groupId>

  <artifactId>spring-data-dynamodb</artifactId>

  <version>4.5.0</version>

</dependency>

**application.properties:**

amazon.dynamodb.endpoint=<http://localhost:8000/>

amazon.aws.accesskey=key

amazon.aws.secretkey=key2

**create DynamoDBConfig class:**



**Model class:**

@DynamoDBTable(tableName = "Book")

public class Book {

    private String id;

    private String name;

    private String price;

    @DynamoDBHashKey

    @DynamoDBAutoGeneratedKey

    public String getId() {

        return id;

    }

    @DynamoDBAttribute

    public String getName() {

        return name;

    }

    @DynamoDBAttribute

    public String getPrice() {

        return price;

    }

}

**Repository Interface:**

@EnableScan

public interface BookRepository extends CrudRepository<Book, String> {

    List<Book> findById(String id);

}

**Testing spring application with DynamoDB:**

we want to test our application to ensure that we can connect and perform operations on our local DynamoDB:



**Elastic cache**

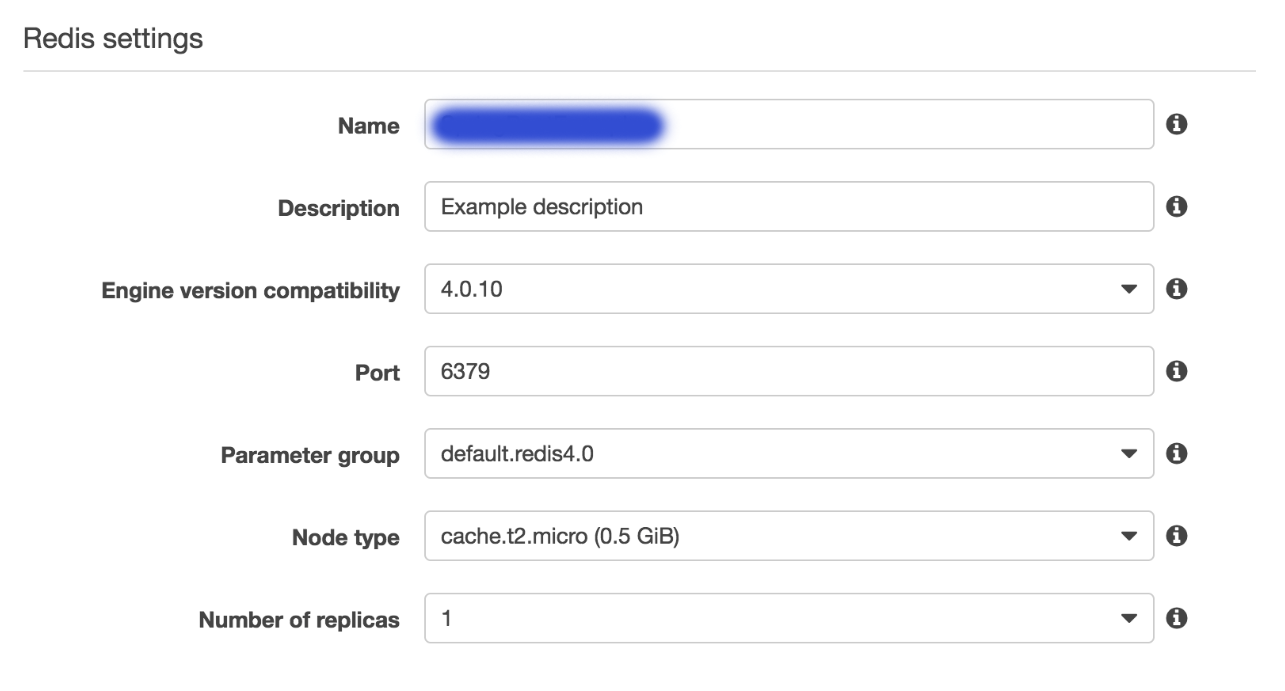
Amazon ElastiCache allows you to seamlessly set up, run, and scale popular open-Source compatible in-memory data stores in the cloud. Elasticcache has support for two open-source engines:

