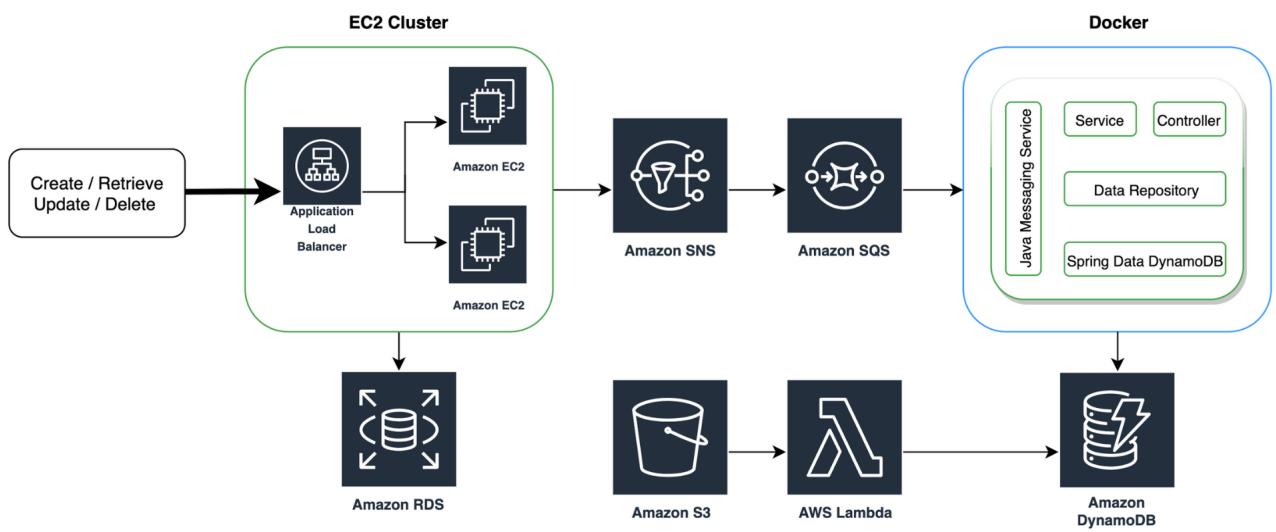




siecola.com.br

Desenvolvendo aplicações em Java para AWS



Desenvolvendo aplicações em Java para AWS

Paulo Cesar Siecola

Esse livro está à venda em <http://leanpub.com/amazonwebservice>

Essa versão foi publicada em 2019-09-09



Esse é um livro [Leanpub](#). A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. [Publicação Lean](#) é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.

© 2018 - 2019 Paulo Cesar Siecola

Para Joana e Rogério, meus dedicados pais. Um agradecimento especial para Isabel Mendes, minha esposa, pela excelente revisão técnica.

Conteúdo

1 - Introdução	1
1.1 - A quem se destina esse livro	1
1.2 - O que é AWS	1
1.3 - Serviços e recursos da AWS	2
1.4 - Estrutura didática do livro	8
1.5 - O que será necessário	8
1.6 - Limitações de recursos AWS pelo tipo de conta	9
1.7 - Próximos passos	9
2 - Preparação do ambiente	10
2.1 - Console da AWS	10
2.2 - Ambiente de desenvolvimento	12
2.3) Docker	13
2.4 - Postman	14
2.5 - Conclusão	15
3 - Conteúdo do livro	16
4 - Contas e regiões	26
4.1 - Identity and Access Management	26
4.2 - Regiões	34
4.3 - Zonas de disponibilidades	35
4.4 - Configuração do AWS CLI	36
4.4 - Conclusão	37
5 - Conceitos de Spring Boot	38
5.1 - Criando a base do projeto aws_project01	39
5.2 - Abrindo projeto no IntelliJ IDEA	40
5.3 - Estrutura do projeto	42
5.4 - Executando a aplicação pela primeira vez	44
5.5 - Gerando logs	46
5.6 - Criando um controller para expor um endpoint	47
5.7 - Depurando a aplicação	54
5.8 - Configurando a aplicação	56
5.9 - Criando serviços gerenciáveis pelo Spring	58

CONTEÚDO

5.10 - Conclusão	60
6 - Amazon Elastic Compute Cloud	61
6.1 - Conceitos básicos do EC2	61
6.2 - Unidades de CPU e memória	63
6.3 - Tipos de instâncias de EC2	63
6.4 - Criando um instância EC2	64
6.5 - Acessando a instância via SSH	66
6.6 - Instalando uma aplicação numa instância EC2	69
6.7 - Monitorando a instância EC2	76
6.8 - Estados da instância	77
6.9 - Criando imagens e templates customizados para EC2	77
6.11 - Balanceador de carga	81
6.12 - Conclusão	88
7 - Criando recursos AWS com CloudFormation	89
7.1 - O que são templates	90
7.2 - Estrutura básica de um template	91
7.3 - O que são stacks	92
7.4 - Construindo um template para criar uma instância EC2	94
7.5 - Criando e gerenciando stacks	101
7.6 - Atualizando uma stack	111
7.7 - Apagando uma stack	113
7.8 - Conclusão	114
8 - Executando aplicações em containers Docker	115
8.1 - Como funcionam containers Docker	116
8.2 - Criando imagens Docker	118
8.3 - Executando aplicações em containers Docker	124
8.4 - Docker Hub	129
8.5 - Conclusão	130
9 - Amazon Elastic Container Service	131
9.1 - Cluster	132
9.2 - Tarefas	136
9.3 - Definição de serviços	149
9.4 - Aumentando o número de instâncias do cluster	160
9.5 - Atualizando o serviço com uma nova definição de tarefa	160
9.6 - Conclusão	164
10 - Amazon CloudWatch	165
10.1 - Visualizando logs da aplicação	166
10.2 - Visualizando métricas de um serviço de um cluster	168
10.3 - Criando alarmes	169

CONTEÚDO

10.4 - Criando e monitorando eventos	175
10.5 - Conclusão	179
11 - Amazon Relational Database Service	180
11.1 - Criando um banco de dados MySQL	181
11.2 - Configurando a aplicação para acessar o banco de dados MySQL	188
11.3 - Persistindo entidades no banco de dados	190
11.4 Consultas avançadas	197
11.5 - Criando uma nova versão da aplicação	198
11.5 - Configurando o container da aplicação no ECS	199
11.6 - Monitorando a instância do banco de dados	202
11.7 - Conclusão	203
12 - Amazon Simple Notification Service	204
12.1 - Criando um tópico SNS	206
12.2 - Inscrevendo um e-mail para receber notificações	207
12.3 - Enviando notificações via SNS	208
12.4 - Atribuindo a permissão de acesso ao SNS ao papel ecsTaskExecutionRole	209
12.4 - Conclusão	222
13 - Amazon Simple Queue Service	223
13.1 - Porque usar um SQS	224
13.2 - Criando uma nova aplicação	225
13.3 - Configurando o projeto com JMS	226
13.4 - Consumindo mensagens de um SQS utilizando JMS	229
13.5 - Criando uma nova fila no AWS SQS	232
13.6 - Executando a aplicação no cluster-01 do ECS	239
13.7 - Testando o consumidor de mensagens da fila	241
13.8 - Monitorando SQS	242
13.9 - Conclusão	244
14 - Amazon DynamoDB	245
14.1 - Criando uma tabela no DynamoDB	246
14.2 - Configurando a aplicação para acessar a tabela	249
14.3 - Salvando entidades na tabela	253
14.4 - Visualizando os dados tabela do DynamoDB	255
14.5 - Testando e monitorando a tabela do DynamoDB	256
14.6 - Conclusão	261
15 - Amazon Simple Storage Service	263
15.1 - Criando a infraestrutura	264
15.2 - Configurando o projeto para acessar o S3 e o SQS	269
15.3 - Criando uma URL para upload de um arquivo	271
15.4 - Recebendo a notificação do S3 pela fila	273

CONTEÚDO

15.5 - Visualizando as notas fiscais	279
15.6 - Publicando a nova versão da aplicação aws_project01	279
15.7 - Testando a importação de arquivos de nota fiscal:	280
15.8 - Conclusão	282
16 - Amazon Lambda	284
16.1 - Agendando a execução de uma função Lambda	285
16.2 - Invocando uma função Lambda através de uma notificação SNS	293
16.3 - Invocando uma função Lambda através de uma notificação do S3	296
16.4 - Monitorando a execução de funções Lambdas	305
16.5 - Conclusão	307

1 - Introdução

Bem vindo ao livro **Desenvolvendo aplicações em Java para AWS!** Nele será ensinado como desenvolver aplicações que serão executadas no ambiente Amazon Web Services. Além disso, o leitor entenderá como funcionam diversos serviços de *cloud computing* e suas interações através de aplicações que serão construídas ao longo dos capítulos.

O leitor que decidiu seguir esse livro já deve saber da relevância que o desenvolvimento de aplicações utilizando *cloud computing* tem nas diversas áreas de tecnologia e negócios. Muito mais do que uma tendência, esse tipo de abordagem, para criação de sistemas e aplicações de software, tornou-se algo necessário para várias áreas de negócio. Atualmente poucos segmentos ainda utilizam infraestrutura própria para hospedagem de suas aplicações, talvez ainda motivados por insegurança e desconhecimento na infraestrutura de *cloud computing*.

Para acompanhar esse livro, é necessário que o leitor tenha conhecimento prévio de desenvolvimento de aplicações utilizando a linguagem **Java e programação orientada a objetos**. Conhecimentos específicos da linguagem, tais como *frameworks* e ferramentas de desenvolvimento serão apresentados ao longo do livro.

As demais linguagens que aparecerem serão utilizadas minimamente para construção de *scripts* ou arquivos de configuração, necessários para construção das aplicações e configuração dos serviços que serão criados.

As aplicações ensinadas nesse livro, como forma de exemplificação dos conceitos que serão apresentados, utilizarão *frameworks* e ferramentas modernas, como por exemplo **Spring Boot** e **Docker**. Dessa forma, o leitor aprenderá a trabalhar com tais tecnologias em conjunto com os serviços de *cloud computing*.

1.1 - A quem se destina esse livro

Esse livro é destinado a **desenvolvedores de software**, com ou sem experiência em *cloud computing* e desenvolvimento Web, que desejam se aprofundar nos serviços oferecidos pelo Amazon Web Services.

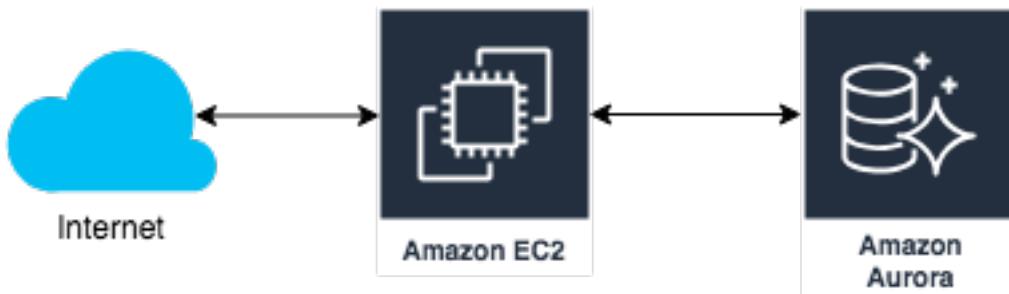
Também pode servir como uma fonte de conhecimento para **administradores e operadores de sistemas**, como forma de se familiarizarem com as ferramentas de criação e administração de recursos de *cloud computing* da Amazon Web Services.

1.2 - O que é AWS

AWS é a sigla para Amazon Web Service, a plataforma de *cloud computing* da Amazon. Nela é possível criar aplicações para diversas finalidades, hospedadas numa infraestrutura robusta e

escalável, utilizando serviços como banco de dados, servidores, dispositivos de armazenamento de dados e muito mais.

Um exemplo típico do que se pode fazer utilizando AWS, é criar uma máquina virtual para executar uma aplicação e conectá-la a um banco de dados para armazenamento de informações, como mostra a figura a seguir:



Exemplo de aplicação com banco de dados

Os dois recursos mostrados na figura são serviços da AWS e serão detalhados a seguir, mas eles representam uma instância de um máquinas virtuais (Amazon EC2), onde a aplicação é executada, e um serviço para hospedagem de banco de dados (Aurora), para armazenamento das informações da aplicação.

Até esse ponto não há nada de novo, porém esses recursos (a máquina virtual e o servidor do banco de dados), podem ser ligados ou desligados quando necessário, além de serem alterados em função de processamento e armazenamento à medida que a demanda da aplicação cresce ou diminui. Com isso, o custo de utilização de tais recursos pode ser ajustado de acordo com o número de requisições que a aplicação recebe.

Essa característica de utilização sob demanda de recursos de *cloud computing* é uma das grandes vantagens dessa tecnologia. Os custos de utilização dos recursos e serviços acompanham, de certa forma, a variação de sua utilização, o que torna seu uso interessante do ponto de vista de negócios e financeiramente mais viável.

Os recursos podem ser acessados diretamente da Internet ou protegidos com **regras de segurança** baseadas em diversos parâmetros de configuração, como por exemplo somente serem acessíveis através de outros recursos da AWS, ou seja, totalmente bloqueados do “mundo externo”.



Segurança de acesso a recursos e informações é algo fundamental em aplicações de *cloud computing*, e uma característica fortemente presente nos serviços oferecidos pela AWS.

1.3 - Serviços e recursos da AWS

Nessa sessão serão apresentados alguns dos **serviços e recursos oferecidos pela AWS**, principalmente os que serão abordados nesse livro.

O intuito é apresentar brevemente alguns serviços com suas nomenclaturas e símbolos, que serão utilizados nos diagramas apresentados ao longo do livro, para que o leitor se acostume quando eles aparecerem.

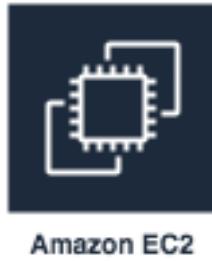
Todos os recursos apresentados aqui serão abordados posteriormente no livro, ao longo de seus capítulos. Porém, é possível que, por questões didáticas, alguns recursos sejam citados antes de serem formalmente apresentados, logo, é importante uma breve introdução a eles.

a) Amazon Elastic Compute Cloud - EC2

Talvez esse seja o serviço mais conhecido que a AWS provê. Com ele é possível criar **instâncias de máquinas virtuais** com capacidade de processamento, memória e armazenamento variados.

Durante o processo de criação de uma instância EC2, é possível escolher o sistema operacional que irá executar na máquina virtual. Com isso, após sua criação, a máquina virtual ficará pronta para ser utilizada.

O símbolo utilizado para representar uma instância EC2 em diagramas é o seguinte:



Amazon EC2

AWS EC2

Com uma instância EC2, é possível, por exemplo, instalar um servidor e hospedar uma aplicação Web disponível para toda a Internet. Também é possível executar uma aplicação dentro de um *container*, utilizando tecnologias como o Docker, que será utilizado e explicado ao longo dos capítulos desse livro).

b) Amazon Elastic Container Service - ECS

Com o ECS é possível executar aplicações dentro de ***containers*, como o Docker**. Dessa forma, não é necessário cuidar da administração de instâncias de máquinas virtuais EC2, que hospedariam tais aplicações.

Dentro de uma ECS, o administrador do sistema pode criar tarefas para executar várias máquinas virtuais, fazendo que a mesma aplicação tenha várias instâncias em execução ao mesmo tempo. Dessa forma, pode-se fazer um balanceamento de carga de processamento e tratamento de requisições entre as várias instâncias que estão rodando a aplicação. E ainda, é possível aumentar ou diminuir a quantidade de instâncias, automaticamente, de acordo com tais parâmetros de processamento e quantidade de requisições.

O símbolo que representa um ECS é o seguinte:



AWS ECS

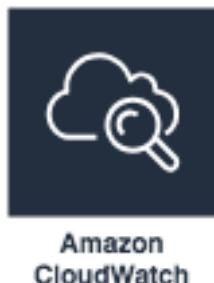
Executar aplicações dentro de *containers* traz muitas vantagens, principalmente no que se refere à orquestração da quantidade de instâncias da aplicação que devem ser executadas e para onde o tráfego deve ser direcionado, em caso de grande volume de requisições ou processamento. Esse é um assunto que será tratado nesse livro no momento que o conceito de ECS for explorado com mais detalhes.

c) Amazon CloudWatch

O CloudWatch é um serviço da Amazon Web Service de monitoramento e gerenciamento. Algumas de suas funções mais comuns são:

- Visualização de logs gerados por outros recursos e aplicações;
- Gráficos de **métricas** como consumo de memória e processamento;
- Visualização de eventos.

O símbolo que representa o CloudWatch é o seguinte:



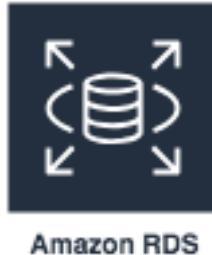
Amazon CloudWatch

Em um sistema com aplicações distribuídas em várias instâncias ou com diferentes tipos de recursos, é importante ter uma forma de concentrar essas informações em um único lugar, para facilitar seu monitoramento.

d) Amazon Relational Database Service - RDS

Este é um serviço para criação de bancos de dados relacionais na AWS, como Aurora, MySQL, PostgreSQL, Oracle e SQL Server, sem a necessidade da administração e operação de servidores.

O símbolo que representa o RDS é o seguinte:



Amazon RDS

O RDS também oferece benefícios como:

- Agendamento de backups;
- Recuperação de dados;
- Réplica de dados;
- Monitoramento e métricas.

e) Amazon DynamoDB

O Amazon DynamoDB é um banco de dados baseado em **documentos e estruturas de chave-valor**. Bem diferente de um banco de dados relacional, em termos de estrutura de armazenamento de dados, esse serviço possui operações de acesso extremamente rápidas em qualquer escala.

O símbolo que representa o Amazon DynamoDB é o seguinte:



Amazon
DynamoDB

AWS DynamoDB

Algumas de suas características são:

- Agendamento de backups;

- Recuperação de dados;
- Mecanismo que possibilita avisar a aplicação em caso de mudança de registros;
- Alta performance para operações de escrita e leitura;
- Mecanismo para exclusão automática de registros.

f) Amazon Simple Queue Service - SQS

O SQS é o **mecanismo de filas para troca de mensagens** da AWS, para, por exemplo, o estabelecimento de comunicação entre aplicações diferentes.

O símbolo que representa o Amazon SQS é o seguinte:



AWS SQS

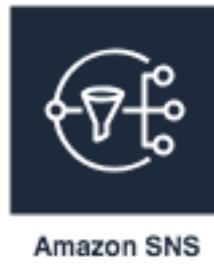
A seguir, algumas das características do SQS:

- Para começar, é necessário conectar o IntelliJ IDEA ao Docker da máquina de desenvolvimento. Isso pode ser feito acessando suas configurações, no menu Preferences\Settings -> Build, Execution, Deployment, na seção Docker, como pode ser visto na figura a seguir:
- Garantia de entrega de pelo menos uma mensagem;
- Possui mecanismos para evitar inversão de ordem de mensagens.

g) Amazon Simple Notification Service - SNS

Com o AWS SNS é possível enviar mensagens de **notificações** para dispositivos móveis, como Android e iOS, além de disparar eventos para outros recursos AWS. Ele também pode trabalhar de forma integrada com o AWS SQS, para permitir a propagação de mensagens de uma aplicação para várias outras, ao mesmo tempo.

O símbolo que representa o Amazon SNS é o seguinte:



Amazon SNS

AWS SNS

A seguir, algumas de suas características:

- Permite a integração com outros recursos AWS, como SQS, Lambda e S3;
- Vários SQS podem se registrar num SNS para recebimento das notificações;
- Independente de protocolo de mensagens.

h) Amazon Simple Storage Service - S3

O S3 da AWS é um dos seus serviços de **armazenamento de objetos em massa**. Ele é especialmente utilizado para armazenamento de arquivos, de forma temporária ou não, com alta disponibilidade de acesso.

O símbolo que representa do AWS S3 é o seguinte:



Amazon S3

AWS S3

A seguir, algumas de suas características:

- Alta disponibilidade;
- Permite notificar outros recursos AWS quando um objeto novo é inserido;
- Possui configurações de segurança para restringir o acesso público ou privado.

i) AWS Lambda

O Lambda é o serviço para **execução de código da AWS sem a utilização de servidores**, ou como é essa técnica conhecida: *serverless*.

Com ele é possível reagir a notificações ou eventos e executar um pequeno trecho de código de forma rápida, sem a necessidade de manter uma infraestrutura de servidores ou instâncias de máquinas virtuais ociosas.

O símbolo que o representa é o seguinte:



Algumas das principais características do AWS Lambda são:

- Não é necessário criar servidores para execução do código;
- Várias instâncias podem ser executadas para atender qualquer demanda de requisições;
- O custo de sua utilização é cobrado por tempo de execução de código, ou seja, somente quando é necessário executar algo.



Existem muitos outros recursos e serviços que são oferecidos pelo AWS. Para ter uma lista atualizada, consulte o seguinte link: <https://aws.amazon.com>

1.4 - Estrutura didática do livro

A estrutura didática desse livro permeia um conceito conhecido como **aprendizado baseado em problemas**, onde o leitor é apresentado e conduzido aos conceitos chaves através de problemas que devem ser resolvidos utilizando tais conhecimentos.

Obviamente, nem todos os conceitos podem ser apresentados dessa forma, mas para aqueles que permitem, será utilizado uma abordagem mais prática de aprendizado, fazendo com que o leitor sempre tenha em mente um problema a ser resolvido utilizando a tecnologia que é detalhadamente apresentada. Dessa forma, tendo-se sempre em mente o problema alvo a ser resolvido, o leitor pode absorver os conceitos que são apresentados de forma mais eficiente.

1.5 - O que será necessário

O próximo capítulo detalha tudo o que o leitor precisará para acompanhar esse livro. Aqui segue um resumo para que possa ser percebido que nada de especial ou com custo será utilizado:

- Computador com Windows, OS X ou Linux;
- Ferramentas de desenvolvimento gratuitas;
- Conta de usuário na AWS, que pode ser criadas de várias formas para um período gratuito de avaliação;
- Repositório de código no GitHub.

A ferramenta Postman será amplamente utilizada nesse livro, para acessar serviços REST que serão criados. É importante que o leitor tenha familiaridade com tal ferramenta.

1.6 - Limitações de recursos AWS pelo tipo de conta

Existem certas restrições de recursos que a AWS impõe, dependendo do tipo de conta utilizada para acessá-la. Um exemplo disso é o programa [AWS Educate](#)¹, que oferece créditos para estudantes e professores, mas que limita alguns recursos ou parte deles.

Essas restrições estão em constante revisão pela AWS, por isso verifique as que são impostas ao tipo de conta escolhido.

1.7 - Próximos passos

Nesse capítulo o leitor teve uma breve introdução do que é AWS, o serviço de *cloud computing* da Amazon, e alguns dos recursos que ela oferece. Além disso, foi apresentado como o livro está estruturado.

No próximo capítulo o leitor verá o que deve fazer para preparar seu ambiente de desenvolvimento e também os primeiros passos que deverão ser dados no console de administração da AWS para que ele possa começar a criar os recursos e desenvolver as aplicações.

¹<https://aws.amazon.com/education/awseducate/>

2 - Preparação do ambiente

Este capítulo descreve tudo o que deverá ser instalado ou preparado para acompanhar esse livro:

- Conta na AWS;
- Ambiente de desenvolvimento com o IntelliJ IDEA
- Outras ferramentas, como o Postman, que é um cliente REST.

2.1 - Console da AWS

A Amazon possui um console de administração para se utilizar a AWS, para criar, gerenciar recursos e posteriormente executar aplicações.

Esse console pode ser acessado no seguinte endereço: <http://console.aws.amazon.com/>

a) Tipos de contas na Amazon Web Services

Para acessar a AWS, através de seu console, é necessário criar uma conta. Dessa forma, será possível controlar a utilização de recursos, gastos (quando aplicáveis), monitorar alarmes de utilização, dentre várias outras funcionalidades.

A AWS possui alguns tipos de contas descritas logo a seguir. Qualquer uma dessas contas pode ser utilizadas para o acompanhamento desse livro:

- **Free tier:** talvez essa seja a forma mais simples de começar a utilizar a AWS para fins de avaliação e aprendizado. Com uma conta desse tipo é possível ter 12 meses gratuitos para utilizar alguns dos recursos da AWS, dentro de certos limites.

Constantemente a AWS renova os recursos e os limites desse tipo de conta. Para ter uma visão atualizada sobre essa modalidade, acesse o seguinte link: <https://aws.amazon.com/free>

Essa modalidade requer o uso de um cartão de crédito para a criação da conta, mas sem cobranças caso a utilização fique dentro dos limites estabelecidos nos critérios do *free tier* da AWS.

- **AWS Educate:** essa é uma opção ideal para professores e alunos de escolas, faculdades e universidades, pois oferece várias modalidades de créditos mensais para os usuários, sem a necessidade de utilização de cartões de créditos.

Para utilizar essa modalidade, é necessário que a instituição de ensino possua um cadastro na AWS. Para maiores informações, acesse o seguinte link: <https://aws.amazon.com/education/awseducate/>

- **Cupons com créditos:** apesar de não ser necessariamente um tipo diferente de conta, é possível obter cupons com créditos para serem utilizados na AWS. Tais cupons são distribuídos em eventos promovidos pela Amazon. Com esses créditos, é possível utilizar os recursos da AWS, até que eles acabem.
- **Conta paga:** esse é o modelo de conta tradicional da AWS, onde empresas e usuários pagam pelos recursos que utilizam na AWS mensalmente.

b) Criação da conta na AWS

Para aproveitar totalmente o conteúdo desse livro é necessário que o leitor crie uma conta na AWS, utilizando uma das opções citadas na seção anterior.

Tendo essa conta criada, independente da modalidade, será possível acompanhar todos os capítulos seguintes, criando e utilizando os recursos da AWS.

Para prosseguir com a criação de sua conta, acesse o seguinte link: <http://console.aws.amazon.com/>. Nessa tela, clique no botão *Create a new AWS account* e siga os passos que se apresentarem.

Depois que a conta for criada, será possível acessar o console de administração da AWS. Perceba que ele é uma interface Web rica em detalhes e opções ao usuário. À medida em que os conceitos forem apresentados ao longo dos capítulos, será possível aprender um pouco mais sobre as suas opções.

c) Usuário IAM

A conta que foi criada na seção anterior funciona com um usuário administrador, dentro do console da AWS. Com ela já é possível criar e utilizar os recursos da AWS, porém, é recomendada a criação de usuários sob o domínio dessa conta. Dessa forma, é possível ter um controle maior, através de grupos e permissões, do que cada usuário pode criar ou acessar dentre os recursos da AWS.

Tais usuários podem ser criados utilizando um recurso gratuito da AWS chamado *Identity and Access Management*. O capítulo 4 trará maiores detalhes sobre esses usuários, principalmente como criá-los e atribuir permissões a eles.



Mesmo que a conta criada seja específica para acompanhamento desse livro, é altamente recomendado que o leitor opte pela criação de um usuário IAM para utilização dos recursos da AWS, conforme será detalhado no capítulo 4.

d) AWS CLI

É possível criar e gerenciar os recursos da AWS através de uma aplicação, que pode ser instalada no computador, chamada *AWS Command Line Interface*.

A princípio, tudo o que pode ser feito no console Web da AWS pode ser feito no CLI, porém através de comandos com uma sintaxe específica.

O capítulo 4 trará os passos necessários para a configuração do AWS CLI, após o usuário IAM ser criado, juntamente com sua chave de acesso. Esses passos são necessários para que o AWS CLI possa ser ativado no computador que será utilizado para o desenvolvimento das aplicações apresentadas nesse livro.

2.2 - Ambiente de desenvolvimento

Para que o leitor possa acompanhar os capítulos desse livro, em sua totalidade, é necessário preparar seu ambiente de desenvolvimento, instalando algumas ferramentas e principalmente a IDE a ser utilizada para criação das aplicações de exemplo.

É importante observar que os passos descritos a seguir estão em linhas gerais do que deve ser feito, pois cada sistema operacional possui suas peculiaridades. Por isso, é necessário seguir os passos definidos nos links que serão fornecidos, de acordo com o sistema operacional que será utilizado.

a) Java Development Kit

Antes de mais nada é necessário instalar o Java Development Kit (JDK), um conjunto de aplicações e bibliotecas necessário para se desenvolver aplicações em Java.

Esse livro está baseado no JDK 8. Ele pode ser encontrado através do seguinte link:

<https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

Para baixá-lo, é necessário aceitar os termos da licença de uso, escolher o sistema operacional e sua versão. Note que alguns sistemas operacionais podem ter passos adicionais para a configuração do JDK 8.

b) IntelliJ IDEA Community Edition

Para desenvolver as aplicações de exemplo que serão criadas nesse livro, é necessário instalar um ambiente integrado de desenvolvimento, ou como é conhecido pela sua sigla em Inglês - IDE.

A IDE que será utilizada é o IntelliJ IDEA Community Edition, do fabricante JetBrains. Essa é uma das mais modernas IDE para desenvolvimento em Java e outras linguagens de programação.

Ele pode ser encontrado no seguinte endereço: <https://www.jetbrains.com/idea/download>

Baixe e instale a versão compatível com seu sistema operacional, seguindo as instruções definidas no site da JetBrains.

Caso o leitor queira aprender mais como trabalhar com o IntelliJ IDEA, há um excelente conjunto de tutoriais em vídeo na seguinte página de documentação do site da JetBrains: <https://www.jetbrains.com/idea/documentation/>

c) Repositório de código

Esse é um passo opcional, caso o leitor queira armazenar o código fonte dos projetos de exemplo em um repositório de código.

Uma excelente opção para isso é utilizar o GitHub, que permite a criação de repositórios públicos e privados sem a necessidade de se pagar uma mensalidade.

Para criar uma conta no GitHub, acesse o link a seguir e siga os passos para o preenchimento do formulário:

<https://github.com/join>

Para detalhes de qual plano escolher, acesse o seguinte link:

<https://github.com/pricing>

O **plano gratuito** é suficiente para acompanhamento desse livro.

d) Cliente MySQL

Como algumas aplicações de exemplo irão utilizar uma base de dados relacional MySQL, será interessante ter um cliente para acessá-lo, ver suas tabelas e seus dados.

Existem várias aplicações clientes para esse tipo de trabalho. Uma excelente opção é o MySQL Workbench em sua versão gratuita. Ele pode ser baixado nesse endereço:

<https://dev.mysql.com/downloads/workbench/>

Selecione a opção adequada ao seu sistema operacional e siga as instruções de instalação oferecidas na página do fabricante.

No capítulo em que uma base de dados for utilizada por uma aplicação, haverá instruções específicas de como configurar o MySQL Workbench para se conectar a tal base. Por isso, não se preocupe em fazer passos adicionais. No momento, só é necessário instalar o cliente do MySQL Workbench.

2.3) Docker

Executar aplicações em *container* é uma tendência em *cloud computing* e também em aplicações distribuídas em micro-serviços. Esses são detalhes que ainda serão melhor explicados em capítulos a frente.

Como o Docker funciona e o no que ele ajudará ficará para alguns capítulos a frente, pois o intuito desse é apenas apresentar o que deve ser instalado.

a) Conta no Docker

Um dos processos que será realizado ao longo desse livro é a criação de uma imagem de um *container* Docker para a execução da aplicação em um AWS ECS. A princípio parece algo complexo, mas na verdade é bem simples se alguns passos forem seguidos corretamente.

Para a utilização dessa imagem, é interessante colocá-la em um repositório para que a AWS possa baixa-lo e utiliza-lo. No momento, apenas crie uma conta no *hub* do Docker, no link a seguir:

<https://hub.docker.com>

Essa conta é gratuita e servirá totalmente para os objetivos desse livro.

b) Docker

Da mesma forma como as demais aplicações apresentadas nesse capítulo, você deve instalar o Docker de acordo com a versão do seu sistema operacional. Para baixá-lo, veja as instruções contidas no link à seguir:

<https://docs.docker.com/install/>

Se quiser ir diretamente aos links para Mac e Windows, aqui estão:

- Mac: <https://docs.docker.com/docker-for-mac/install/>
- Windows: <https://docs.docker.com/docker-for-windows/install/>

É importante ler as instruções de preparação e instalação. Normalmente não há problemas em computadores mais novos, mas como o Docker trabalha com virtualização de hardware, assim como o VMWare e o VirtualBox, podem existir certas incompatibilidades ou configurações adicionais que deverão ser feitas.

c) Repositório no Docker Hub

Depois que a conta foi criada no Docker Hub, será necessário criar um repositório para cada imagem que for sendo criada. Esse passo será detalhado mais adiante, quando a primeira imagem for criada.

Nesse momento, basta que o leitor tenha apenas a conta no Docker Hub.

2.4 - Postman

Postman é um aplicativo gratuito muito útil e versátil. Com ele será possível fazer requisições às aplicações desenvolvidas nesse livro, tanto quando estiverem em execução na própria máquina de desenvolvimento, bem como quando estiverem hospedadas na AWS.

Para baixá-lo, siga as instruções contidas nesse link: <https://www.getpostman.com>

Com o Postman é possível organizar e salvar requisições montadas nele em forma de coleções, organizadas por projeto ou por contexto.

Também é possível criar uma conta no Postman, para armazenar e compartilhar as coleções de requisições que forem sendo criadas nele. Isso facilita bastante o trabalho em uma equipe de desenvolvedores e testadores.

2.5 - Conclusão

Nesse capítulo foram introduzidas as ferramentas e pacotes a serem instalados, bem como as contas a serem criadas para que o leitor possa acompanhar esse livro em sua totalidade.

Obviamente, algumas configurações ainda deveram ser feitas, seja em aplicações ou dentro da conta da AWS e do Docker Hub. Porém, é mais interessante do ponto de vista didático que o leitor faça isso no momento em que o recurso tiver que ser utilizado.

O próximo capítulo detalha o que será desenvolvido nesse livro, em termos de projetos e aplicações, explicando o que será utilizado de recursos da AWS. Ele é importante para que o leitor possa ter uma noção clara do que vem pela frente, capítulo a capítulo.

3 - Conteúdo do livro

Neste capítulo você terá uma visão geral do que será desenvolvido ao longo do livro, utilizando os recursos que a AWS oferece, além das tecnologias que serão utilizadas para tais aplicações

Para cobrir todo o conteúdo planejado nesse livro, mais de um sistema ou contexto de aplicações será desenvolvido, com propósitos didáticos para poder cobrir os serviços da AWS que serão explicados nesse livro.

O capítulo 4 dá uma visão geral sobre alguns conceitos como **contas, regiões, usuários e papéis**.

O capítulo 5 dá uma introdução à construção de aplicações em Java utilizando o **Spring Boot**. Ao longo do livro serão desenvolvidas duas aplicações utilizando essa base, por isso é importante apresentar tais conceitos de forma a deixar o leitor com um nível mínimo de preparação.

O capítulo 6 irá introduzir alguns conceitos do **Elastic Compute Cloud**, para criação de instâncias de máquinas para executarem aplicações. No exemplo a ser demonstrado, uma aplicação simples em Java será executada em duas instâncias do EC2, tendo um **application load balancer** controlando as requisições externas à aplicação, como pode ser visto na figura a seguir:

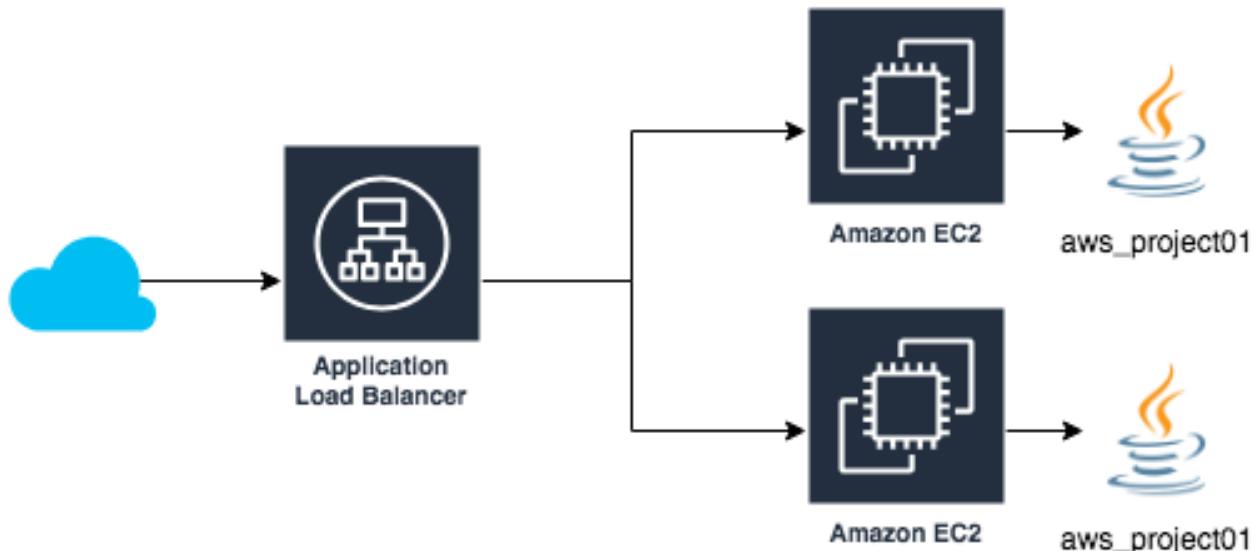


Diagrama do capítulo 6

No capítulo 7 será apresentado um recurso muito interessante da AWS, o **CloudFormation**, que é capaz de criar recursos a partir de arquivos em formato YAML ou JSON. Nesse capítulo, instâncias EC2 serão criadas e configuradas a partir de um arquivo no formato JSON, que será processado e “executado” pelo AWS CloudFormation:

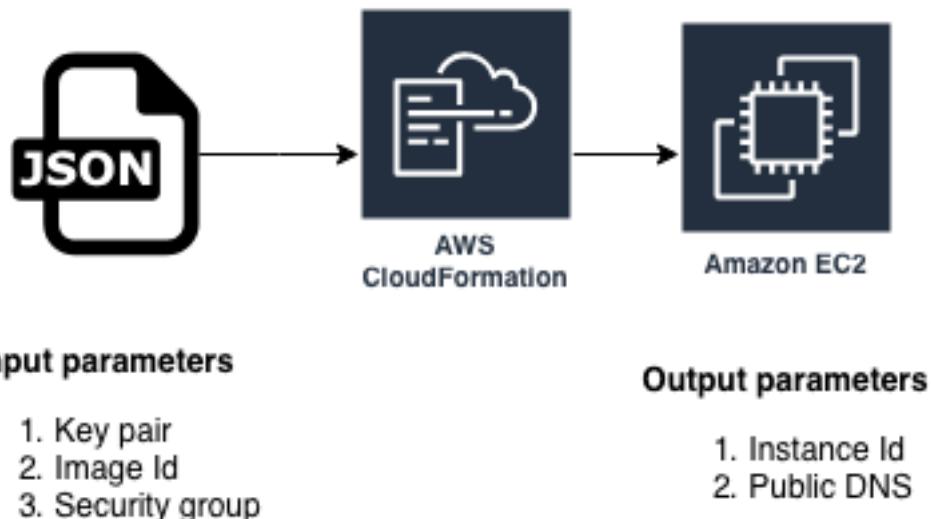
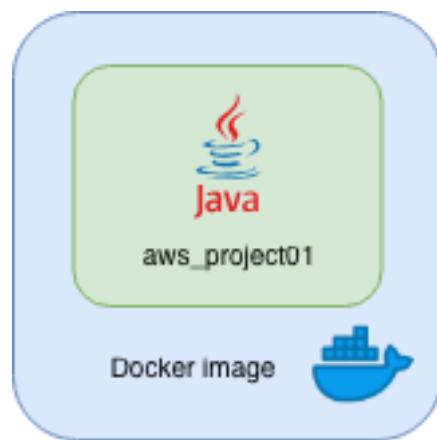


Diagrama do capítulo 7

O **capítulo 8** irá mostrar como colocar uma aplicação desenvolvida em Java e com Spring Boot para ser utilizada em *containers Docker*:



Aplicação aws_project01 rodando em uma imagem Docker

Isso será essencial para a apresentação do **capítulo 9**, onde será introduzido o conceito do **Amazon Elastic Container Service**, responsável pela criação de *clusters* para execução de tarefas de aplicações baseadas em *containers Docker*:

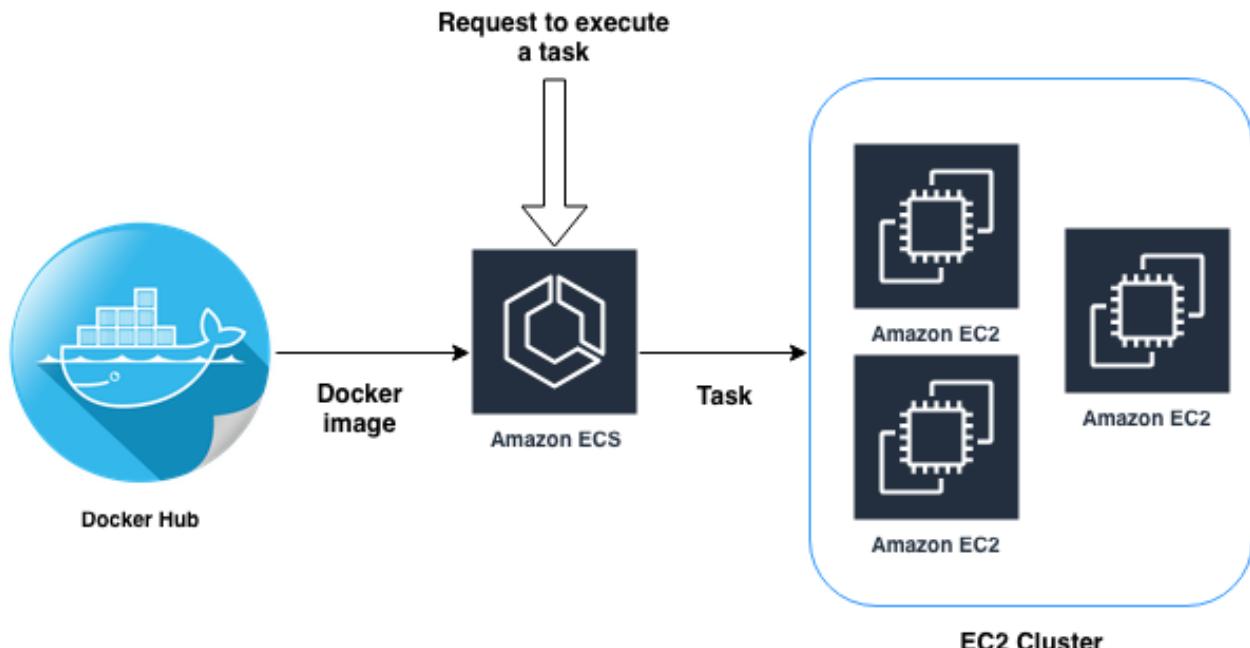
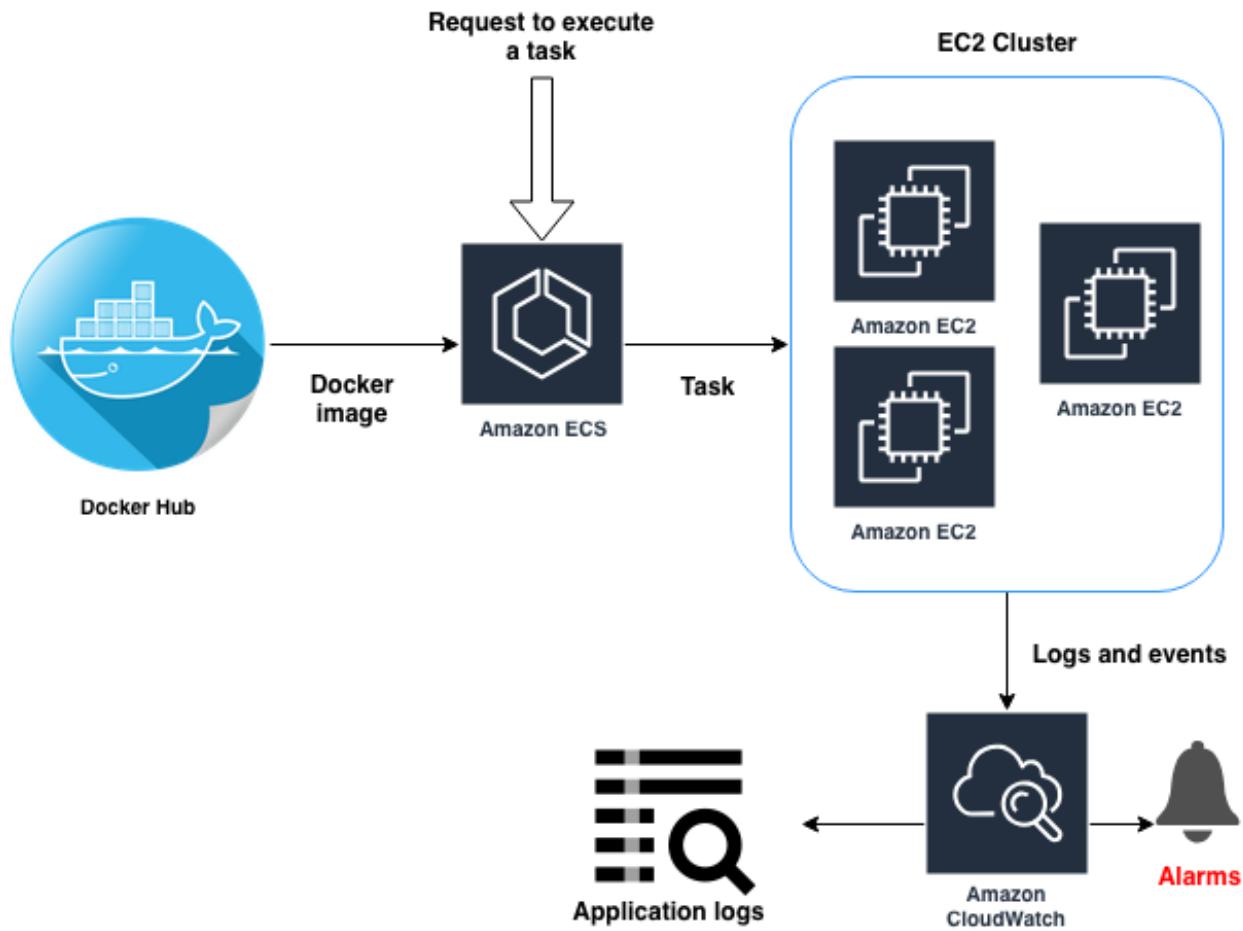


Diagrama do capítulo 9

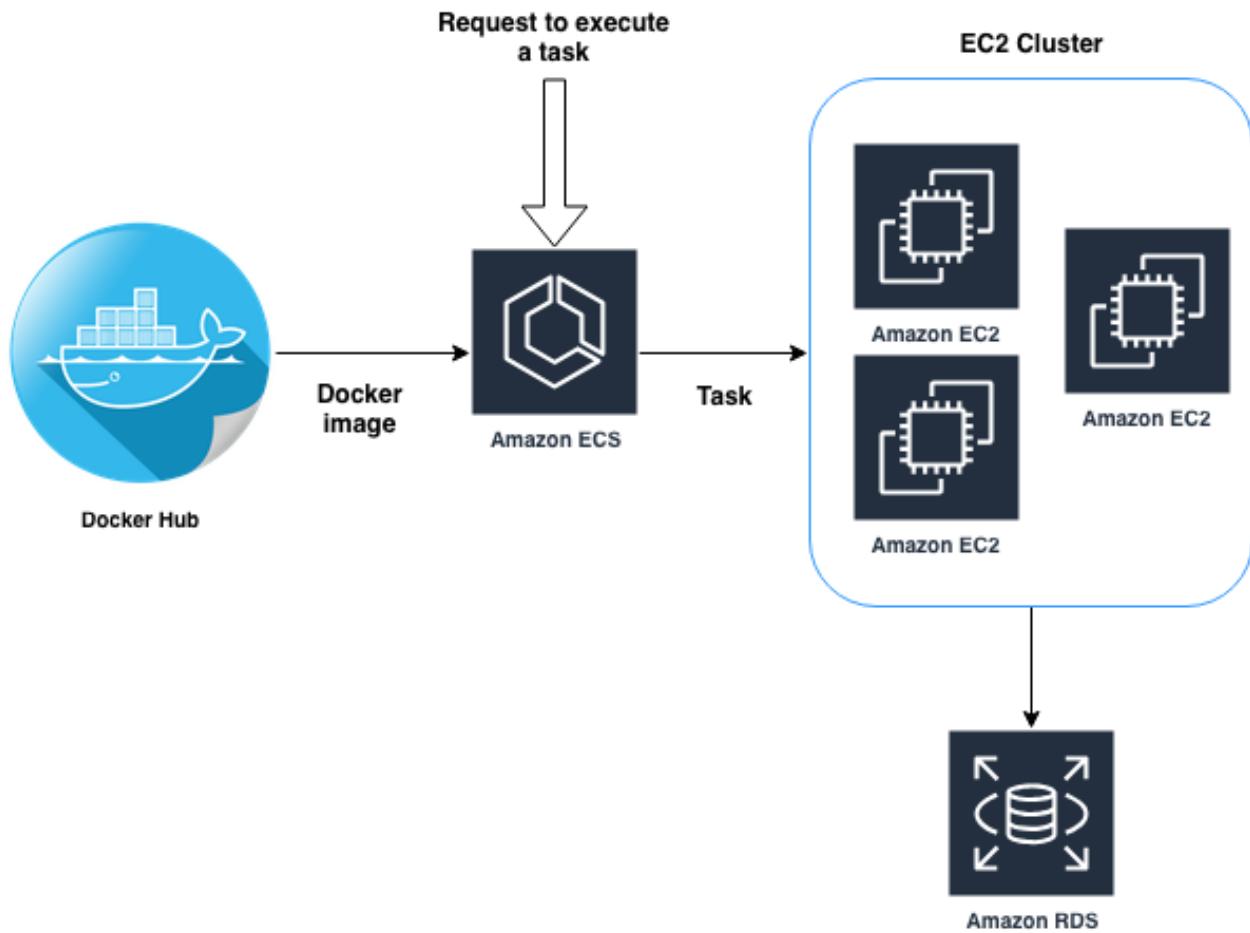
Nesse capítulo serão apresentados os conceitos de definições de tarefas e serviços a serem executados no ECS, através de *clusters* de instâncias EC2, capazes de executar mais de um serviço ao mesmo tempo, compartilhando os recursos oferecidos pelo *cluster*.