* Redis
* Memcached

**Elasticcache with Redis:**

[ElastiCache for Redis](https://aws.amazon.com/elasticache/redishttps:/aws.amazon.com/elasticache/redis/) is a super fast, in memory, key-value store database. ElastiCache for Redis is a fully managed service for a standard Redis installation and uses all the standard Redis APIs

1. Select Amazon Elasticcache cluster as Redis.
2. Make your own Redis shard(configure redis settings)

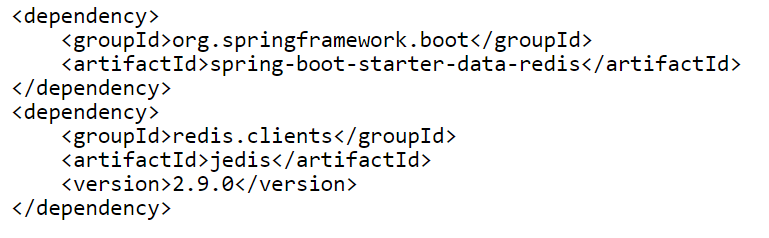


1. Add VPC(Virtual Private Network Configuration) & security group related settings
2. Now, Elasticcache will provision and launch the new Redis cluster.  When the status turns to available the cluster is ready to handle connections.



1. Now you can build your application.

**Maven Dependencies:**



**RedisElasticCache config file:**

@Configuration

@EnableCaching

public class RedisConfig {

    @Value("${redis.hostname}")

    private String redisHostName;

    @Value("${redis.port}")

    private int redisPort;

    @Value("${redis.prefix}")

    private String redisPrefix;

    @Bean

    JedisConnectionFactory jedisConnectionFactory() {

        RedisStandaloneConfiguration redisStandaloneConfiguration = new RedisStandaloneConfiguration(redisHostName, redisPort);

        return new JedisConnectionFactory(redisStandaloneConfiguration);

    }

    @Bean(value = "redisTemplate")

    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory redisConnectionFactory) {

        RedisTemplate<String, Object> redisTemplate = new RedisTemplate<>();

        redisTemplate.setConnectionFactory(redisConnectionFactory);

        return redisTemplate;

    }

    @Primary

    @Bean(name = "cacheManager") // Default cache manager is infinite

    public CacheManager cacheManager(RedisConnectionFactory redisConnectionFactory) {

        return RedisCacheManager.builder(redisConnectionFactory).cacheDefaults(RedisCacheConfiguration.defaultCacheConfig().prefixKeysWith(redisPrefix)).build();

    }

    @Bean(name = "cacheManager1Hour")

    public CacheManager cacheManager1Hour(RedisConnectionFactory redisConnectionFactory) {

        Duration expiration = Duration.ofHours(1);

        return RedisCacheManager.builder(redisConnectionFactory)

                .cacheDefaults(RedisCacheConfiguration.defaultCacheConfig().prefixKeysWith(redisPrefix).entryTtl(expiration)).build();

    }

}

**application.properties:**

redis.hostname=URL\_TO\_ELASTIC\_CACHE\_REDIS\_CLUSTER

redis.port=6379

redis.prefix=testing

**Implementing service:**

@Cacheable(value = "getLongRunningTaskResult", key="{#seconds}", cacheManager = "cacheManager1Hour")

public Optional<TaskDTO> getLongRunningTaskResult(long seconds) {

    try {

        long randomMultiplier = new Random().nextLong();

        long calculatedResult = randomMultiplier \* seconds;

        TaskDTO taskDTO = new TaskDTO();

        taskDTO.setCalculatedResult(calculatedResult);

        Thread.sleep(2000); // 2 Second Delay to Simulate Workload

        return Optional.of(taskDTO);

    } catch (InterruptedException e) {

        return Optional.of(null);

    }

}

**Testing:**