O **capítulo 10** apresentará um poderoso serviço de monitoramento da AWS, o **CloudWatch**, que permite a concentração de logs de aplicações, execução de outros recursos AWS, além da configuração de eventos e visualização de métricas de serviços através de gráficos.



CloudWatch e a aplicação aws_project01

O capítulo 11 mostrará como criar, utilizar e gerenciar instâncias de banco de dados no AWS Relational Database Service.



Acessando um banco de dados com o RDS

Com ele, será possível fazer com aplicações possam persistir seus dados em uma instância de um banco de dados, como o MySQL.

O capítulo 12 explica como utilizar o AWS Simple Notification Service, um serviço que permite que aplicações publiquem notificações em tópicos.

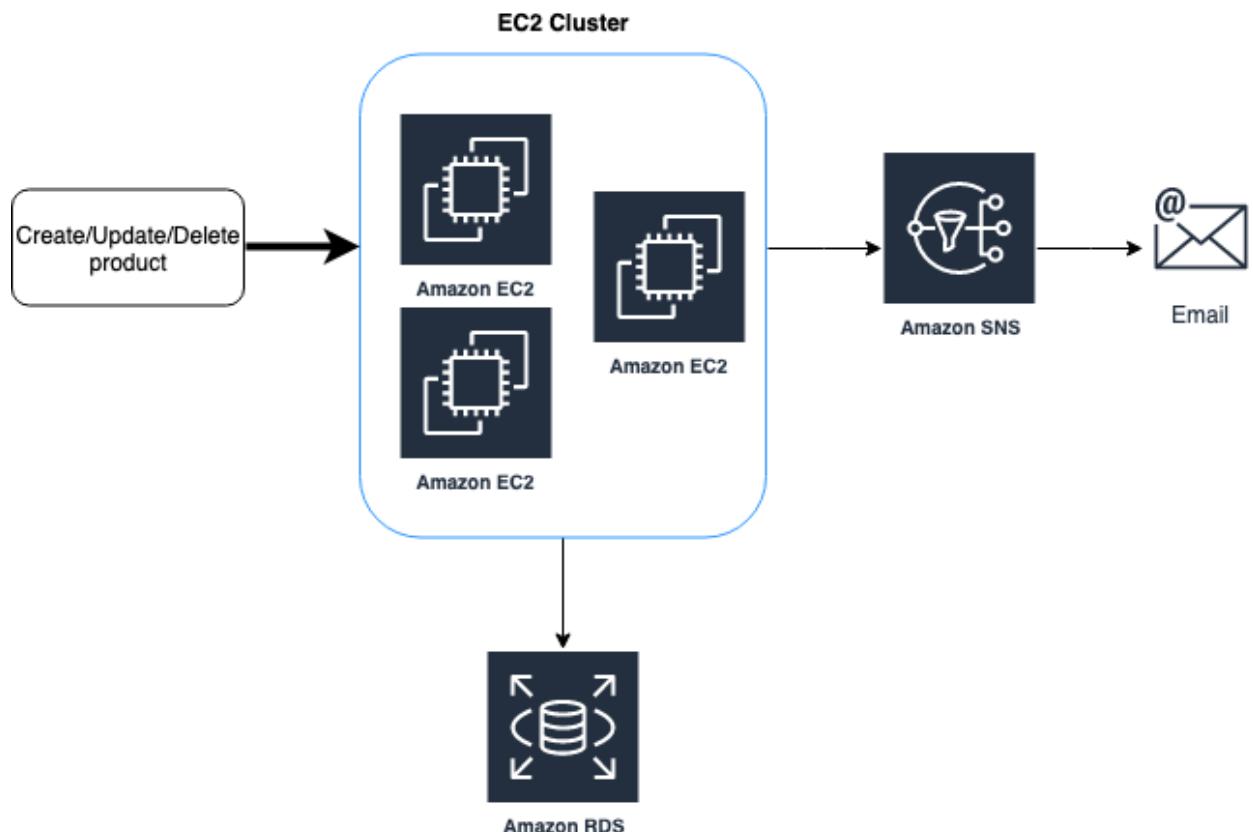
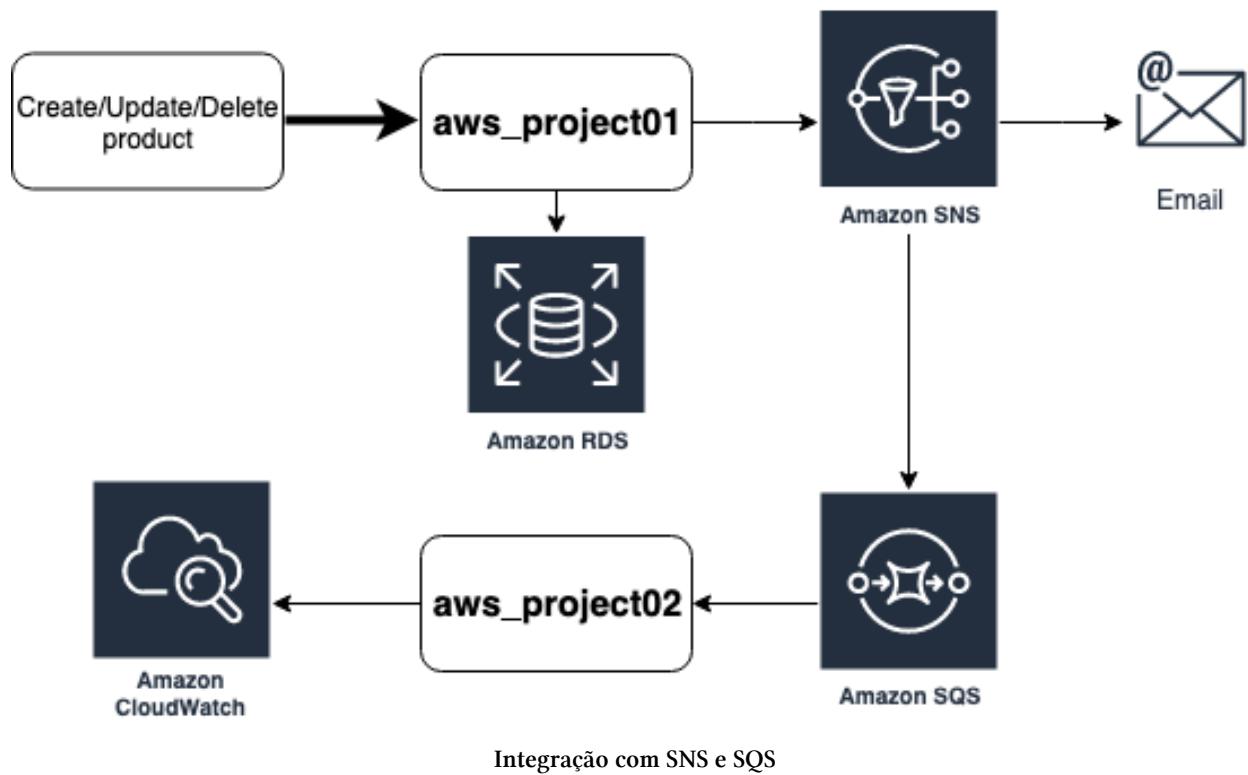


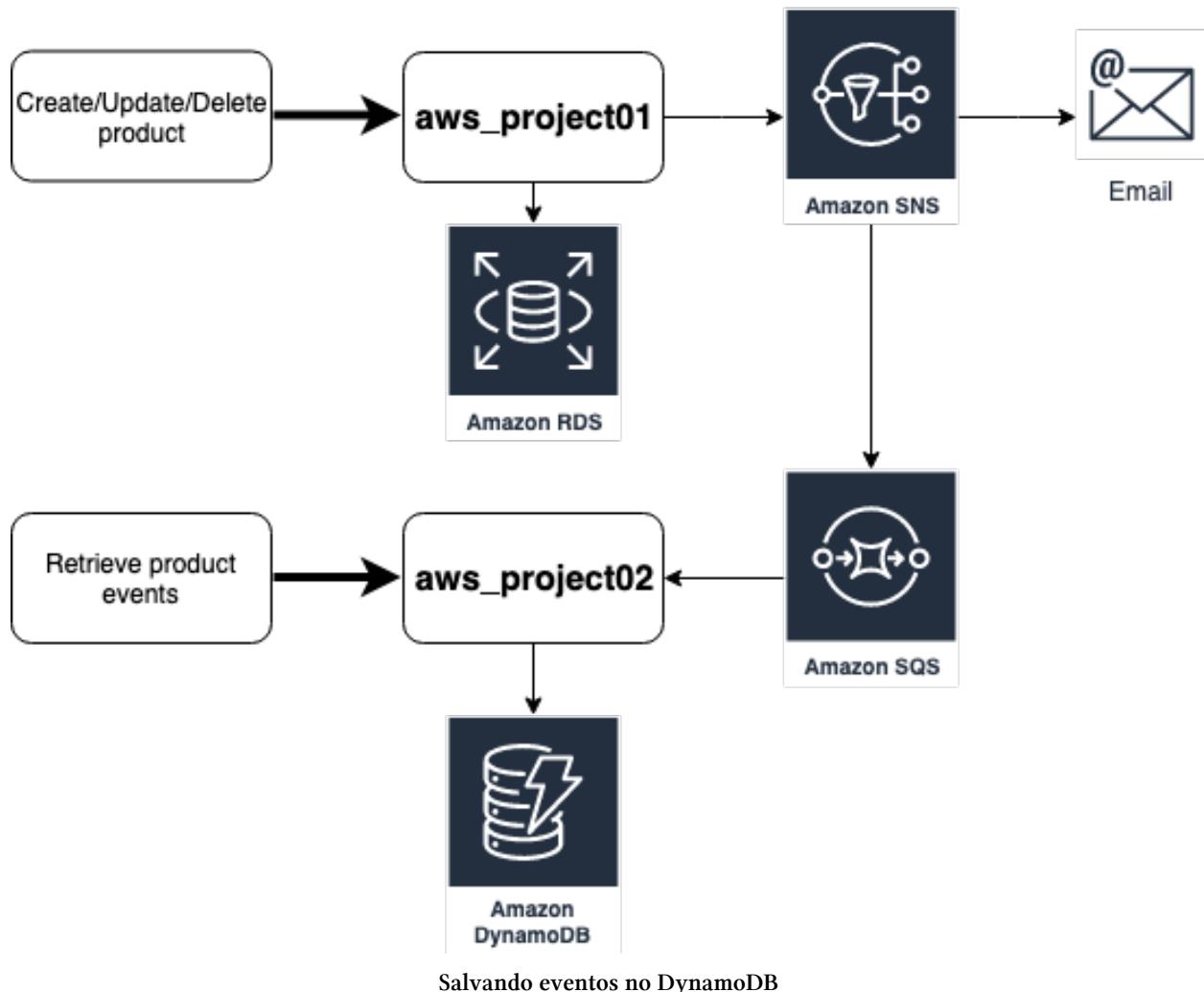
Diagrama da aplicação com SNS

Esses tópicos podem entregar mensagens para endereços de e-mail ou podem ser utilizados para publicar mensagens em filas, como o AWS Simple Queue Service, como será visto no capítulo 13:

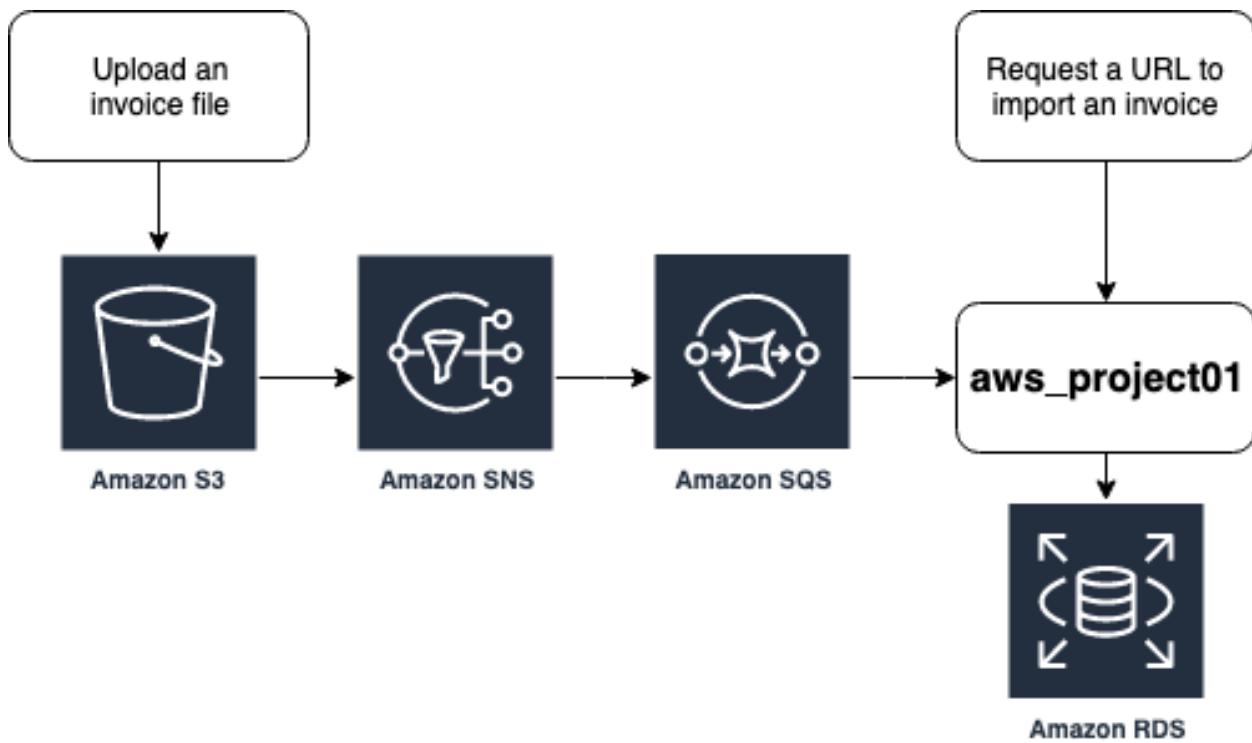


Dessa forma, é possível criar uma comunicação de forma assíncrona entre duas aplicações.

No capítulo 14, será detalhado conceitos iniciais do AWS DynamoDB, um serviço de banco de dados não relacionais, que permite a criação de itens que pode ser armazenados em forma de chave-valor e também em documentos.



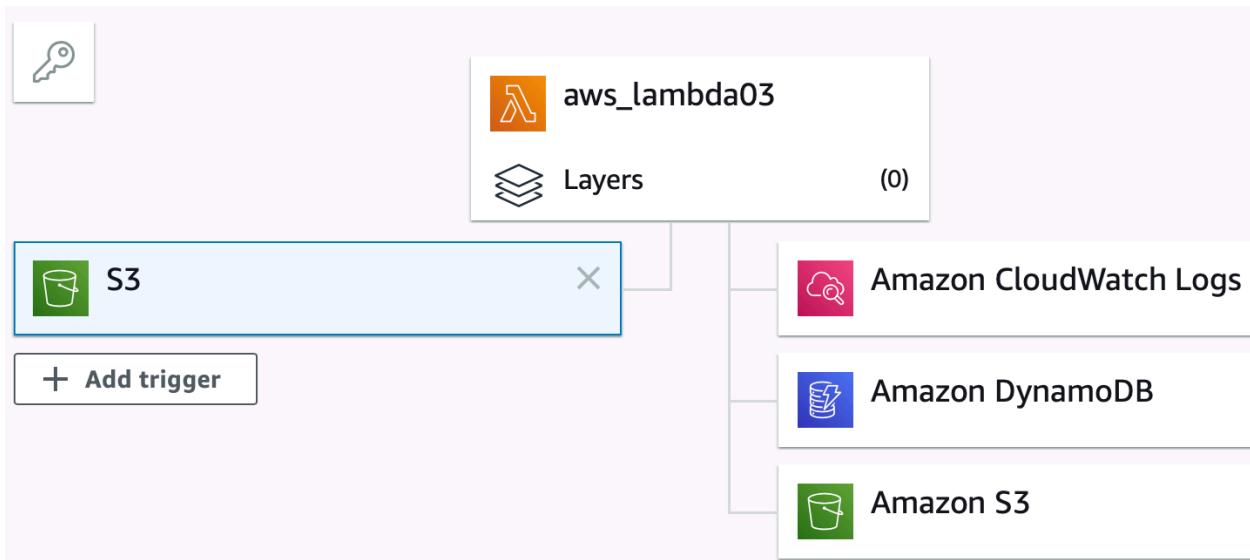
O capítulo 15 irá mostrar como utilizar o AWS Simple Storage Service, para armazenamento de arquivos, que podem ser gerenciados através de aplicações sendo executadas em clusters:



Importando arquivos com o S3

O exemplo que será desenvolvido conta com uma integração entre os serviços S3, SNS, SQS, ECS e RDS, mostrando como uma aplicação pode ser notificada sobre a inserção de um novo arquivo em um **bucket** do S3.

Finalmente, no capítulo 16, será apresentado conceitos de como construir aplicações *serverless* com o AWS Lambda.



Função Lambda acessando S3 e DynamoDB

Alguns exemplos serão desenvolvidos, mostrando como funções Lambdas podem ser utilizadas para as mais variadas aplicações. Na figura anterior mostra um diagrama de uma função Lambda acessando um *bucket* S3 e salvando os dados do arquivo em uma tabela do DynamoDB, além de gerar logs de sua execução no CloudWatch Logs.

Aproveite essa jornada de conhecimento e, caso necessite de ajuda, não hesite em chamar o autor: siecola@gmail.com

Divirta-se!

4 - Contas e regiões

Os recursos na AWS sempre são criados dentro de uma conta, como a que foi descrita no capítulo 2. Isso é para que a cobrança por tais recursos seja feita de forma centralizada, caso eles sejam utilizados além dos limites pre-estabelecidos de acordo com cada tipo de conta.

Esse capítulo irá tratar um pouco mais afundo sobre os usuários dentro das contas, bem como grupos, papéis e permissões.

Um outro fator que também é importante no momento da criação de cada recurso é a região onde ele será disponibilizado. Isso também será explicado ainda nesse capítulo, em algumas seções mais adiante.

4.1 - Identity and Access Management

No capítulo 2 foi apresentado o conceito do serviço IAM. Essa seção dedica-se a aprofundar mais sobre esse serviço, oferecido gratuitamente pela AWS.

Ele foi criado para permitir um melhor gerenciamento do acesso aos recursos AWS.

Imagine um cenário onde, dentro de uma mesma conta, o administrador deseje permitir que:

- Um conjunto de **recursos A** somente possam ser acessados pelo grupo de **usuários desenvolvedores**;
- Outro conjunto de **recursos B** somente possam ser acessados pelo grupo de **usuários testadores**.

Independente de quais sejam os recursos dentro dos conjuntos A ou B, para que isso seja possível de ser realizado, é necessário:

- Criar usuários específicos dentro dessa conta
- Criar um grupo para cada tipo de usuário: desenvolvedores e testadores
- Alocar os usuários criados em cada um dos grupos
- Definir os recursos AWS e colocá-los dentro das permissões de cada grupo.

Além disso, ainda seria possível definir algumas permissões, como por exemplo:

- Os usuários pertencentes ao grupo dos desenvolvedores podem alterar as configurações dos recursos AWS;
- Os usuários pertencentes ao grupo dos testadores **não podem realizar nenhuma configuração** nos recursos AWS.

Isso pode ser feito com o serviço IAM oferecido pelo AWS, que será detalhado nesse capítulo.

4.1.1 - Usuários

Como dito na seção anterior, os usuários dentro de uma conta da AWS são úteis para a definição de permissões de acesso aos recursos criados. Nessa seção serão mostrados os passos para a criação de um usuário, bem como a obtenção de suas credenciais de acesso.

Quando uma aplicação está em execução na AWS e ela deseja, por exemplo, ler uma fila qualquer ou acessar um banco de dados, ela deverá fazer através de uma chave de acesso atribuída a um usuário. Dessa forma a AWS pode verificar se tal usuário possui ou não permissão para realizar a operação desejada, em nome da aplicação.



O usuário que será criado aqui será utilizado nas **aplicações** que serão desenvolvidas ao longo do livro. Dessa forma, não será necessário utilizar as credenciais da conta criada no capítulo 2, pois essa deve ser mantida somente para a administração geral dos recursos da AWS, bem como os usuários e suas permissões.

Para criar um novo usuário dentro da sua conta, acesse o console da AWS, no endereço <https://console.aws.amazon.com> e entre com as credenciais de acesso da conta criada no capítulo 2.

Assim que o console é aberto, há um caixa para pesquisar o serviço ou recurso dentro da AWS, como pode ser visto na figura a seguir:

The screenshot shows the AWS Management Console homepage. At the top, there is a dark header bar with the AWS logo, a 'Services' dropdown, a 'Resource Groups' dropdown, and a user icon. Below the header, the title 'AWS Management Console' is displayed in large, bold letters. On the left side, there is a sidebar with the heading 'AWS services'. In the main content area, there is a 'Find services' section with a search bar containing the text 'IAM'. Below the search bar, the word 'IAM' is listed with the description 'Manage User Access and Encryption Keys'.

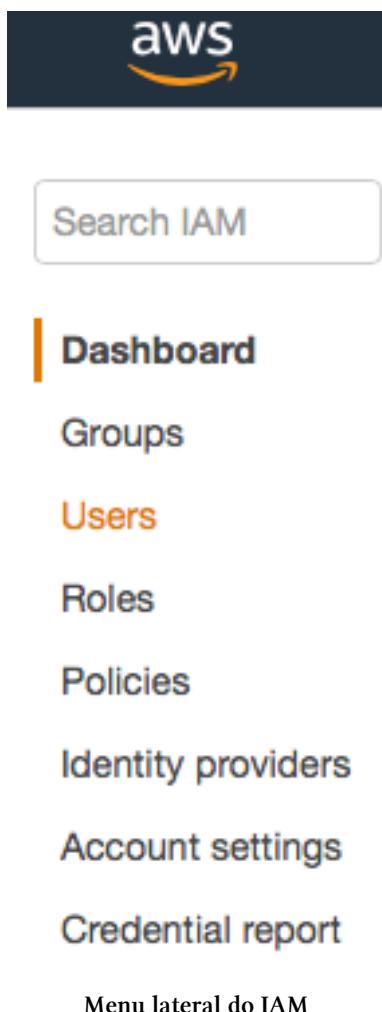
Pesquisa por serviços no console AWS

Nessa caixa de pesquisa, digite **IAM** para localizar o serviço para gerenciamento de usuários e acessos.

Dentro do IAM, é possível criar:

- Usuários;
- Grupos;
- Papéis;
- Políticas de acesso.

Na aba lateral da página do IAM, selecione a opção *Users* para acessar as opções de criação de usuários:



Nessa tela, clique no botão *Add user* para abrir as opções de criação de um novo usuário.

O **primeiro passo** da criação do usuário pede seu nome e o tipo de acesso:

Add user

1

2

Set user details

You can add multiple users at once with the same access type and permissions. [Learn more](#)

User name*

[Add another user](#)

Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. [Learn more](#)

Access type* **Programmatic access**

Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

AWS Management Console access

Enables a **password** that allows users to sign-in to the AWS Management Console.

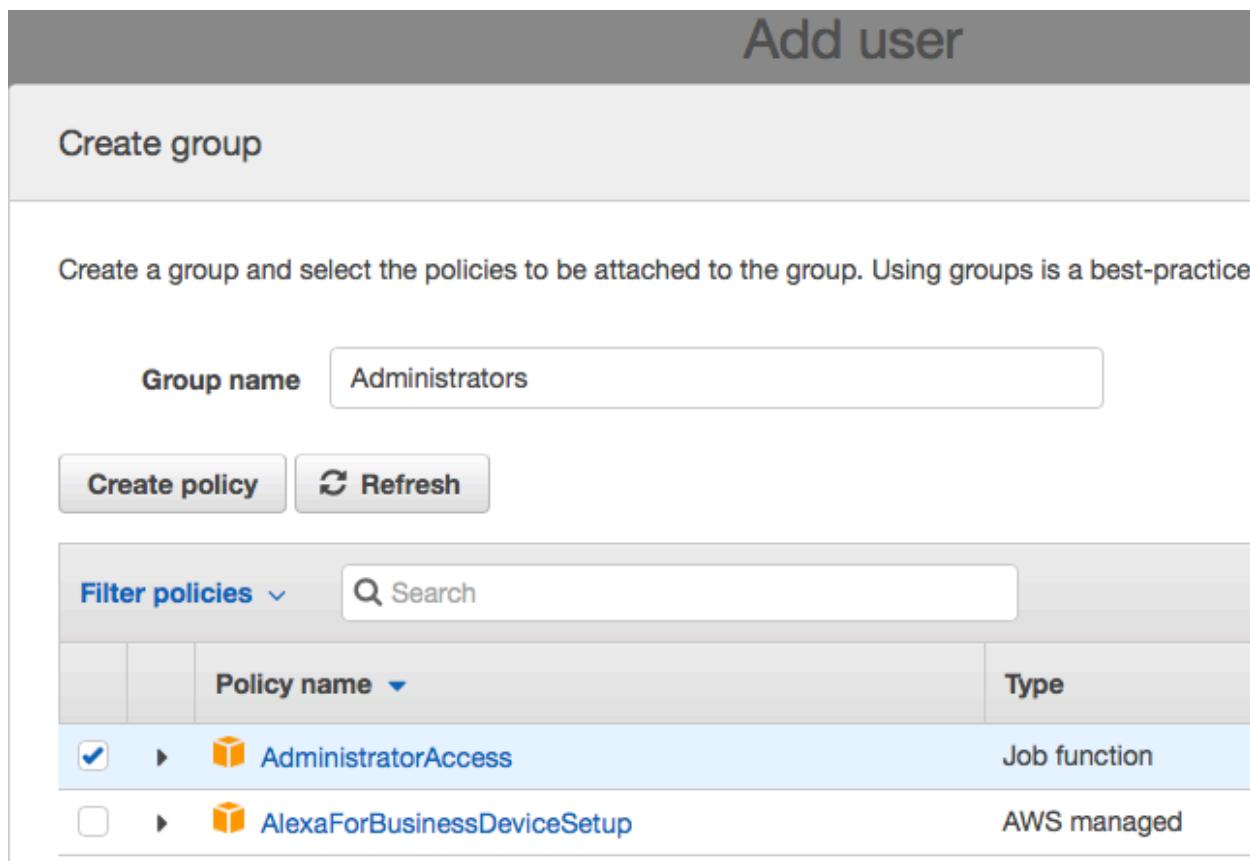
Criando usuário no IAM

Coloque **user1** para o nome do usuário e deixe somente a opção *Programmatic Access* habilitada, pois esse usuário será utilizado somente pelas aplicações que serão executadas na AWS.

A segunda opção **AWS Management Console Access** criaria uma forma desse usuário acessar o console da AWS, da mesma forma com que é feita com a conta que foi criada no capítulo 2. Essa opção é interessante para dar a possibilidade de outros usuários, além do dono da conta, realizarem operações no console da AWS, que não é o caso para os exemplos desse livro.

Em seguida, clique no *Next* para passar para o próximo passo, que é a tela para a associação do usuário a um grupo.

Provavelmente não haverá nenhum grupo criado, por isso clique no botão *Create group*, que abrirá um janela com o campo nome do grupo e uma lista de políticas já pre-definidas:



Criando um grupo no IAM

Deixe o nome do grupo como *Administrators* e selecione a primeira política, com nome *AdministratorAccess*. Isso fará com que os usuários que pertencerem a esse grupo poderão ter acesso total a todos os recursos da AWS. A seção **Políticas**, mais adiante, irá detalhar um pouco mais sobre o que isso realmente significará.

Em seguida clique no botão *Create group* dessa janela.

De volta na tela do passo 2 da criação do usuário, marque o grupo *Administrators*, recém criado, para incluir o novo usuário nele. Em seguida, clique no botão *Next*.

O passo 3 para a criação de *Tags* é opcional e serve para colocar informações adicionais ao usuário, como nome, e-mail, departamento, etc. Deixe tudo em branco e passe para o passo seguinte.

No passo 4, que é a revisão das configurações do novo usuário, ele deverá ter ficado como mostra a figura a seguir:

Add user

1 2 3 4

Review

Review your choices. After you create the user, you can view and download the autogenerated password and access key.

User details

User name	user1
AWS access type	Programmatic access - with an access key
Permissions boundary	Permissions boundary is not set

Permissions summary

The user shown above will be added to the following groups.

Type	Name
Group	Administrators

Tags

No tags were added.

Revisando o usuário a ser criado

Caso tudo esteja como a figura anterior, clique no botão *Create user* para finalizar o processo.

Na tela seguinte será exibida uma lista dos usuários que foram criados, que deverá conter somente o *user1*. Clique no botão *Download .csv* para baixar o arquivo com as credenciais de acesso do usuário.

Nesse arquivo existem 2 campos:

- **Acess key ID:** essa é a chave que identifica o usuário. Ela será usada pelas aplicações que forem desenvolvidas, quando elas quiserem acessar algum recurso da AWS;
- **Secret access key:** é a senha utilizada para autenticar a chave de acesso do usuário e também será utilizada pela aplicação para identificar o usuário quando desejar acessar recursos da AWS.

Guarde essas informações em um local seguro, pois elas serão utilizadas mais adiante.

Pronto! O usuário já está criado e pronto para ser utilizado.

4.1.2 - Grupos

Agrupar os usuários pode ser uma boa estratégia para definir políticas de acesso a eles, por exemplo como administradores, desenvolvedores e testadores. Dessa forma, cada um deles possuirá um conjunto de permissões que se aplicam a cada um deles.

Na seção anterior foi criado um grupo *Administrators*, com o objetivo de colocar o usuário recém criado nele.

No IAM, vários grupos podem ser criados, com políticas de acesso definidas a cada um deles.

4.1.3 - Políticas

Quando o grupo *Administrators* foi criado, a política *AdministratorAccess* foi associada a ele:

The screenshot shows the AWS IAM Groups page. On the left, there's a sidebar with links like Dashboard, Groups (which is selected and highlighted in orange), Users, Roles, Policies, Identity providers, Account settings, Credential report, and Encryption keys. The main content area has a breadcrumb navigation: IAM > Groups > Administrators. Below this, there's a summary section with fields for Group ARN (empty), Users (in this group): 1, Path: /, and Creation Time (empty). At the bottom of this section are three tabs: Users (disabled), Permissions (selected and highlighted in orange), and Access Advisor. Under the Permissions tab, there's a section titled "Managed Policies" with a message stating "The following managed policies are attached to this group. You can attach up to 10 managed policies." Below this is a blue "Attach Policy" button. A table follows, showing one policy: **AdministratorAccess**. The table has columns for Policy Name and Actions. The "Actions" column contains a "Show Policy" link.

Policy Name	Actions
AdministratorAccess	Show Policy

Política associada ao grupo

Essa é uma das várias políticas que existem criadas, como se fossem templates que podem ser utilizados. Especificamente essa dá acesso total a todos os recursos da AWS, dentro da conta em que o grupo e consequentemente seus usuários existem. Clicando na opção *Show Policy*, é possível ver o que ela realmente faz:



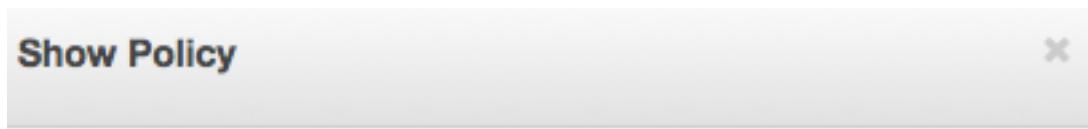
```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": "*",  
            "Resource": "*"  
        }  
    ]  
}
```

Política de acesso do administrador

Obviamente que essa política dá um poder muito grande aos usuários que a possuem, mas para os objetivos desse livro, não necessidade de restringir acesso ao usuário que foi criado, uma vez que o intuito é justamente esse.

Em uma situação diferente, como o cenário que foi descrito no início desse capítulo, com usuários **desenvolvedores e testadores**, é interessante refinar as permissões para cada grupo.

Em um outro exemplo hipotético, caso usuários de um grupo pudesse ter somente acesso de leitura em uma fila da AWS, o SQS, a política ideal seria a de nome *AmazonSQSReadOnlyAccess*, que possui as seguintes configurações:



```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Action": [  
                "sns:Publish",  
                "sns:ListTopics",  
                "sns:ListSubscriptionsByTopic",  
                "sns:GetTopicAttributes",  
                "sns:DeleteTopic",  
                "sns:CreateTopic",  
                "sns:Subscribe",  
                "sns:ListSubscriptions",  
                "sns:DeleteSubscription",  
                "sns:Unsubscribe"  
            ],  
            "Effect": "Allow",  
            "Resource": "*"  
        }  
    ]  
}
```

Permissão de leitura somente em SQS

As ações dessa política definem que os usuários que a possuem podem apenas ler informações de filas, ou *queues* do Inglês.

As definições corretas das políticas que devem ser atribuídas a cada grupo de usuários, aos recursos AWS e às ações nesses recursos não é uma tarefa simples, pois deve-se ter conhecimento específico sobre cada recurso AWS. A ideia aqui é apenas mostrar onde e como isso pode ser feito, através do serviço IAM da AWS.

A política que foi criada para o grupo *Administrator* garante que os usuários dele terão acesso a todas as operações em todos os recursos da AWS, facilitando o acompanhamento das atividades desse livro. Porém, lembre-se que essa é uma política muito abrangente e que deve ser utilizada em casos muito específicos e com cuidado.

4.2 - Regiões

Quando um recurso é criado na AWS, na verdade ele é criado em uma das regiões que a Amazon possui. Tais regiões estão geograficamente separadas para proporcionar uma experiência melhor

aos usuários finais que utilizam as aplicações hospedadas na AWS, pois, a princípio, quanto menor a distância física entre o usuário e o local onde a aplicação está hospedada, menor a **latência na comunicação** via Internet.

Obviamente, a AWS não possui regiões em todos os países do mundo, mas ela tenta ter pontos de presença nos locais de maior tráfego de seus clientes.

No momento da criação de um recurso, por exemplo uma instância EC2 para execução de uma determinada aplicação, o administrador de tal aplicação pode escolher em qual região esse recurso será criado. Um dos critérios pode ser a posição geográfica da maioria de seus clientes, tentando deixar a aplicação fisicamente mais próxima deles.

Outros critérios para a escolha da região onde o recurso será criado podem ser o **custo de hospedagem** ou até mesmo **políticas de proteção de dados** de cada país.

Existe ainda um fator importante na consideração da região a ser utilizada, que é a disponibilidade da existência do recurso, pois nem todas as regiões estão aptas a ter qualquer recurso da AWS. Isso acontece com maior frequência em recursos ou serviços recém lançados pela AWS, mas que persiste por pouco tempo.

Para ter informações de todas as regiões e de quais estão serviços estão disponíveis em cada uma deles, consulte o link a seguir:

<https://aws.amazon.com/about-aws/global-infrastructure/regional-product-services/>

Essa página é constantemente atualizada com a lista completa de serviços e regiões da AWS.

Para cada região, há uma sigla que a define e que também é utilizada como prefixo nos nomes dos recursos. A seguir, uma pequena tabela com os nomes de algumas regiões e suas siglas:

Nome da região	Sigla da região
US West (Oregon)	us-west-2
US East (N. Virginia)	us-east-1
Asia Pacific (Tokyo)	ap-northeast-1
South America (São Paulo)	sa-east-1
EU (London)	eu-west-2

Nos próximos capítulos, quando os recursos AWS começarão a ser criados, as regiões terão que ser escolhidas e ficará mais claro a composição do nome do recurso, juntamente com a sigla da região escolhida.

4.3 - Zonas de disponibilidades

Ainda no contexto de regiões da AWS, existe o conceito das zonas de disponibilidade, que são **locais fisicamente separados dentro de uma mesma região**, conectados com uma rede de baixa latência e alta disponibilidade.

As zonas podem ser utilizadas para elevar a disponibilidade de um serviço ou aplicação, princi-

palmente em casos de catástrofes, pois a AWS consegue redirecionar o tráfego de uma região para outra, mantendo tal aplicação disponível.

No momento da criação de um recurso, por exemplo uma instância EC2, o administrador do sistema pode escolher que tal recurso ficará apenas em uma zona, dentro da região, ou nas múltiplas zonas dela. Obviamente, a segunda opção tem um custo elevado, pois há um consumo de recursos lógicos e físicos maior.

A utilização de várias zonas de disponibilidades dentro de uma mesma região é um recurso muito interessante do ponto de vista técnico, mas sua aplicação está restrita a sistemas que exigem uma disponibilidade e tolerância a falhas muito elevada, fazendo com que seu custo de operação seja igualmente elevado.

4.4 - Configuração do AWS CLI

Agora que o usuário a ser utilizado pelas aplicações desse livro foram criadas e os conceitos sobre regiões foi explicado, a configuração do AWS CLI, que foi instalado no capítulo 2, pode ser concluída.

Como dito, o AWS CLI é um conjunto de ferramentas que permite ao administrador do sistema criar e configurar recursos na AWS, através de comandos no prompt de seu computador, sem a necessidade de acessar o console Web.

Esse recurso também pode ser utilizado para obter informações mais detalhadas e de forma rápida sobre qualquer recurso que já tenha sido criado, além de permitir que scripts possam ser criados para automatização de tarefas a serem executadas no recursos da AWS.

Tendo instalado o AWS CLI, abra o prompt de comando de seu computador e digite o seguinte comando:

```
aws configure
```

A **primeira opção** que deve ser configurada é a *access key* do usuário que irá acessar a AWS através da linha comando. Essa informação está presente no arquivo `.csv` que foi baixado na seção 4.1.1 desse capítulo, no momento da criação do usuário. Digite o valor desse campo nessa primeira configuração.

O **segundo campo** é a *secret key*, que também está presente no mesmo arquivo `.csv` que foi baixado. Digite seu valor nesse campo.

O **terceiro campo** é a região padrão que o AWS CLI vai se conectar. Nesse campo, digite `us-east-1` para configurá-lo para apontar para US East (N. Virginia), pois todos os recursos que serão criados nesse livro ficarão nessa região, por conveniência.

O **quarto campo** se refere ao formato das respostas que serão fornecidas pelo AWS CLI no prompt de comandos. Digite o valor `json`, dessa forma, as respostas estão nesse padrão, para melhor visualização.

Pronto! Agora o AWS CLI já está totalmente configurado.

Apenas para ver se tudo está realmente funcionando, digite um comando de exemplo, para listas as filas SQS que sua conta possui:

```
aws sqs list-queues
```

Se nenhuma fila foi criada ainda dentro dessa conta, é provável que a resposta seja uma lista vazia, mas se isso aconteceu, pelo menos significa que o AWS CLI já está configurado para poder acessar os recursos AWS.



Para ter uma documentação completa dos comandos do AWS CLI, consulte sua documentação de referência no seguinte link: <https://docs.aws.amazon.com/cli/latest/reference/>

Nos capítulos seguintes, onde recursos AWS serão criados, será possível obter informações adicionais e até interagir com eles através do AWS CLI.

4.4 - Conclusão

Nesse capítulo foi ensinado conceitos de contas, regiões e zonas de disponibilidade, além do importante serviço Identity and Access Management da AWS.

Além disso, foi finalizada a configuração do AWS CLI, com informações do usuário que foi criado.

No capítulo seguinte, serão apresentados alguns conceitos iniciais de **Spring Boot**, que será o *framework* base para a construção das aplicações que serão desenvolvidas nesse livro.

5 - Conceitos de Spring Boot

Spring Boot é um *framework* que permite a criação de aplicações Web em Java, de forma rápida e descomplicada, sem a necessidade de trabalhar com complexos arquivos `.xml` de configuração. A seguir, algumas de suas características mais evidentes:

- Pode-se utilizar **Maven** ou **Gradle** para gerenciamento de dependências;
- Permite a criação de aplicações *stand-alone*, que podem ser executadas sem a necessidade de se gerar um arquivo `.war` para instalação em servidores de aplicações;
- Não necessita de arquivos `.xml` de configuração;
- Ideal para criar micro-serviços a serem executados em *containers* como o Docker.

A maioria das aplicações que serão desenvolvidas nesse livro terão como base o Spring Boot, por isso esse capítulo oferece uma breve introdução de como criar uma aplicação simples, com as seguintes funcionalidades:

- Um **controller REST** para expor um endpoint;
- Mecanismo de geração de **logs**;
- Configuração de parâmetros de execução da aplicação;
- Criação de serviços gerenciados pelo Spring, para servirem a toda aplicação.

A aplicação que será construída nesse capítulo será utilizada nos dois capítulos seguintes, onde os seguintes conceitos serão ensinados:

- Criação de uma instância EC2 para execução de uma aplicação Java, no capítulo 6;
- Criação de um *cluster* ECS para a execução da aplicação em um *container* Docker, no capítulo 9;
- Visualização de logs e métricas da aplicação no AWS CloudWatch, no capítulo 10;
- E muito mais!



Muitos conceitos de Spring Boot ainda serão apresentados ao longo dos próximos capítulos, principalmente quando a aplicação for ganhando novas funcionalidades e interagindo com mais serviços da AWS. Se quiser se aprofundar em seus conceitos, consulte a documentação oficial: <https://spring.io/projects/spring-boot>

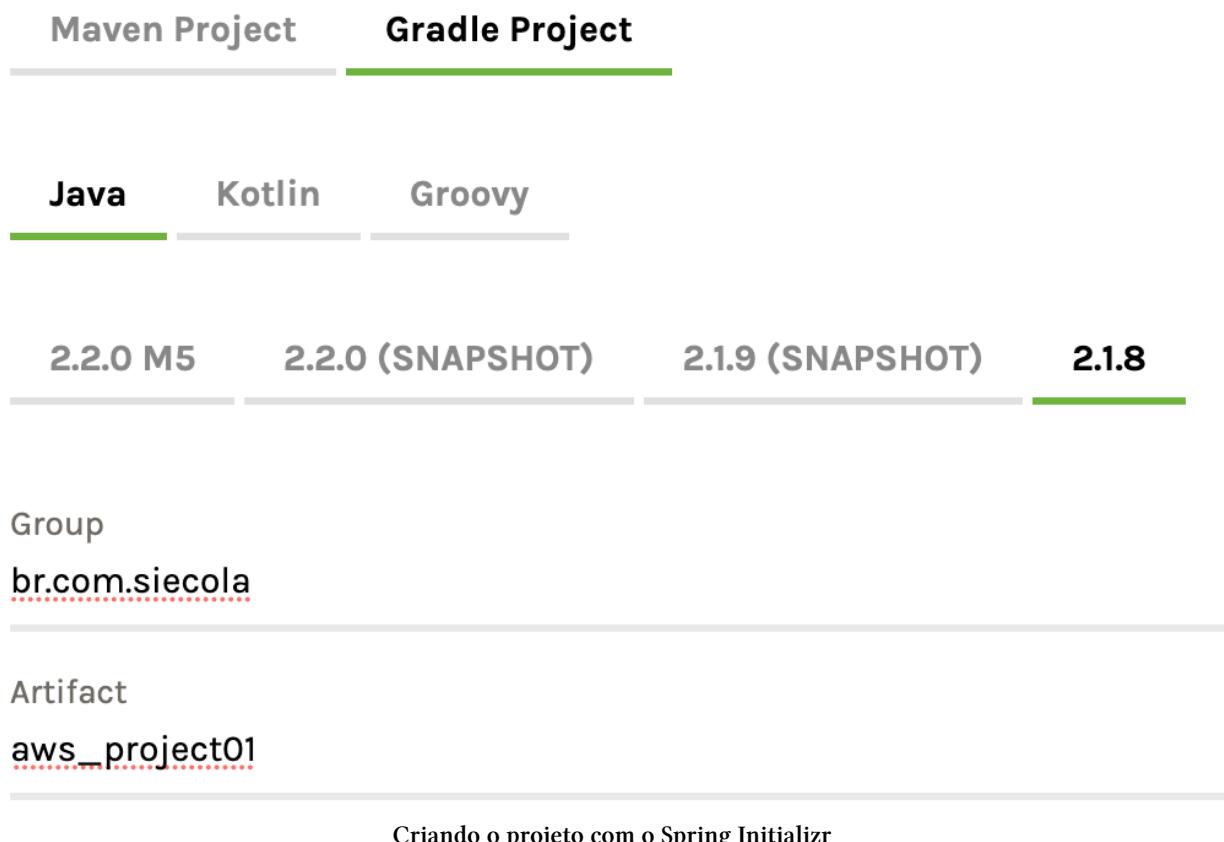
5.1 - Criando a base do projeto aws_project01

Nessa seção será ensinado como criar uma aplicação Spring Boot do zero, utilizando um projeto chamado **Spring Initializr**, que é uma página Web onde o usuário pode configurar seu projeto e adicionar várias dependências. Então, para começar acesse-o através do endereço: <https://start.spring.io>

Nas primeiras configurações que aparecem, selecione as seguintes opções:

- Gradle project;
- Java;
- Spring Boot versão 2.1.8 ou superior;
- Group: br.com.siecola
- Artifact: aws_project01

De tal forma que a tela fique semelhante a da figura a seguir:

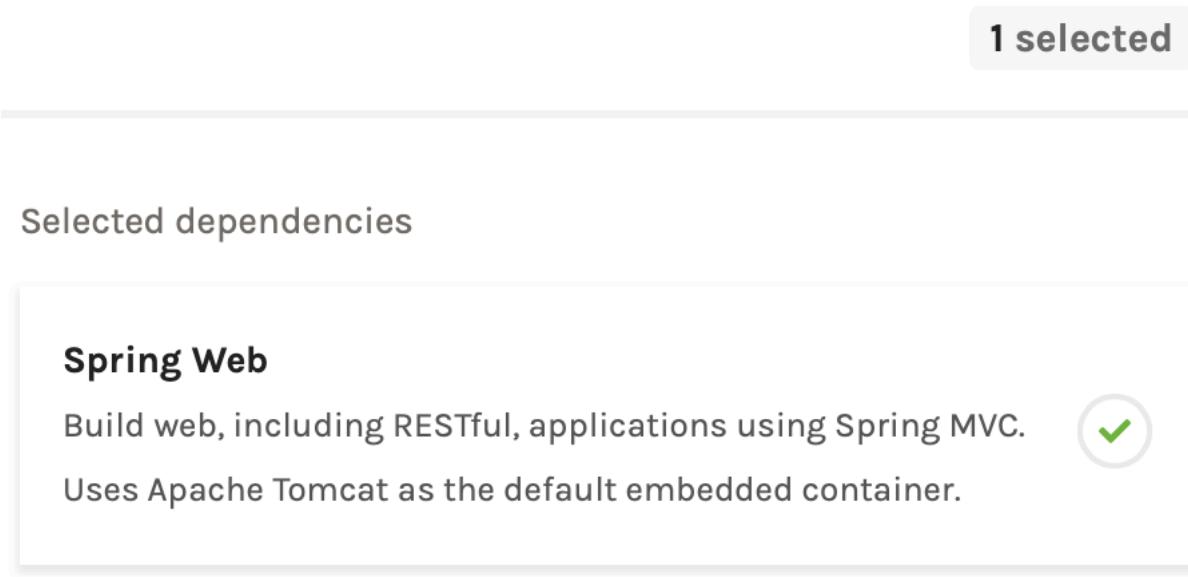


Ainda existem algumas opções que devem ser verificadas e configuradas. Isso pode ser feito clicando-se no link *Options*, para que tais opções apareçam.

Certifique-se que as seguintes configurações estão feitas como a seguir:

- Packaging: jar
- Java version: 8

Na seção Search dependencies to add, digite Web para adicionar essa dependência ao projeto. A figura a seguir mostra exatamente qual é essa dependência:

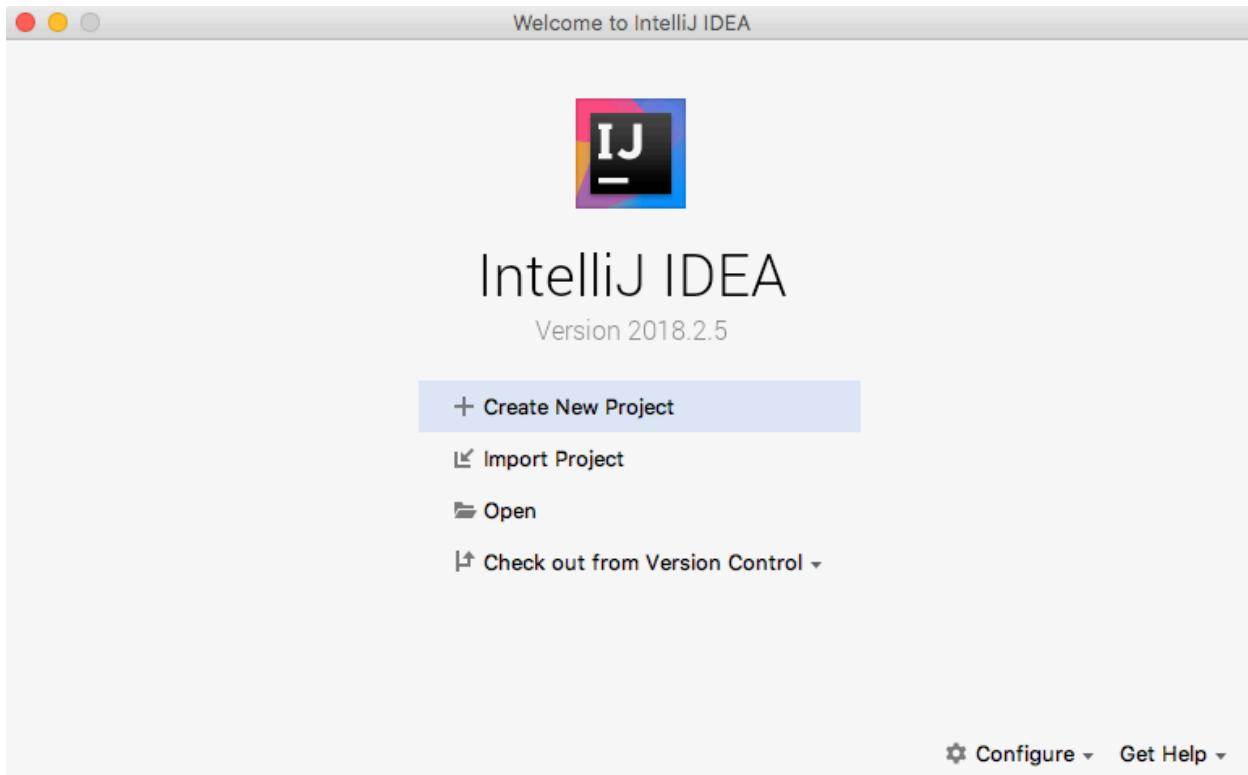


Adicionando a dependência Spring Web ao projeto

Para finalizar, clique no botão *Generate Project*, que fará com que o projeto seja criado e baixado para o seu computador, compactado em um arquivo .zip. Agora descompacte esse arquivo em um local de sua preferência para abri-lo no IntelliJ IDEA e começar a escrever o código!

5.2 - Abrindo projeto no IntelliJ IDEA

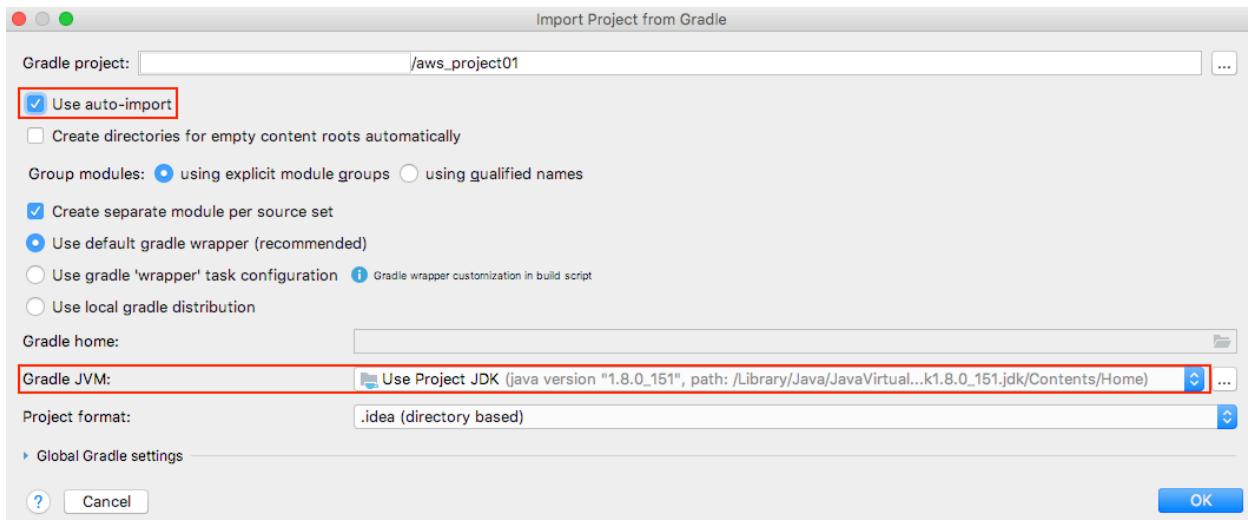
Depois de descompactar o arquivo .zip do projeto que foi criado, abra o IntelliJ IDEA. Sua primeira tela deverá ser semelhante a da figura a seguir:



Abrindo IntelliJ IDEA

Clique no opção *Open* e selecione a pasta onde o arquivo foi descompactado. O local correto para abrir o projeto deve ser onde está o arquivo `build.gradle`, que deverá estar dentro da pasta `aws_project01`.

Nesse momento, o IntelliJ IDEA irá apresentar uma tela com as seguintes opções de configuração:



Abrindo o projeto

Nessa é tela, é importante configurar duas opções:

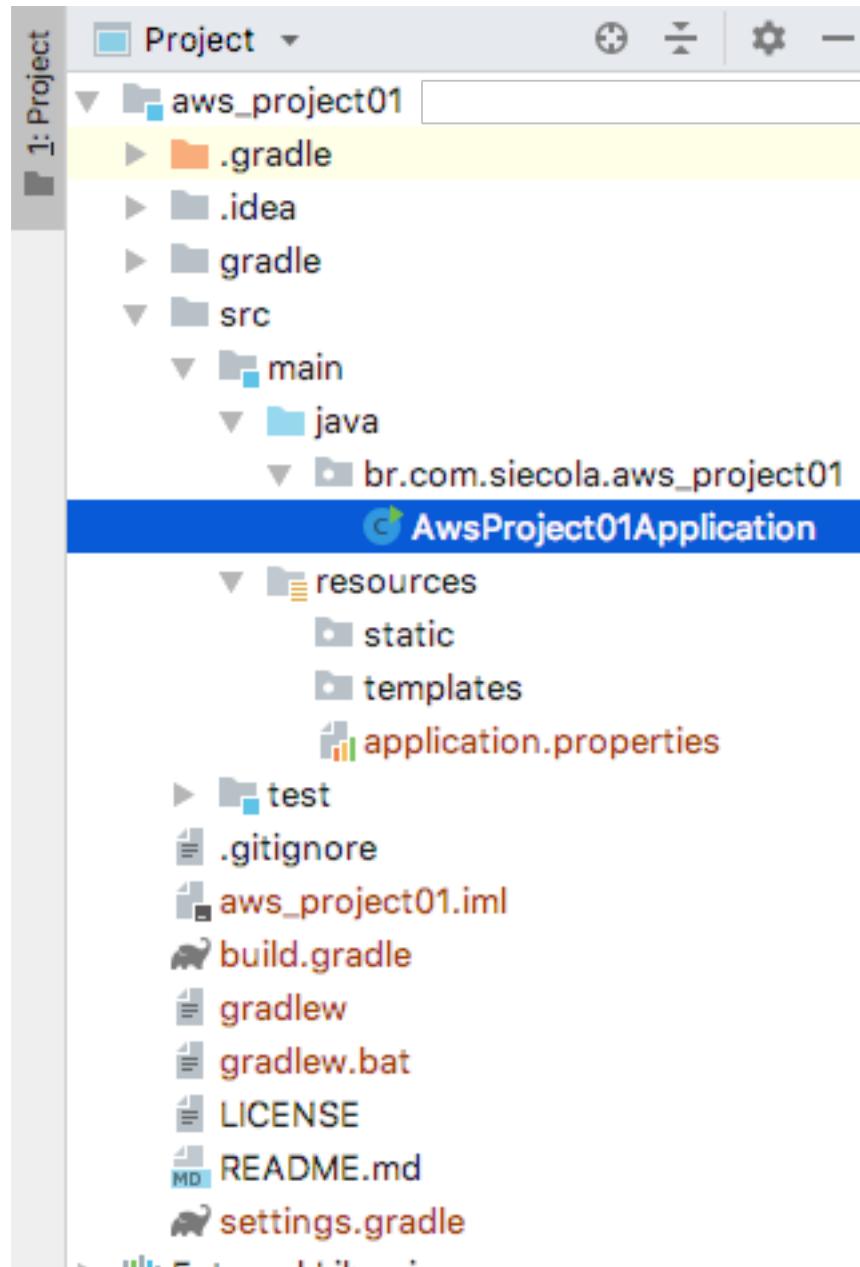
- Habilitar a opção *User auto-import*, que fará com que o IntelliJ IDEA baixe as dependências quando o projeto for aberto e sempre que uma nova for adicionada no arquivo `build.gradle`;
- Configurar o local do JDK: aponte para o local onde o JDK foi instalado no capítulo 2. Não se preocupe se a versão da figura está diferente.

Feito isso, clique no botão *OK* para concluir a abertura do projeto. Nesse momento, o IntelliJ IDEA vai abrir o projeto e começar a baixar as dependências necessárias para executá-lo. Dentre essas dependências estão partes do *Spring*, o *framework* base do projeto.

Aguarde até que todas as dependências sejam baixadas.

5.3 - Estrutura do projeto

Após a abertura do projeto e da finalização da importação de todas as dependências, é possível analisar a estrutura do projeto na aba *Project* do IntelliJ IDEA, como mostra a figura a seguir:



Estrutura do projeto

A seguir, uma breve explicação sobre as partes mais relevantes da estrutura do projeto, dentro da pasta `src`:

- O pacote `br.com.siecola.aws_project01` é o local onde todo o código fonte da aplicação será criado, organizado em pacotes;
- O arquivo `AwsProject01Application` é classe que contém a função `main`, ou seja, é o ponto de entrada da aplicação;

- A pasta `resources` é o local onde alguns arquivos de configuração deverão ficar. Já existe o arquivo `application.properties`, que como o próprio nome diz, é o local onde algumas propriedades da aplicação podem ser configuradas.

Fora da pasta `src` existem alguns arquivos que merecem uma atenção especial nesse momento:

- O arquivo `build.gradle` é o local onde, dentre outras coisas, deve-se fazer a configuração das dependências do projeto. Esse é o principal arquivo do projeto no que se refere ao processo de compilação e geração do executável da aplicação;
- O arquivo `settings.gradle` é o local onde as configurações do *Gradle* podem ser feitas. Normalmente não é necessário fazer nenhuma configuração adicional nesse arquivo;
- Por fim, o arquivo `aws_project01.iml`, criado pelo IntelliJ IDEA, é o que carrega informações do projeto dessa IDE, portanto, não é necessário alterá-lo.

A medida que novas funcionalidades forem sendo adicionadas nesse projeto, será necessário trabalhar com alterações em alguns arquivos de configuração, principalmente o `application.properties` e o `build.gradle`.

5.4 - Executando a aplicação pela primeira vez

Abra o arquivo `AwsProject01Application.java` para analisar a sua estrutura. Ele deve ser bem parecido com o trecho a seguir:

```
package br.com.siecola.aws_project01;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AwsProject01Application {

    public static void main(String[] args) {
        SpringApplication.run(AwsProject01Application.class, args);
    }
}
```

A primeira linha se refere ao pacote onde esse arquivo está e as duas linhas a seguir são os *imports* das bibliotecas utilizadas por ele.

Repare que na linha 7 há a declaração da classe `AwsProject01Application`, obviamente, com o mesmo nome do arquivo Java.

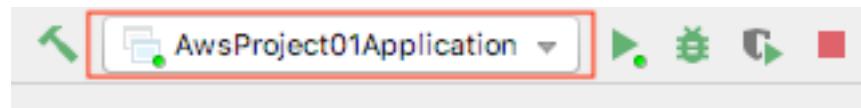
Dentro dessa classe há o método estático `main`, que é o ponto de entrada, ou seja, a aplicação começa a ser executada por ele.

A anotação `SpringBootApplication`, logo acima da declaração da classe, indica que essa é uma aplicação **Spring Boot**, e que tudo será preparado para tal. Posteriormente, novas anotações serão adicionadas nessa classe, bem como em outras para realização de configurações ou configuração de recursos e serviços da aplicação.



Essa forma de se utilizar anotações ao invés de configurações em arquivos `.xml` é uma das inúmeras vantagens de se utilizar Spring Boot para construção de aplicações Web em Java.

Para executar a aplicação de dentro do IntelliJ IDEA pela primeira vez, clique com o botão direito sobre o método `main` e escolha a opção *Run AwsProject01Application main()*. Isso fará com que uma configuração de execução seja criada no projeto do IntelliJ IDEA para essa aplicação, que pode ser observada no canto superior direito da janela do mesmo:



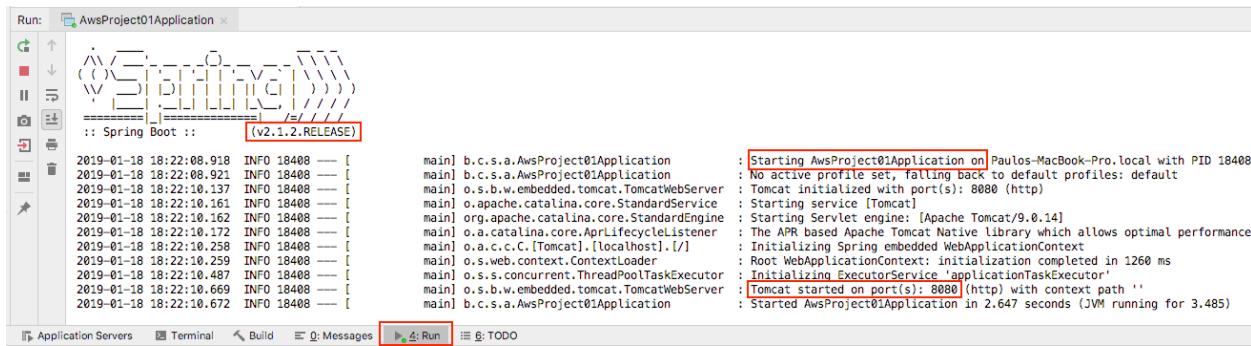
Configuração de execução criada

Repare que a execução da aplicação pode ser controlada pelos quatro botões no canto direito da imagem anterior. São eles, da esquerda para a direita:

- Executar a configuração de execução selecionada no combo à esquerda;
- Depurar a configuração de execução selecionada;
- Executar a configuração de execução selecionada, com cobertura de código;
- Para a execução ou depuração.

Por isso, caso queira parar, executar novamente ou depurar a aplicação, utilize esses controles para tais tarefas.

Agora perceba que, quando a aplicação é executada, a aba *Run* do IntelliJ IDEA aparece na parte inferior da tela, exibindo o log de execução da aplicação, como mostra a figura a seguir:



Nessa figura é possível perceber, logo no começo do log, que a versão utilizada do Spring Boot aparece abaixo de seu *banner* de inicialização, que nesse caso é a versão **2.1.2.RELEASE**.

Ainda nesse log, repare em sua penúltima linha, que a aplicação está sendo executada através do **Tomcat na porta 8080**. Essas são uma das configurações padrões de uma aplicação Spring Boot, ou seja:

- Utilizar o **Tomcat embarcado**, como servidor de aplicação. Isso significa que quando ela é executada, na verdade o Tomcat é executado antes, que a lança imediatamente;
- Exportar a **porta 8080** para requisições HTTP de entrada na aplicação.

Essas configurações e outras, principalmente qual porta HTTP utilizar, podem ser alteradas através do arquivo `application.properties`. Mais adiante e também nos capítulos seguintes, serão mostradas várias opções que podem ser configuradas nesse arquivo.

Por enquanto essa aplicação não faz nada de interessante. Ela apenas fica em execução indefinidamente.

5.5 - Gerando logs

Apesar dos logs de inicialização serem bastante informativos, como visto na seção anterior, é interessante, em muitas ocasiões, gerar suas próprias mensagens de log, seja para efeitos de depuração de alguma forma, ou seja simplesmente para registro de operações realizadas pela aplicação.

Para inserir seus próprios logs na classe `AwsProject01Application`, primeiramente é necessário acrescentar as duas dependências a seguir:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

Com essas duas bibliotecas é possível declarar uma instância estática do gerador de logs, logo no início da declaração da classe, como no trecho a seguir:

```
@SpringBootApplication
public class AwsProject01Application {

    private static final Logger log = LoggerFactory.getLogger(AwsProject01Application.class);
```

Com essa instância criada, é possível acessar métodos como `info` e `debug` para gerar mensagens nesses dois níveis, dependendo da necessidade, como no trecho a seguir:

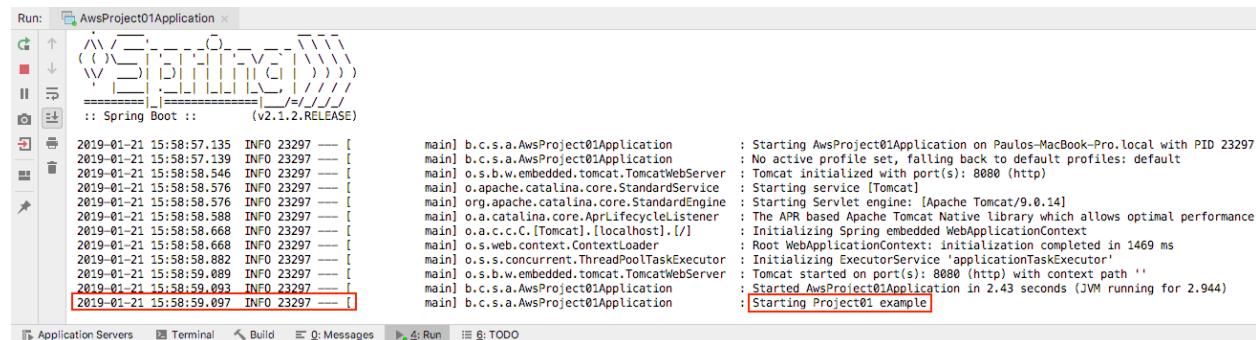
```
public static void main(String[] args) {
    SpringApplication.run(AwsProject01Application.class, args);

    log.info("Starting Project01 example");
}
```

Quando a aplicação for iniciada, essa mensagem irá aparecer com o nível `INFO`, conforme linha 4, que foi acrescentada no trecho acima.

Quando essa aplicação for executada dentro de um EC2 ou em um *container* em uma ECS, esse log, assim como os demais que estão aparecendo na inicialização da aplicação, irão aparecer no AWS CloudWatch.

Por enquanto, se a aplicação for executada no IntelliJ IDEA, essa linha aparecerá com os demais logs, como na figura a seguir:



```
2019-01-21 15:58:57.135 INFO 23297 --- [main] b.c.s.a.AwsProject01Application : Starting AwsProject01Application on Paulos-MacBook-Pro.local with PID 23297
2019-01-21 15:58:57.135 INFO 23297 --- [main] b.c.s.a.AwsProject01Application : No active profile set, falling back to default profiles: default
2019-01-21 15:58:58.546 INFO 23297 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2019-01-21 15:58:58.576 INFO 23297 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2019-01-21 15:58:58.576 INFO 23297 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.14]
2019-01-21 15:58:58.588 INFO 23297 --- [main] o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows optimal performance
2019-01-21 15:58:58.668 INFO 23297 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2019-01-21 15:58:58.668 INFO 23297 --- [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1469 ms
2019-01-21 15:58:58.882 INFO 23297 --- [main] o.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2019-01-21 15:58:59.089 INFO 23297 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2019-01-21 15:58:59.093 INFO 23297 --- [main] b.c.s.a.AwsProject01Application : Started AwsProject01Application in 2.43 seconds (JVM running for 2.944)
2019-01-21 15:58:59.097 INFO 23297 --- [main] b.c.s.a.AwsProject01Application : Starting Project01 example
```

Adicionando novos logs

Perceba que o nível do log aparece como `INFO`, que foi o método que foi utilizado para sua geração.

5.6 - Criando um controller para expor um endpoint

O projeto que foi criado no início desse capítulo possui uma dependência de nome `spring-boot-starter-web`, que pode ser vista no arquivo `build.gradle` na seção `dependencies`:

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.boot:spring-boot-starter-actuator'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

Esse conjunto de bibliotecas permite a criação de pontos de entrada na aplicação, para o tratamento de requisições HTTP, dentre outras coisas. Isso permite a criação de *endpoints* REST, para a implementação de serviços dessa natureza.

A ideia dessa seção é adicionar um serviço REST na aplicação que está sendo construída, para:

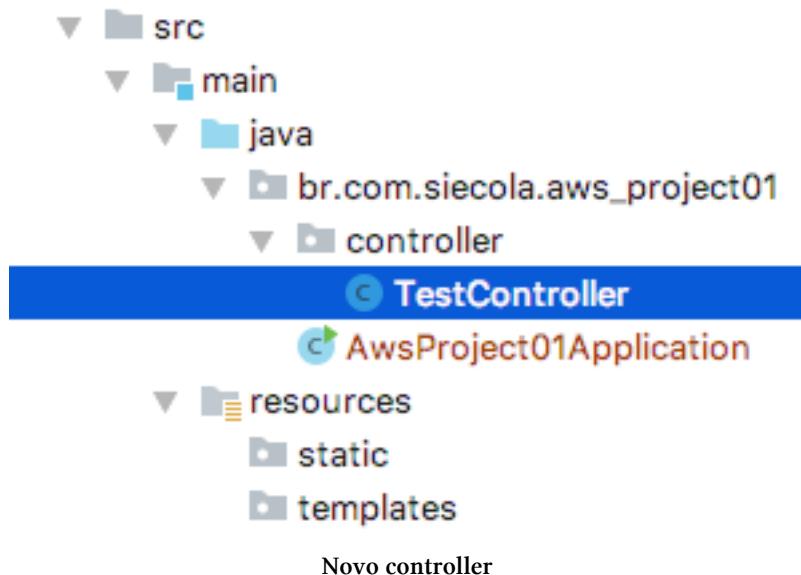
- Aprender alguns conceitos de serviços e operações REST;
- Como essas operações podem ser construídas em uma aplicação Spring Boot;
- Integrar o mecanismo de geração de logs nas operações;
- Permitir que a operação possa ser acessada de qualquer lugar da Internet.

O último item descrito acima se refere ao que será construído no próximo capítulo, onde a aplicação será instalada em uma instância EC2 na AWS.

Para começar, tendo o projeto aberto no IntelliJ IDEA, crie um novo pacote chamado `controller`. Nele, serão criadas as classes e os serviços REST da aplicação. Para isso, clique com o botão direito no pacote principal, de nome `br.com.siecola.aws_project01` e selecione a opção `New > Package`. Na janela que aparecer, digite `controller`.

Agora, no pacote que foi criado, crie uma classe chamada `TestController`. Para isso, clique com o botão direito nele e selecione a opção `New > Java Class`. Na janela que aparecer, digite `TestController`.

A estrutura do projeto deverá ficar como a figura a seguir:



O IntelliJ IDEA já cria a classe com sua estrutura padrão:

```
package br.com.siecola.aws_project01.controller;

public class TestController {
```

Agora é importante declará-la como uma classe especial, que representa um *controller* REST. Isso pode ser feito através de anotações do Spring, como mostra o trecho a seguir:

```
package br.com.siecola.aws_project01.controller;

import org.springframework.web.bind.annotation.RestController;

@RestController
public class TestController {
```

Veja que na linha 5 a anotação @RestController foi adicionada, para indicar que essa classe será um *controller* REST, ou seja, representará um serviço que conterá operações que poderão ser acessadas por requisições HTTP.

Pode-se definir um endereço base para que esse *controller* possa ser acessado, através de uma outra anotação, como mostra o trecho a seguir:

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/test")
public class TestController {
```

A anotação @RequestMapping que foi inserida define o mapeamento base para esse *controller*, ou seja, as operações que aqui forem criadas terão um endereço base definido pelo argumento que for passado a essa anotação, que nesse caso foi /api/test. Isso significa que, por exemplo, quando a aplicação for executada na máquina local de desenvolvimento, as operações poderão ser acessadas a partir do endereço <http://localhost:8080/api/test/>.

Além disso, é necessário criar pelo menos um método público, para representar um *endpoint* desse *controller*. Nele também será adicionado uma anotação para declarar as seguintes configurações:

- Método HTTP de acesso;

- Endereço de acesso, compondo a URL base do *controller*;
- Parâmetros a serem passados na URL.

Veja como ele deve ficar:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

@GetMapping("/dog/{name}")
public ResponseEntity<?> dogTest(@PathVariable String name) {

    return ResponseEntity.ok("Name: " + name);
}
```

A anotação `@GetMapping` define que o método HTTP para acessar essa operação deverá ser o GET, bem como seu endereço base: /dog. A última parte do endereço, ou seja `{name}`, diz que a operação deverá receber um parâmetro na URL, que será passado para o método quando ele for chamado.

Com a junção do mapeamento da classe e desse método, seu endereço de acesso deverá ficar da seguinte forma:

<endereço da aplicação>/api/test/dog/{name}

Mais adiante, quando a aplicação for testada utilizando o cliente Postman, essa composição do endereço ficará mais clara.

Na declaração do método, chamado `dogTest`, o parâmetro de entrada foi definido como:

```
@PathVariable String name
```

Esse tipo de declaração de parâmetros de entrada, em métodos que representam operações REST, define as seguintes características:

- O método receberá um parâmetro através da URL, definido pela anotação `@PathVariable`;
- O parâmetro será do tipo `String`, como definido em seu tipo;
- O nome do parâmetro de entrada, que nesse caso é `name`, será preenchido com o valor que vier na requisição, no local onde está definido a variável `{name}` da URL.

Além disso, a declaração do método também define seu tipo de retorno, que nesse caso foi `ResponseEntity<?>`. Esse é um tipo versátil para ser utilizado em métodos que definem operações REST. Com ele é possível definir o código de resposta da operação, bem como seu conteúdo.

O sinal `?` que foi utilizado no tipo de retorno do método significa que ele pode devolver uma informação de qualquer tipo, ou seja, um coringa. Mais adiante, quando operações mais complexas

forem criadas, para devolver tipos com objetos complexos, essa sintaxe ficará um pouco mais clara e poderá mostrar seu poder de versatilidade.

Dentro do método, há apenas uma linha, que é o retorno da operação:

```
return ResponseEntity.ok("Name: " + name);
```

Usando a classe `ResponseEntity`, é possível definir o código de retorno da operação, que nesse caso foi **HTTP 200 OK**, simplesmente chamando o método `ok` dessa classe. Esse método pode receber uma informação de qualquer tipo, inclusive uma instância de um objeto complexo. Nesse caso, foi passado somente uma string com o nome recebido como parâmetro de entrada do método.

Existem outros métodos da classe `ResponseEntity` que podem ser usados para definir o código HTTP de resposta, bem como o corpo da mensagem. Nos próximos capítulos, quando operações REST mais complexas forem criadas, ela voltará a ser utilizada, mostrando mais de suas funcionalidades.

Para finalizar a implementação desse método de exemplo, acrescente uma linha de log, para gerar uma mensagem contendo o valor do parâmetro passado, assim será possível registrar todos os acessos a essa operação. Para isso, crie um atributo estático do mecanismo de log, abaixo da declaração da classe e sua utilização, antes no retorno do método, como mostra o trecho a seguir, com toda a implementação da classe:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/test")
public class TestController {
    private static final Logger log = LoggerFactory.getLogger(TestController.class);

    @GetMapping("/dog/{name}")
    public ResponseEntity<?> dogTest(@PathVariable String name) {
        log.info("Test controller - name: {}", name);

        return ResponseEntity.ok("Name: " + name);
    }
}
```

Repare que na linha 12 foi criado o atributo estático para a geração de logs e na linha 15, a geração de uma mensagem contendo uma frase concatenada com o valor do atributo que a operação receber.

Para testar toda essa implementação, é necessário utilizar um cliente que possa gerar requisições HTTP para a aplicação. Como instruído no capítulo 2, o cliente que será utilizado durante todo esse livro para tarefas desse tipo será o Postman, por isso, abra essa aplicação em seu computador.

Em seguida, execute a aplicação no IntelliJ IDEA e aguarde até que ela esteja iniciada completamente.

Utilizando o Postman, configure uma aba de requisição conforme a figura a seguir:

The screenshot shows the Postman interface with a red box highlighting the request configuration area. The URL field contains `http://localhost:8080/api/test/dog/matilde`. Below the URL, the method is set to `GET`. The interface includes tabs for Params, Authorization, Headers, Body, Pre-request Script, and Tests. A table below the tabs shows a single parameter entry: `Key` under `KEY` and `Value` under `VALUE`.

Testando o novo controller

Nesse momento, para testar o novo *controller* que foi criado, é necessário configurar duas coisas no Postman:

- O método da requisição HTTP para `GET`, que é o que foi configurado na primeira operação do projeto, através da anotação `@GetMapping`
- O endereço da requisição para `http://localhost:8080/api/test/dog/{name}`, onde o parâmetro `name` pode variar, para passar uma informação diferente em cada requisição, se necessário.

Tendo o Postman configurado com esses parâmetros, basta pressionar o botão `Send` para que ele dispare uma requisição HTTP em direção à aplicação. Se tudo funcionar corretamente, a resposta deverá aparecer no Postman da seguinte forma:

The screenshot shows the Postman interface with the following details:

- Request URL: `http://localhost:8080/api/test/do`
- Method: `GET`
- Response URL: `http://localhost:8080/api/test/dog/matilde`
- Params tab selected:

KEY	VALUE
Key	Value
- Body tab selected:

Body	Cookies (1)	Headers (3)	Test Results
Pretty	Raw	Preview	Auto
1 Name: matilde			

Analisando resposta da operação

Veja na imagem que a String de resposta contém o valor do parâmetro que foi passado na requisição, exatamente o que foi feito no código que implementou a operação:

```
return ResponseEntity.ok("Name: " + name);
```

Ainda é importante observar os logs que foram gerados pela aplicação no IntelliJ IDEA, através da seguinte linha:

```
log.info("Test controller - name: {}", name);
```

Veja na figura a seguir como foi que ele apareceu no log da aplicação quando a operação foi invocada pelo Postman:

```

Run: AwsProject01Application
2019-01-27 16:03:03.846 INFO 28059 — [main] b.c.s.a.AwsProject01Application : Starting AwsProject01Application on Paulos-MacBook-Pro.local with PID 28059
2019-01-27 16:03:03.850 INFO 28059 — [main] b.c.s.a.AwsProject01Application : No active profile set, falling back to default profiles: default
2019-01-27 16:03:05.087 INFO 28059 — [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2019-01-27 16:03:05.115 INFO 28059 — [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2019-01-27 16:03:05.115 INFO 28059 — [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.14]
2019-01-27 16:03:05.115 INFO 28059 — [main] o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows optimal performance
2019-01-27 16:03:05.224 INFO 28059 — [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2019-01-27 16:03:05.224 INFO 28059 — [main] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 1263 ms
2019-01-27 16:03:05.444 INFO 28059 — [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2019-01-27 16:03:05.632 INFO 28059 — [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2019-01-27 16:03:05.632 INFO 28059 — [main] b.c.s.a.AwsProject01Application : Started AwsProject01Application in 2.43 seconds (JVM running for 2.897)
2019-01-27 16:03:05.638 INFO 28059 — [main] b.c.s.a.AwsProject01Application : Starting Project01 example
2019-01-27 16:21:09.213 INFO 28059 — [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2019-01-27 16:21:09.213 INFO 28059 — [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : : Initializing Servlet 'dispatcherServlet'
2019-01-27 16:21:09.219 INFO 28059 — [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : : Completed initialization in 6 ms
2019-01-27 16:21:09.245 INFO 28059 — [nio-8080-exec-2] b.c.s.a.controller.TestController : Test controller - name: matilde

```

Visualizando o log da operação

Faça outras requisições com o Postman, alterando o valor do parâmetro passado na URL, para ver que, tanto a resposta da operação, quanto o log gerado pela aplicação correspondem ao valor do parâmetro.

5.7 - Depurando a aplicação

Nem sempre os logs gerados pela aplicação ajudam a depurá-la. Às vezes é necessário interromper a aplicação enquanto ela está em execução e analisar com calma valores de variáveis e outras informações.

Com o IntelliJ IDEA é possível depurar a aplicação, permitindo que o desenvolvedor possa, além de executá-la passo-a-passo, observar várias informações relevantes ao contexto de execução atual.

Para fazer isso com o projeto de exemplo que está sendo construído nesse capítulo, faça um teste para depurar a execução do método que implementa a primeira operação que foi construída. Para isso, vá até a classe `TestController` e localize a seguinte linha:

```

16     @GetMapping("/dog/{name}")
17     public ResponseEntity<?> dogTest(@PathVariable String name) {
18         log.info("Test controller - name: {}", name);
19
20         return ResponseEntity.ok("Name: " + name);
21     }

```

Criando um ponto de parada

O intuito aqui é criar um ponto de parada da aplicação, para que quando ela for executada em modo de depuração, o IntelliJ IDEA pare a execução exatamente nesse ponto.

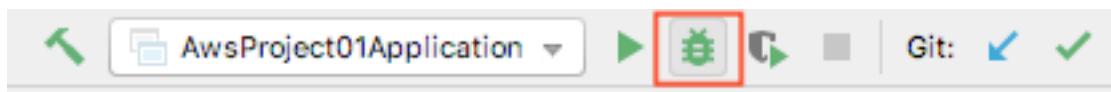
Então, para criar esse ponto de parada, clique logo do lado direito do número da linha que foi ressaltada na figura. O IntelliJ IDEA deve criar esse ponto de parada, sinalizando com um pequena bola vermelha, como mostra a figura a seguir:

```

16     @GetMapping("/dog/{name}")
17     public ResponseEntity<?> dogTest(@PathVariable String name) {
18         log.info("Test controller - name: {}", name);
19
20     }
21 }
```

Ponto de parada criado

Agora, para depurar a aplicação e entender como funciona o ponto de parada e o que ele pode oferecer, estando com a aplicação parada, clique no botão de depuração, localizado no canto superior direito do IntelliJ IDEA:



Depurando a aplicação

Isso fará com que a aplicação inicie em modo de depuração.

Quando a aplicação acabar de iniciar, repita o teste com o Postman, para disparar uma requisição na operação que foi criada na seção anterior. Isso fará com que a requisição comece a ser tratada pela aplicação, mas tenha sua execução paralisada no ponto da geração de log, exatamente onde o ponto de parada foi criado, como mostra a figura a seguir:

```

11     @RestController
12     @RequestMapping("/api/test")
13     public class TestController {
14         private static final Logger log = LoggerFactory.getLogger(TestController.class);
15
16         @GetMapping("/dog/{name}")
17         public ResponseEntity<?> dogTest(@PathVariable String name) { name: "matilde"
18             log.info("Test controller - name: {}", name); name: "matilde"
19
20         }
21     }
```

Debug: AwsProject01Application

Variables: this = {TestController@5468}, name = "matilde"

Analizando o ponto de parada

Veja que a execução do código é paralisada no ponto em que foi configurado para ser. Perceba também que o valor do atributo name do método aparece ao lado de sua declaração.

Isso permite que o desenvolvedor possa analisar o valor das variáveis e atributos durante a execução de um código, sem a necessidade de gerar logs a todo instante.

Para controlar a execução do código, utilize os controles de depuração localizados na aba Debug, na parte inferior do IntelliJ IDEA. Com eles é possível executar o código passo-a-passo, paralisar totalmente a aplicação ou liberar a execução para que a aplicação execute normalmente.

5.8 - Configurando a aplicação

Uma aplicação Spring boot pode ser configurada de várias formas. Aqui algumas opções mais comuns:

- Através do arquivo `application.properties`;
- Criando-se classes de configuração que são executadas na inicialização da aplicação;
- Através de parâmetros passados na linha de execução da aplicação.

Obviamente, ainda não é um momento oportuno para apresentar todas as configurações que podem ser feitas em uma aplicação Spring Boot. O objetivo aqui é apenas mostrar onde isso pode ser feito, ainda que seja algo simples.

Como exemplo, será mostrado como configurar a porta de execução da aplicação, que hoje está com seu valor padrão, ou seja 8080. Essa é uma configuração interessante para se mostrar, pois caso a aplicação esteja sendo executada em um ambiente onde essa porta não possa ser utilizada, é necessário então alterá-la.

O primeiro método a ser mostrado para alterar a porta de execução será através do arquivo `application.properties`. Por isso, dentro do IntelliJ IDEA, abra esse arquivo e adicione a seguinte linha a ele:

```
server.port=9090
```

A próxima vez que a aplicação for executada ele utilizará a porta 9090 para expor seus serviços, ou seja, as requisições HTTP terão que ser direcionadas para essa porta.

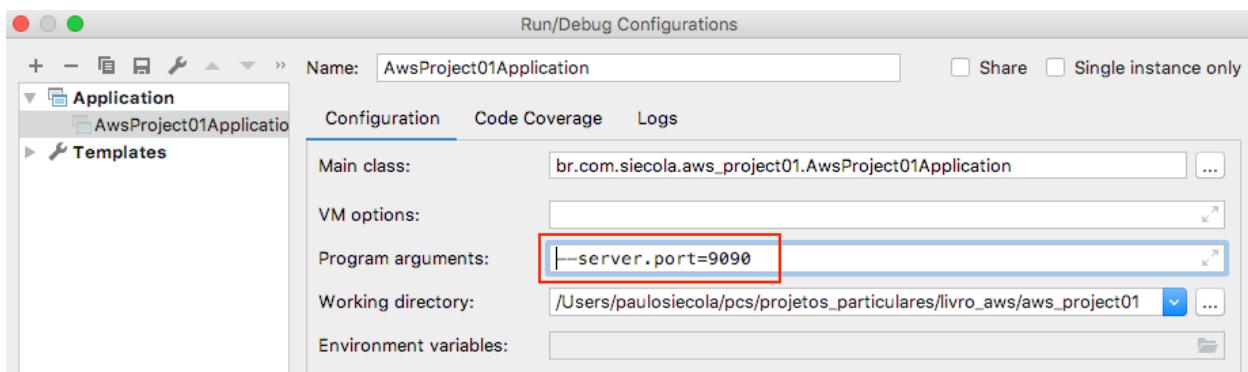
Na documentação oficial do Spring há uma lista com todas as configurações que podem ser feitas, que estão disponíveis nesse link: <https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html>

A outra forma de fazer essa mesma configuração é passar esse valor na linha de execução que chama a aplicação, como no exemplo a seguir:

```
java -jar application.jar --server.port=9090
```

No IntelliJ IDEA isso pode ser feito alterando a configuração de execução da aplicação. Para isso, acesse o menu Run > Edit Configuration.

Na janela que aparecer, configure o parâmetro Program arguments com o valor `--server.port=9090`, como mostra a figura a seguir:



Configurando a porta

Isso fará o mesmo efeito da configuração feita no arquivo `application.properties`, mas estará atrelado somente ao projeto do IntelliJ IDEA e não ao código do projeto em si.

A outra opção de configuração, que é através de classes criadas para serem executadas na inicialização da aplicação, é um pouco avançada para esse momento do livro. Por isso será deixada para ser mostrada em um momento adiante.

Com tantas opções de configuração, é interessante saber qual a ordem de precedência, que o Spring assume, caso haja mais de um lugar configurando o mesmo parâmetro. Essa ordem é a seguinte:

- Valores padrões que já estão embutidos no Spring Boot;
- Parâmetros passados por linha de comando;
- Configurações existentes no arquivo `application.properties`;
- Configurações feitas em classes executadas na inicialização da aplicação.

Essa é a ordem em que o Spring Boot avalia uma configuração, ou seja, se ela existir em todos os lugares listados acima, a que vai valer é a que aparece por último, ou seja, na classe que foi criada para ser executada na iniciação da aplicação.

É importante ter essa lógica de avaliação de configurações do Spring Boot sempre em mente, para que não haja confusão de qual valor prevalece, caso ele esteja em mais de um lugar.



Volte a porta de execução da aplicação para 8080, no arquivo `application.properties`.

5.9 - Criando serviços gerenciáveis pelo Spring

Em muitas situações é necessário criar um serviço reutilizável na aplicação, mas sem a necessidade de se criar uma instância dessa classe a todo instante em que ela for utilizada.

Seguindo alguns padrões de projeto, poderia-se utilizar o padrão *singleton*, que basicamente implementa essa ideia. Isso seria uma opção se a aplicação não fosse baseada no *framework* Spring, que já possui mecanismos mais eficientes e elegantes para isso.

Seguindo essa ideia, para se criar um serviço dentro de uma aplicação Spring Boot, basta utilizar a anotação `@Service` na declaração de uma classe.

Como exemplo, crie um novo pacote chamado `service` no projeto e em seguida, crie uma nova classe chamada `TestService`:

```
package br.com.siecola.aws_project01.service;

import org.springframework.stereotype.Service;

@Service
public class TestService {
```

Essa classe especial, chamada então de *service*, pode possuir métodos públicos e realizar operações comuns a várias partes da aplicação, como por exemplo:

```
public boolean isUserMatilde(String name) {
    if ("matilde".equals(name)) {
        return true;
    } else {
        return false;
    }
}
```

Veja o exemplo de como essa classe poderia ser utilizada de forma elegante, como um *service* gerenciado pelo Spring Boot, sendo inserido na classe `TestController` através da anotação `@Autowired`:

```

@RestController
@RequestMapping("/api/test")
public class TestController {
    private static final Logger log = LoggerFactory.getLogger(TestController.class);

    @Autowired
    private TestService testService;

```

Repare nas linhas 6 e 7 que agora o *controller* TestController possui uma instância do *service* TestService, simples assim!

A anotação @Autowired injeta uma instância de TestService nessa classe, sem a necessidade de se criar uma a todo instante com a instrução new Java. Isso faz com que o Spring Boot crie essa instância, ou a destrua, da melhor forma que ele achar necessário, sem que o desenvolvedor se preocupe em saber se existem ou não muitas instâncias dela em atividade na aplicação.

Para utilizar essa instância, basta fazê-lo da forma tradicional, como se o objeto tivesse sido criado com a instrução new do Java, como no exemplo a seguir dentro do método dogTest:

```

@GetMapping("/dog/{name}")
public ResponseEntity<?> dogTest(@PathVariable String name) {
    log.info("Test controller - name: {}", name);

    log.info("Is matilde? {}", testService.isUserMatilde(name));

    return ResponseEntity.ok("Name: " + name);
}

```

Veja na linha 5 do trecho acima que a instância do *service* é utilizada, chamando-se o método que nele foi criado.

Obviamente esse é apenas um exemplo para demonstrar como um *service* deve ser utilizado, mas imagine que essa validação tenha que ser feita em vários pontos do projeto, utilizando-se apenas um instância da classe TestService, para que a aplicação não consuma memória de forma descontrolada. Para isso, o desenvolvedor teria que escrever mais código, com mais cuidado em sua implementação. Utilizando Spring Boot, é possível delegar toda essa responsabilidade para ele.



Services, assim como controllers, não devem possuir atributos de classe que armazenem estados, pois eles são gerenciados pelo Spring e têm apenas um instância.

Essa técnica será amplamente utilizada nesse livro, para a construção de serviços com as mais variadas aplicações.



É importante lembrar que o código do projeto está no GitHub e pode ser acessado [aqui](#)².

5.10 - Conclusão

Essa foi uma breve introdução sobre como construir uma aplicação Web em Java utilizando o Spring Boot. Como dito algumas vezes nesse capítulo, esse é um *framework* muito grande e cheio de funcionalidades e configurações. À medida em que essa e outras aplicações forem evoluindo, mais será mostrado sobre esse poderoso *framework*.

No próximo capítulo essa aplicação será instalada em uma instância EC2 que será criada na AWS. Obviamente, muitos conceitos serão introduzidos e detalhados para que isso se realize.

²https://github.com/siecola/aws_project01

6 - Amazon Elastic Compute Cloud

O Amazon Elastic Compute Cloud ou apenas EC2, é um serviço oferecido pela AWS para criação de instâncias de máquinas virtuais com capacidade de processamento, memória e disco configuráveis, de acordo com a necessidade da aplicação.

Este capítulo apresentará conceitos importantes para utilização desse serviço, bem como sua configuração e dicas para sua utilização.

Ao final, o leitor será capaz de criar sua própria instância e instalar a aplicação que foi desenvolvida no capítulo anterior, para que ela fique exposta na Internet. Além disso, será possível monitorar a execução dessa aplicação através de logs, consumo de memória e CPU, tudo isso utilizando ferramentas oferecidas pela própria AWS.

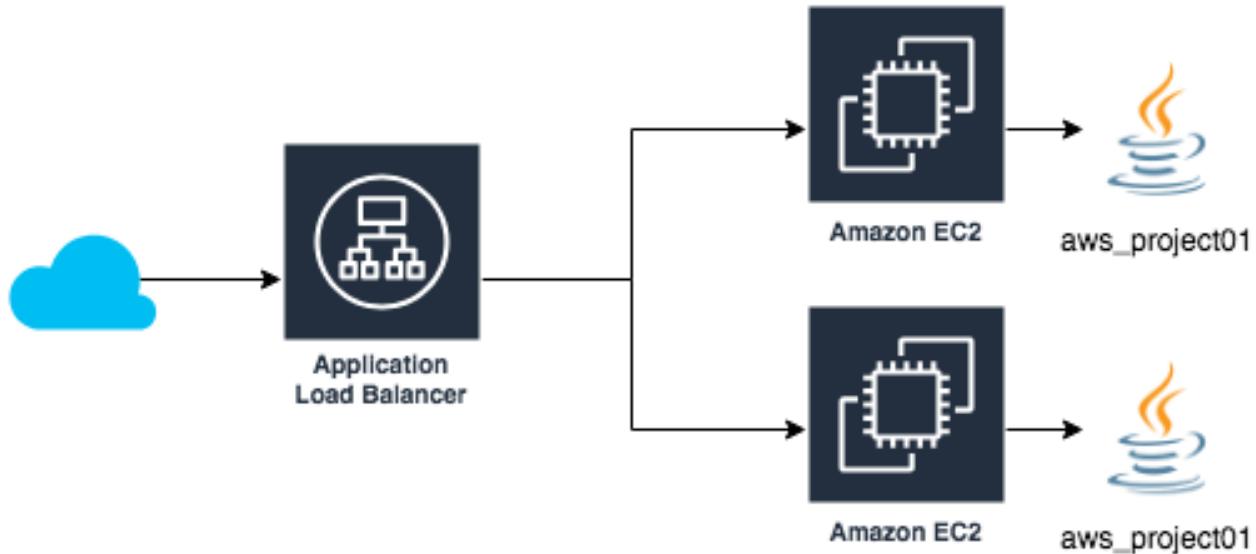


Diagrama final

Também será criado um *application load balancer*, que irá dividir a carga de requisições entre as duas instâncias EC2 que serão criadas, cada uma contendo a aplicação `aws_project01`.

6.1 - Conceitos básicos do EC2

A criação de máquinas virtuais talvez seja um dos passos mais básicos para a criação de uma aplicação a ser hospedada na nuvem, visto que é necessário que ela possua alguma infraestrutura mínima para a sua execução.

A utilização de máquinas virtuais, num ambiente de computação em nuvem, é o que viabiliza diversos fatores característicos dessa tecnologia, como escalabilidade, elasticidade e, de certa forma, alta disponibilidade.

Com o AWS EC2, o administrador de um sistema pode criar instâncias de máquinas virtuais para hospedar cada parte desse sistema, sem a necessidade de gerenciar os recursos de hardware necessários para elas. Além disso, ele pode criar ou apagar instâncias à medida que seu sistema demandar mais ou menos poder de processamento.

Esse trabalho de gerenciamento das instâncias EC2, seja para criação, inicialização, paralização ou até mesmo exclusão, pode ser feito manualmente, através do console da AWS. Porém, tal trabalho também pode ser feito através de mecanismos automáticos, configurados para entrar em ação a partir de certos parâmetros, como por exemplo a quantidade de requisições que a aplicação recebe ou a quantidade de memória consumida por ela.

Durante o processo de criação de uma instância virtual para ser executada no EC2, o administrador do sistema pode escolher alguns parâmetros para melhor defini-la. A seguir um breve resumo de alguns desses pontos:

- a) **Tipos de instâncias:** as instâncias EC2 são classificadas em diversos tipos, que definem características como poder de processamento e quantidade de memória. Sua escolha depende da demanda de recursos que o sistema, a ser executado nessa instância, exigirá.
- b) **Sistema operacional:** durante o processo de criação das instâncias EC2 é possível escolher qual o sistema operacional será utilizado, dentre algumas distribuições Linux e Windows.
- c) **Tipo de sistema de armazenamento:** o tipo de sistema de arquivo a ser escolhido depende obviamente do tipo de aplicação. A AWS oferece algumas opções desde pequenos discos de estado sólido, de vários tipos, até discos com grandes capacidades.
- d) **Localização geográfica e disponibilidade:** assim como a maioria dos recursos que podem ser criados na AWS, é possível escolher qual será a localização geográfica que a instância será criada, bem como a forma como ela será disponibilizada para outras regiões em casos de catástrofes.
- e) **Características de rede para comunicação:** é inevitável que uma instância criada no EC2 tenha que se comunicar com outro recurso da AWS ou mesmo com a Internet. Por esse motivo, é possível configurar parâmetros para liberar ou restringir o acesso à EC2, bem como o que ela pode acessar.
- f) **Características de escalabilidade automática:** como dito anteriormente, é possível definir configurações para que uma instância EC2 se replique, caso haja a necessidade de atender uma quantidade de requisições superior à suportada por apenas uma instância. Dessa forma, o tráfego é dividido entre todas as máquinas disponíveis, sem deixar de atender nenhuma requisição.
- g) **Imagens:** as instâncias são criadas a partir de uma imagem, que pode conter simplesmente um sistema operacional de uma determinada versão ou até sistemas mais complexos já pré-instalados. As imagens também podem ser construídas pelo administrador do sistema, já com a aplicação instalada e com todo o ambiente configurado, dessa forma não é necessário repetir todo o processo de instalação e configuração para cada instância que for criada para o mesmo propósito.

h) Pague pelo uso: assim como outros recursos AWS, as instâncias EC2 são cobradas pelo uso, dependendo do tipo de cada uma. Dessa forma, é importante saber dimensionar com cautela os recursos que serão utilizados, para que não haja cobrança desnecessária por recursos ociosos.

Obviamente, não é tão simples definir precisamente o que escolher para cada aplicação, uma vez que muitas variáveis definem as necessidades exatas de um sistema, para ser executado no EC2. Porém, é possível começar definindo um conjunto pequeno de parâmetros, como por exemplo o tipo da instância, o que já traz a quantidade de memória e CPU. À medida que as necessidades de uma aplicação forem melhor entendidas, é possível refinar melhor tais configurações.

6.2 - Unidades de CPU e memória

Para entender e melhor escolher os tipos de instância para uma determinada aplicação, é necessário conhecer dois parâmetros importantes: unidades de CPU e memória.

a) Memória: cada instância é criada com uma quantidade de memória RAM que pode ser utilizada pelo sistema operacional e pelas aplicações que foram instaladas nele. Esse valor é expresso em *gigabytes*.

b) Unidades de CPU: esse é um valor de referência para definir a capacidade de processamento de uma instância. Obviamente, cada instância possui um processador virtual equivalente a um processador físico de um computador real. Porém, para que se possa estabelecer um valor comparativo entre instâncias, no que se refere a capacidade de processamento, é utilizado então a unidade de CPU, que pode ter valores como 512, 1024, 4096, etc.

c) vCPU: esse é o número inteiro que define quantas unidades de CPU uma instância possui. Cada vCPU corresponde a 1024 unidades de CPU. Dessa forma, se um determinado tipo de instância A possui 2 vCPU, isso significa que ela possui 2048 unidades de CPU. Se outro tipo B possui 4 vCPU, então ela possui 4096 unidades CPU. Logo, a instância do tipo B tem mais poder de processamento que a instância do tipo A.

6.3 - Tipos de instâncias de EC2

A AWS possui vários tipos de instâncias que podem ser utilizadas para a criação de uma EC2. Os tipos estão divididos em propósitos de uso, como:

- uso geral;
- computação intensa;
- otimizadas para uso intenso de memória;
- processamento gráfico;
- armazenamento.

Dentre esses tipos, existem subdivisões com variações de memória, CPU, armazenamento e capacidade de comunicação via rede.

Durante o processo de criação de uma EC2 é necessário escolher o tipo de sua instância de acordo com as necessidades da aplicação que ela executará. É importante lembrar, como descrito na seção 6.2, a EC2 é cobrada pelo uso, mas também com o seu tipo, ou seja, instâncias mais poderosas são mais caras.

No site da AWS é possível ver uma lista completa e atualizada dos tipos de instâncias disponíveis: <https://aws.amazon.com/ec2/instance-types/>

6.4 - Criando um instância EC2

Essa seção detalha os passos para criar uma instância EC2 através do console da AWS, por isso, entre com sua conta para iniciar o processo.

Estando no console da AWS, acesse a opção Services > EC2 no menu principal. O *dashboard* de controle das instâncias será exibido nessa tela:

The screenshot shows the AWS EC2 Dashboard. On the left, there's a sidebar with options like EC2 Dashboard, Events, Tags, Reports, Limits, and INSTANCES (which is expanded to show Instances and Launch Templates). The main area is titled "Resources" and displays the message: "You are using the following Amazon EC2 resources in the US West (Oregon) region:". Below this, it lists: 0 Running Instances, 0 Dedicated Hosts, 0 Volumes, 0 Key Pairs, and 0 Placement Groups. At the bottom of the dashboard, it says "Dashboard de instâncias EC2".

Como todo recurso AWS, é necessário escolher em qual região ele será criado. Como dito no capítulo 4, todos os recursos criados durante esse livro deverão estar na região US East (N. Virginia) - us-east-1. Por isso, vá no canto superior direito da página e selecione essa região:



US East (N. Virginia)

US East (Ohio)

US West (N. California)

Selecionando a região



Tenha certeza de que sempre está nessa região ao entrar no console da AWS, pois os recursos que forem criados aqui não estarão acessíveis em outras regiões.

Agora vá no menu lateral esquerdo e selecione a opção Instances. A página que será exibida é a que mostra uma tabela listando todas as instâncias existentes, assim como várias informações sobre seu estado.

Para começar o processo de criação de uma instância, clique no botão Launch instance, localizado no canto superior esquerdo. O primeiro passo é a escolha da imagem a ser utilizada pela instância. Nesse primeiro momento, escolha a imagem **Ubuntu Server 18.04 LTS**, ou uma versão superior do mesmo sistema operacional:



Escolhendo a imagem da primeira instância EC2

Tenha certeza de que a opção 64-bit (x86) está selecionada para essa imagem. Após isso, clique no botão Select, para passar para o próximo passo de criação.

O segundo passo é a escolha do tipo da máquina que executará a instância. Nessa página há uma tabela com os tipos de máquinas disponíveis e com os detalhes de cada uma, no que se refere a vCPU, memória e disco. Lembre-se sempre que uma escolha com recursos mais poderosos resulta em um custo maior de operação.

Nesse primeiro exemplo, escolha a opção **t2.micro**, que será mais do que suficiente para o propósito de executar a aplicação que foi desenvolvida no capítulo 5:

	Family	Type	vCPUs	Memory (GiB)
<input type="checkbox"/>	General purpose	t2.nano	1	0.5
<input checked="" type="checkbox"/>	General purpose	t2.micro Free tier eligible	1	1

Escolhendo o tipo da instância EC2

Essa máquina escolhida possui as seguintes características:

- 1 vCPU, o que significa 1024 unidades de CPU;
- 1 GB de RAM;
- 1 SSD de 8 GB;
- Capacidade baixa a moderada de rede.

As demais opções nos passos seguintes oferecem opções de configurações avançadas que não serão vistas no momento. Por isso, clique no botão `Review and Launch` para deixar que a AWS configure tais opções com valores padrão.

A tela que aparece se refere ao último passo do processo, que é um resumo do que foi configurado. Em questões de configuração, é somente isso. Agora o que falta é providenciar as credenciais de acesso. Para isso, clique no botão `Launch` para abrir a janela de diálogo que se refere a esse passo, explicado na próxima sessão desse capítulo.

6.5 - Acessando a instância via SSH

As credenciais de acesso são um par de chaves (pública e privada) que permitem que clientes SSH acessem remotamente o terminal do sistema operacional das instâncias.

No último passo de criação da instância, logo após o botão `Launch` ter sido pressionado, uma janela de diálogo aparece para que tais chaves sejam configuradas:

Select an existing key pair or create a new key pair X

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance.

Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more about [removing existing key pairs from a public AMI](#).

The screenshot shows a dialog box with the title "Select an existing key pair or create a new key pair". At the top left is a "Create a new key pair" button. Below it is a "Key pair name" label followed by an input field containing a single vertical bar character. At the bottom right is a "Download Key Pair" button.

Criando as chaves de acesso

Nessa janela, selecione a opção Create a new key pair e digite um nome para as chaves, por exemplo: key_pair_01.

Em seguida, clique no botão Download Key Pair para baixar o arquivo. O arquivo terá a extensão .pem.



Salve o arquivo em um **local seguro**, pois com ele é possível acessar o terminal da sua instância.

Para finalizar, clique no botão Launch Instances para fazer com que a instância seja criada e entre em execução. A próxima página mostra o status do processo de criação da instância. Se tudo deu certo, clique no botão View Instances para ir para o *dashboard*, onde todas as instâncias são exibidas.

As instâncias existentes são exibidas em uma tabela, que mostra várias informações sobre cada uma delas, como por exemplo:

- **Name:** nome da instância, que pode ser configurado na própria tabela;
- **Instance ID:** identificação única da instância;
- **Instance Type:** tipo da máquina escolhida para criar a instância;
- **Instance State:** estado atual da instância, que será detalhado em uma seção mais adiante desse capítulo;
- **Public DNS:** o domínio público de acesso a essa instância, através da Internet. Esse valor será utilizado para se conectar a essa instância através de um cliente SSH.

Quando a instância é selecionada nessa tabela, um painel é mostrado na parte inferior da página, com informações mais detalhadas, como:

- Descrição de todos os parâmetros de execução;
- Status de alarmes e de verificações de execução da instância;
- Gráficos de monitoramento de uso de rede e CPU.

Perceba que na tabela onde a instância é listada, na coluna Instance State, há a informação running, que significa que a instância está em execução e pronta para ser utilizada. Para isso, é necessário utilizar as chaves de acesso que foram geradas no final do processo de criação da instância, armazenadas no arquivo .pem que foi baixado. Com essas chaves, será possível acessar seu console e interagir com ela.

O objetivo aqui é fazer com que a máquina de desenvolvimento possa acessar o console da instância que está em execução na AWS. Isso será necessário para instalar a aplicação que foi desenvolvida no capítulo anterior.

Os passos necessários para **acessar a máquina através de SSH** são muito bem descritos nesses tutoriais da própria AWS, de acordo com o sistema operacional cliente a ser utilizado:

- **Windows:** https://docs.aws.amazon.com/pt_br/AWSEC2/latest/UserGuide/putty.html
- **Linux ou Mac:** https://docs.aws.amazon.com/pt_br/AWSEC2/latest/UserGuide/AccessingInstancesLinux.htm

É necessário configurar a máquina de desenvolvimento, seguindo os passos descritos nos tutoriais citados anteriormente, para se ter acesso à instância EC2, algo que será necessário no decorrer desse capítulo.

No Windows, utilize o programa PuTTY para acessar o console da máquina e o WinSCP para transferência de arquivos, necessário na seção 6.6 para copiar o arquivo JAR da aplicação aws_project01 para dentro da instância EC2.

No Linux ou no Mac, o próprio terminal será utilizado.

Para qualquer que seja o sistema operacional do cliente a ser utilizado para acessar o console da instância, serão necessárias as seguintes informações:

- **ID da instância:** esse é o valor informado na segunda coluna da tabela que lista as instâncias no console da AWS;
- **DNS público da instância:** esse valor está presente na oitava coluna da mesma tabela que lista as instâncias;
- **Usuário de acesso:** no caso da instância que foi criada, o usuário de acesso é ubuntu;
- **Arquivo de chaves:** esse é o arquivo com extensão .pem que foi baixado no início dessa seção.

Acesse o console da instância que foi criada, utilizando os passos descritos nos links que foram fornecidos anteriormente, de acordo com o sistema operacional da máquina de desenvolvimento.

6.6 - Instalando uma aplicação numa instância EC2

Depois de ter acesso o console da instância que foi criada, será possível instalar a aplicação que foi desenvolvida no capítulo anterior, porém é necessário **instalar o Java Runtime Environment (JRE)** antes dessa tarefa, para que a aplicação possa ser executada.

Para começar a instalação do JRE, estando no console da instância criada na AWS, digite o seguinte comando:

```
sudo apt install openjdk-8-jre-headless
```

Isso fará com que o processo de instalação do JRE 8 comece. Após a finalização desse processo, digite o comando a seguir para verificar se agora a máquina possui o Java instalado:

```
java -version
```

O resultado desse comando irá informar a versão do Java que foi instalado, dentre outras informações, como mostra o trecho a seguir:

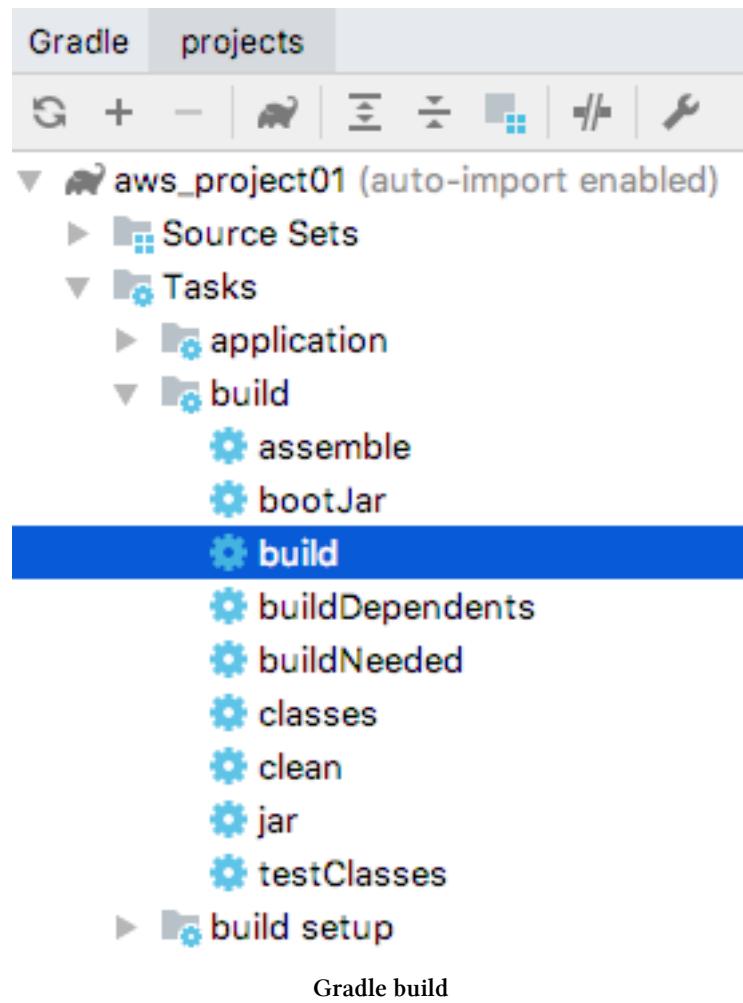
```
openjdk version "1.8.0_191"
OpenJDK Runtime Environment (build 1.8.0_191-8u191-b12-2ubuntu0.18.04.1-b12)
OpenJDK 64-Bit Server VM (build 25.191-b12, mixed mode)
```

Para que a aplicação seja executada nessa máquina, é necessário criar o arquivo .jar executável dentro do projeto que foi criado no capítulo anterior. Para isso, abra-o no IntelliJ IDEA e adicione o seguinte trecho no final do arquivo build.gradle:

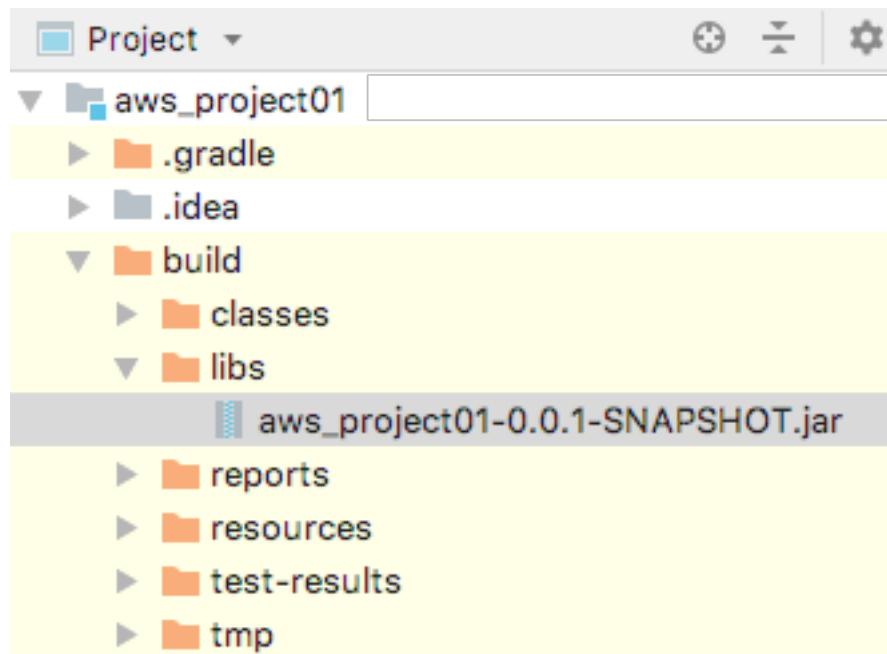
```
bootJar {
    launchScript()
}
```

Isso fará com que o arquivo a ser gerado nos próximos passos possa ser executado diretamente, como um binário executável comum. E para gerar tal arquivo, execute os seguintes passos no IntelliJ IDEA:

- Vá no menu `View > Tool Windows > Gradle`. Essa ação abre a janela de tarefas que podem ser executadas pelo Gradle;
- Na janela que abrir, dê um duplo clique na opção `build` dentro de `Tasks > build`:



Esse comando fará com que o arquivo executável .jar da aplicação seja criado dentro da pasta aws_project01/build/libs:



Arquivo jar da aplicação

Esse é o arquivo que deverá ser copiado para dentro da máquina que foi criada na AWS, para que possa ser executado dentro dela. Para isso, proceda com os seguintes passos:

- Localize o arquivo `.jar` da aplicação. Isso pode ser feito através do IntelliJ IDEA, clicando com o botão direito sobre ele e selecionando a opção a opção para abrir o navegador de arquivos do sistema operacional;
- Utilizando o WinSCP, para Windows, ou o terminal para Linux e Mac, copie esse arquivo `.jar` para a pasta `/home/ubuntu` da máquina na AWS;

Para verificar se o arquivo foi realmente copiado, dentro do terminal da instância, execute o comando a seguir dentro da pasta `/home/ubuntu`:

```
ls -la
```

O arquivo `aws_project01-0.0.1-SNAPSHOT.jar` deverá aparecer listado no resultado desse comando.

Agora é possível finalmente executar a aplicação, com o seguinte comando:

```
java -jar aws_project01-0.0.1-SNAPSHOT.jar
```

Repare que o *banner* do Spring Boot aparece e logo em seguida os logs da aplicação em execução, da mesma forma como aconteceu quando a aplicação foi executada dentro do IntelliJ IDEA, como mostra a figura a seguir:

Executando a aplicação na EC2

Apesar da aplicação já estar em execução, não existe uma regra de configuração de tráfego de entrada da instância que permita que requisições provenientes da Internet possam acessar a porta 8080 da aplicação, que é a que foi configurada para ela. Por padrão, somente a porta de acesso via SSH, que é a 22, é liberada no momento da criação da instância.

Para liberar a porta 8080 e permitir que requisições externas possam acessar a aplicação, vá no console da AWS, clique sobre a instância que foi criada e acesse a aba Description.

Nessa aba, localize as informações de Security groups, como mostra a figura a seguir:

Instance: **i-06137dbb1cdbf19b8 (ec2_01)** Public DNS: **ec2-**

Description Status Checks Monitoring Tags

Instance ID	i-06137dbb1cdbf19b8
Instance state	running
Instance type	t2.micro
Elastic IPs	
Availability zone	us-east-1b
Security groups	launch-wizard-1 , view inbound
Scheduled events	No scheduled events
Security groups	

Clique no primeiro grupo, como ressaltado na imagem anterior. Isso fará com que a tela de configuração de segurança desse grupo, no qual a instância que foi criada pertence, aparece com suas opções.

Nessa tela, clique na aba Inbound para visualizar as regras de tráfego de entrada desse grupo. Para adicionar uma nova, clique no botão Edit. Na tela que aparecer, clique no botão Add rule para configurar uma nova regra com as seguintes informações:

- **Type:** Custom TCP rule;
- **Protocol:** TCP
- **Port Range:** 8080
- **Source:** Custom e endereço 0.0.0.0/0

A nova regra deverá ficar como a figura a seguir:

Edit inbound rules

Type	Protocol	Port Range	Source
Custom TCP Rule	TCP	8080	Custom 0.0.0.0/0
SSH	TCP	22	Custom 0.0.0.0/0

Nova regra de entrada

Tendo configurado esses campos, clique no botão Save. A nova regra de tráfego de entrada já está valendo.

Para verificar se tudo está realmente funcionando, utilize o Postman para acessar o *endpoint* criado dentro da aplicação, da mesma forma como foi feito no capítulo anterior, porém substituindo o treço localhost do endereço da aplicação pelo DNS público da instância, o mesmo utilizado para acessá-la via SSH. Veja na figura a seguir como deve ficar, com um exemplo de DNS público de uma instância:

GET http://ec2-54-167-121-205.compute-1.amazonaws.com:8080/api/test/dog/matilde

GET http://**ec2-54-167-121-205.compute-1.amazonaws.com**:8080/api/test/dog/matilde

Params Authorization Headers Body Pre-request Script Tests

Acessando pelo Postman

Substitua o trecho realçado em vermelho na figura anterior pelo DNS público da instância que está em execução na AWS, que é o mesmo endereço que foi utilizado para acessá-la via SSH.

Faça alguns testes com o Postman, colocando valores diferentes para o último parâmetro da requisição e acompanhe os logs da aplicação, que devem reagir imprimindo logs de informação, como no exemplo a seguir:

Requisições e logs

Pronto! Agora a aplicação já está em execução dentro da instância criada na AWS, porém ainda é necessário configura-la para ser executada sozinha toda vez que a máquina for ligada, sem que alguém tenha que entrar em seu console e executá-la manualmente. Isso pode ser feito criando um serviço no sistema operacional da instância, através dos comandos a seguir:

```
chmod a+x aws_project01-0.0.1-SNAPSHOT.jar
```

Esse comando faz com que o arquivo .jar possa ser iniciado diretamente, como se fosse um arquivo executável.

```
sudo ln -s /home/ubuntu/aws_project01-0.0.1-SNAPSHOT.jar /etc/init.d/aws_project01
```

Isso fará com que um serviço, chamado `aws_project01` seja criado no sistema operacional e seja executado toda vez que a máquina for ligada.

Para verificar se o serviço foi criado corretamente, execute o comando a seguir para recarregar o sistema de *daemons* e reconhecer o novo serviço que foi criado:

```
sudo systemctl daemon-reload
```

Agora, basta executar o comando a seguir para subir a aplicação

```
sudo service aws_project01 start
```

E para onde foram para os logs? Eles ficam na pasta `/var/log/aws_project01.log` e para monitorá-los durante a execução da aplicação, basta executar o seguinte comando:

```
tail -f /var/log/aws_project01.log
```

Isso fará com que o console fique constantemente mostrando a execução da aplicação através de seus logs.

E o que acontece se a instância for reiniciada? A aplicação não voltará a ser executada sozinha, para isso é necessário realizar um último comando para fazer com que o serviço que foi criado possa iniciar assim que o sistema operacional concluir seu boot. Para isso, execute o seguinte comando:

```
sudo update-rc.d aws_project01 defaults 99
```

Para testar se tudo está funcionando, volte no console da AWS e clique com o botão direito sobre a instância e selecione no menu a opção Instance State > Reboot. Isso fará com que a instância seja reiniciada, da mesma forma como acontece com um computador.

Após alguns segundos, a instância voltará a execução normal e o serviço certamente foi iniciado pelo próprio sistema operacional. Para verificar isso, acesse novamente o console da instância através do protocolo SSH (PuTTY no Windows ou terminal no Linux e no Mac) e digite o seguinte comando:

```
sudo service aws_project01 status
```

O resultado desse comando trará informações sobre o serviço, mas principalmente que ele está ativo e em execução, como pode ser visto na figura a seguir:

```
ubuntu@ip-172-31-92-33:~$ sudo service aws_project01 status
● aws_project01.service - LSB: aws_project01
  Loaded: loaded (/etc/init.d/aws_project01; generated)
  Active: active (running) since Sun 2019-03-10 17:38:39 UTC; 7min ago
    Docs: man:systemd-sysv-generator(8)
   Process: 831 ExecStart=/etc/init.d/aws_project01 start (code=exited, status=0/SUCCESS)
     Tasks: 25 (limit: 1152)
    CGroup: /system.slice/aws_project01.service
           └─893 /usr/bin/java -Dsun.misc.URLClassPath.disableJarChecking=true -jar /home/
```

Serviço em execução

A outra forma de verifica que o serviço está em execução é obviamente acessá-lo pelo Postman, repetindo uma requisição a ele.

Tudo certo! Agora o serviço aws_project01 estará em execução sempre que a instância estiver ligada, voltando a funcionar mesmo se ela for reiniciada.

6.7 - Monitorando a instância EC2

A aba Monitoring, localizada no parte inferior da tabela de instâncias, exibe dados interessantes sobre sua execução, como por exemplo:

- Utilização de CPU;
- Vários parâmetros de acesso a disco;
- Gráficos de acesso de rede.

Esses gráficos podem sinalizar a saúde da instância, principalmente em termos de utilização de CPU. É importante observar tais gráficos, principalmente para conseguir entender o correto dimensionamento dos recursos utilizados para a execução da aplicação.

6.8 - Estados da instância

É possível controlar o estado de uma instância EC2, fazendo com que ela fique em execução, reinicie ou mesmo seja desligada. Tais estados podem ser controlados através do menu Instance State, quando clica-se com o botão direito sobre uma instância. Esse menu possui os seguintes comandos:

- **Start:** inicia uma instância, se ela estiver parada. É o equivalente a ligar um computador convencional;
- **Stop:** interrompe a execução de uma instância. É o equivalente a desligar um computador convencional;
- **Reboot:** reinicia uma instância. É o equivalente a reiniciar um computador convencional;
- **Terminate:** essa ação desliga e apaga uma instância e seus discos associados.

Quando uma instância está parada, não há consumo de recursos, por isso é seguro e recomendável sempre acionar a opção **Stop** quando não se deseja mais utilizar uma instância, evitando assim gastos desnecessários com horas de instância.

6.9 - Criando imagens e templates customizados para EC2

Criar a instância, configurá-la, instalar e configurar a JRE e a aplicação foram trabalhos consideravelmente longos. Se houvesse a necessidade de se criar outra instância para a mesma aplicação, por exemplo para balanceamento de carga, tudo isso teria que ser feito novamente. Porém existe um atalho, que é a criação de imagens e *templates* a partir de uma instância já pré-configurada.

6.9.1 - Criando uma imagem

A imagem é uma cópia da instância, ou seja, tudo o que foi configurada nela, em termos de bibliotecas, aplicações e scripts, são replicados para essa imagem, ou seja, uma cópia de seu disco. Com ela, podem ser criados *templates* para o lançamento de outras instâncias.

A instância que foi criada nesse capítulo foi a partir de uma imagem pré-configurada com o Ubuntu Server e com as configurações padrões para que somente esse sistema operacional fosse executado.

Porém na instância que foi criada, a aplicação `aws_project01` foi instalada e configurada para entrar em execução assim que o sistema operacional fosse executado. Seria muito interessante se esse trabalho não tivesse que ser repetido para a criação de outra instância para a execução dessa mesma aplicação. E é para isso que servem as imagens.

Para criar uma imagem da instância configurada no início desse capítulo, basta clicar com o botão direito sobre ela no *dashboard* de instâncias e selecionar a opção **Image > Create image**. Na janela

de diálogo que abrir, apenas configure o nome da nova image e sua descrição, como mostra a figura a seguir:

The screenshot shows a 'Create Image' dialog box. It has four input fields: 'Instance ID' with value 'i-06137dbb1cdbf19b8', 'Image name' with value 'aws_project01' (the 'aws' part is redacted), 'Image description' with value 'AWS Project 01', and 'No reboot' with an unchecked checkbox. Below the fields is a button labeled 'Criando uma imagem'.

Instance ID	i	i-06137dbb1cdbf19b8
Image name	i	aws_project01
Image description	i	AWS Project 01
No reboot	i	<input type="checkbox"/>

Criando uma imagem

Para finalizar a criação da imagem, clique no botão Create Image.

Em alguns segundos a imagem será criada e exibida no menu Images > AMIs.

Essa imagem será a base do *template* a ser criado na próxima seção.

6.9.2 - Criando um template

Os *templates* podem ser utilizados para criação de outras instâncias iguais, a partir de uma imagem, sem o trabalho de se refazer toda configuração e instalação novamente.

Para se criar um *template* a partir da instância que já foi configurada nesse capítulo, simplesmente clique com o botão direito sobre ela, na lista de instâncias EC2, e escolha a opção Create Template From Instance.

Dentro da página de criação de *templates*, existem seções de configurações que devem ser preenchidas de forma obrigatória e outras que podem ser deixadas com valores padrões. São elas:

a) Identificação do template:

Essa primeira seção traz as configurações principais sobre a criação do *template*. A primeira opção questiona se um novo *template* deve ser criado ou se deve ser gerada uma nova versão a partir de um *template* já criado. Selecione a primeira opção para a criação de um novo *template*.

O segundo campo pede uma identificação do *template*, para que ele possa ser referenciado no futuro.

Por último, há um campo que deve ser preenchido com uma breve descrição sobre o novo *template*.

Veja como devem ficar as configurações para essa primeira seção:

What would you like to do? Create a new template from an instance
 Create a new template version from an instance

Source instance i-06137dbb1cdbf19b8

Launch template* aws_project01

Template version description AWS Project 01

Criação de templates

b) Conteúdo do template:

O primeiro campo dessa seção é a identificação da imagem do sistema operacional que deverá ser utilizada para construção de uma nova instância a partir desse *template*. Nessa opção, **escolha a imagem criada na seção anterior**, clicando na opção Search for AMI e selecionando a opção My AMIs. A imagem criada na seção anterior deverá ser exibida. Ela já contém a aplicação que foi desenvolvida, configurada para entrar em execução assim que o sistema operacional for iniciado.

O segundo campo representa o tipo da instância EC2 que será criada. Isso fará com que, quando uma nova instância for criada a partir desse *template*, o tipo da instância já seja escolhido a partir dessa configuração. Escolha a opção t2.micro, do mesmo tipo da instância que foi escolhida no início desse capítulo.

O terceiro campo é a escolha do par de chaves a ser utilizado para acessar as instâncias criadas a partir desse *template*. Aqui entra um ponto interessante, pois pode-se utilizar o mesmo par de chaves para acessar várias instâncias diferentes. Escolha o par de chaves que foi criado nesse capítulo, o mesmo utilizado para acessar a instância que foi criada até o momento.

O último campo, de nome *Launch template security groups*, é utilizado para criar instâncias com configurações de acesso baseadas em um grupo de segurança pré-definido. No caso da aplicação que foi construído e hospedado na instância, foi necessário configurar um redirecionamento de porta para que a aplicação pudesse ficar acessível pela Internet. Por isso, escolha o *security group* que foi configurado na seção 6.6 desse capítulo, que nesse caso foi o de nome **launch-wizard-1**. Isso fará com que o redirecionamento de porta que foi configurado nesse grupo de segurança seja aplicado a todas as instâncias que forem criadas a partir desse *template*.

Veja como deve ficar essa seção de configuração:

Launch template contents

AMI ID: ami-0ac019f4fcb7cb7e6

Instance type: t2.micro

Key pair name: key_pair_01

Network type: VPC

Launch template security groups: sg-07c73f2081ad98214 | launch...

Content do template:

c) Rede, armazenamento e outros:

As últimas seções trazem configurações sobre rede, armazenamento e outros comportamentos que podem ser deixados com os valores padrões que aparecem na tela de criação do *template*. Porém, a **configuração da interface de rede eth0 deve ser removida**, pois a escolha do *security group*, da seção anterior, já traz o que é necessário para tal.

Tendo finalizado todas as configurações, clique no botão *Create Template From Instance* para iniciar o processo de criação do *template*.

Pronto! O *template* foi criado, o que significa que novas instâncias podem ser criadas a partir dele.

A lista de *templates* aparece no menu *Instances > Launch Templates*.

6.9.3 - Criando uma instância a partir do template

Agora que uma imagem e um template já foram criados, é possível criar uma instância idêntica a que já foi criada nesse capítulo, mas agora, sem a necessidade de se refazer todo o processo de instalação e configuração. Para isso, estando na página que lista os *templates*, clique com o botão direito sobre o *template* que foi criado na seção anterior e selecione a opção *Launch instance from template*.

Na tela que abrir, apenas selecione a versão do *template*, que no momento é apenas um, e clique no botão *Launch instance from template* para iniciar o processo de criação da nova instância.

Volte na lista de instâncias para ver que uma nova instância foi criada. Perceba que ela é exatamente igual a que foi criada no início desse capítulo, incluindo a aplicação que já está instalada e configurada para ser executada assim que o sistema operacional for iniciado.

Quando a nova instância entrar em execução, copie o DNS público e acesse-a através do Postman, substituindo o endereço da primeira instância por esse endereço da nova.

Perceba que o serviço que foi criado na aplicação `aws_project01` responderá, da mesma forma que da primeira instância. Isso significa que essa nova instância é uma cópia fiel da primeira.

6.11 - Balanceador de carga

Imagine uma situação onde a aplicação `aws_project01` recebesse um alto tráfego de requisições, ao ponto de não conseguir tratar todos esses pedidos. Nesse caso, algumas requisições ficariam sem ser atendidas.

Uma solução para esse problema seria a criação de novas instâncias para poder atender a um maior número de requisições simultâneas. Embora essa seja uma boa solução, ainda seria necessário fazer com que tais requisições fossem divididas igualmente entre as instâncias da aplicação, ou seja, fazer com que a carga de trabalho entre elas seja balanceada, não deixando que uma fique mais sobrecarregada do que outra.

Felizmente, a AWS possui um serviço chamado `Load Balancers` que pode ser configurado para fazer exatamente esse trabalho: dividir a carga de trabalho entre várias instâncias que servem a um mesmo serviço.

A ideia é fazer com que as duas instâncias da aplicação `aws_project01`, que estão em execução, possam receber requisições de forma equilibrada, dessa forma a carga de trabalho entre elas tenderá a ficar dividida:

	Name	Instance ID	Instance Type	Availability Zone	Instance State
<input type="checkbox"/>	ec2_01	i-05ced0d39023bdfd1	t2.micro	us-east-1d	running
<input type="checkbox"/>	ec2_01	i-06137dbb1cdbf19b8	t2.micro	us-east-1b	running

Instâncias da aplicação `aws_project01`

Para começar com esse trabalho, acesse a opção `Load Balancers` no menu `Load Balancing`, dentro da mesma página do console de administração de EC2. Isso fará com que seu console seja exibido com a lista de todos os recursos criados desse tipo.

Para criar um novo *load balancer*, clique no botão `Create Load Balancer`, localizado no canto superior esquerdo da tela. Nessa tela, serão exibidas as opções de criação desse tipo de recurso, que são:

- **Classic Load balancer:** essa é a primeira versão desse recurso, podendo ser configurado para平衡ear as requisições a nível de protocolo.
- **Network Load Balancer:** essa é uma opção que abrange situações que envolvam controle de tráfego a nível de comunicação de rede.

- **Application Load Balancer:** esse é o tipo ideal de *load balancer* para controlar tráfego de requisições a nível de aplicação.

Selecione a opção **Application Load Balancer**, clicando no botão *Create* dessa opção. Nas configurações básicas do primeiro, dê um nome ao *load balancer* e deixe o restante das configurações como apresentadas na figura a seguir:

Step 1: Configure Load Balancer

Basic Configuration

To configure your load balancer, provide a name, select a scheme, specify one or more listeners, and select a network.

Name	<input type="text" value="project01-lb"/>
Scheme	<input checked="" type="radio"/> internet-facing <input type="radio"/> internal
IP address type	<input type="text" value="ipv4"/>

Nome do load balancer

Na seção **Listeners**, configure para que o *load balancer* possa aceitar conexões do protocolo HTTP e da porta 8080:

Listeners

A **listener** is a process that checks for connection requests, using the protocol and port that you configured.

Load Balancer Protocol	Load Balancer Port
<input type="text" value="HTTP"/>	<input type="text" value="8080"/>
<input type="button" value="Add listener"/>	

Porta de conexão do load balancer

Na última seção desse passo, configure as zonas de disponibilidade nas quais o *load balancer* deverá atuar:

Availability Zones

Specify the Availability Zones to enable for your load balancer. The load balancer routes traffic to the targets in these Availability Zones only.

balancer.

The screenshot shows the 'Availability Zones' section of the AWS Load Balancer configuration. It lists six zones: us-east-1a, us-east-1b, us-east-1c, us-east-1d, us-east-1e, and us-east-1f. For each zone, there is a checkbox followed by a dropdown menu showing the subnet assigned by AWS. The 'us-east-1b' and 'us-east-1d' checkboxes are checked, and their dropdown menus show 'subnet-36f2d41a' and 'subnet-e8752eb2' respectively, both labeled as 'Assigned by AWS'.

Availability Zone	Subnet Assigned by AWS
us-east-1a	subnet-3d69b459
us-east-1b	subnet-36f2d41a
us-east-1c	subnet-5951b512
us-east-1d	subnet-e8752eb2
us-east-1e	subnet-55d82a6a
us-east-1f	subnet-a27f29ae

Zonas de disponibilidade

As zonas que devem ser marcadas devem ser as mesmas onde as instâncias estão criadas. Isso pode ser visto no console de administração do EC2:

The screenshot shows a table of EC2 instances. The columns are: Select, Name, Instance ID, Instance Type, Availability Zone, and Instance State. There are two rows: one for 'ec2_01' (Instance ID i-05ced0d39023bdf01, t2.micro, us-east-1d, running) and one for 'ec2_01' (Instance ID i-06137dbb1cdbf19b8, t2.micro, us-east-1b, running). The 'Availability Zone' column is highlighted with a red box.

Select	Name	Instance ID	Instance Type	Availability Zone	Instance State
<input type="checkbox"/>	ec2_01	i-05ced0d39023bdf01	t2.micro	us-east-1d	running
<input type="checkbox"/>	ec2_01	i-06137dbb1cdbf19b8	t2.micro	us-east-1b	running

Zonas das instâncias

Perceba que as duas instâncias em execução estão localizadas em us-east-1d e us-east-1b, que foram as mesmas configuradas para o *load balancer*.

Agora, passe para o próximo passo clicando no botão Next dessa página. Nesse segundo passo não há nada para ser configurado para esse exemplo, por isso clique em Next novamente.

No passo 3, é necessário selecionar os *security groups* nos quais as instâncias da aplicação aws-project01 estão inseridas:

Step 3: Configure Security Groups

A security group is a set of firewall rules that control the traffic to your load balancer. Create a new security group or select an existing one.

- Assign a security group:
- Create a new security group
 - Select an existing security group

Security Group ID	Name
sg-3eb4bc4e	default
sg-07c73f2081ad98214	launch-wizard-1

Security groups

Em seguida clique em Next para passar ao passo 4, onde as configurações de roteamento do *load balancer* devem ser configuradas de acordo com a figura a seguir:

Step 4: Configure Routing

Your load balancer routes requests to the targets in this target group using the protocol

Target group

Target group	i	New target group
Name	i	project01-tg
Target type	<input checked="" type="radio"/> Instance <input type="radio"/> IP <input type="radio"/> Lambda function	
Protocol	i	HTTP
Port	i	8080

Health checks

Protocol	i	HTTP
Path	i	/actuator/health

Roteamento do load balancer

A primeira opção questiona se um novo grupo deve ser criado ou não. Caso seja selecionado a criação de um novo, seu nome deve ser configurado no campo Name.

Nesse passo, o mais importante é configurar o protocolo HTTP e a porta 8080, que são as configurações que estão feitas na aplicação aws_project01 para o recebimento de requisições externas.

A última configuração a ser feita é o caminho para o *endpoint* para verificar se a aplicação está rodando e funcionando. Felizmente o projeto aws_project01 já inclui uma biblioteca chamada **Spring Actuator**, que fornece esse *endpoint* pronto para ser utilizado, por isso configure no campo Path o endereço /actuator/health.

Passe para o próximo passo, clicando em Next. Nessa última página de configuração será necessário escolher quais instâncias o *load balancer* irá trabalhar:

Step 5: Register Targets

Register targets with your target group. If you register a target in an enabled Availability Zone, the load balancer starts routing requests to the target.

Registered targets

To deregister instances, select one or more registered instances and then click Remove.

<input type="checkbox"/>	Instance	Name	Port	State
No instances available.				

Instances

To register additional instances, select one or more running instances, specify a port, and then click Add. The default port is the port specified for the target instance.

<input type="checkbox"/>	Instance	Name	State	Security groups	Zone
<input type="checkbox"/>	i-05ced0d39023bdfd1	ec2_01	running	launch-wizard-1	us-east-1d
<input type="checkbox"/>	i-06137dbb1cdbf19b8	ec2_01	running	launch-wizard-1	us-east-1b

Selecionando as instâncias

Repare que as duas instâncias que foram criadas para a aplicação aws_project01 aparecem na lista. Para instruir o *load balancer* que está sendo criado a trabalhar com elas, selecione-as e clique no botão *Add to registered*. Dessa forma elas passarão para a lista Registered targets, como mostra a figura a seguir:

Registered targets

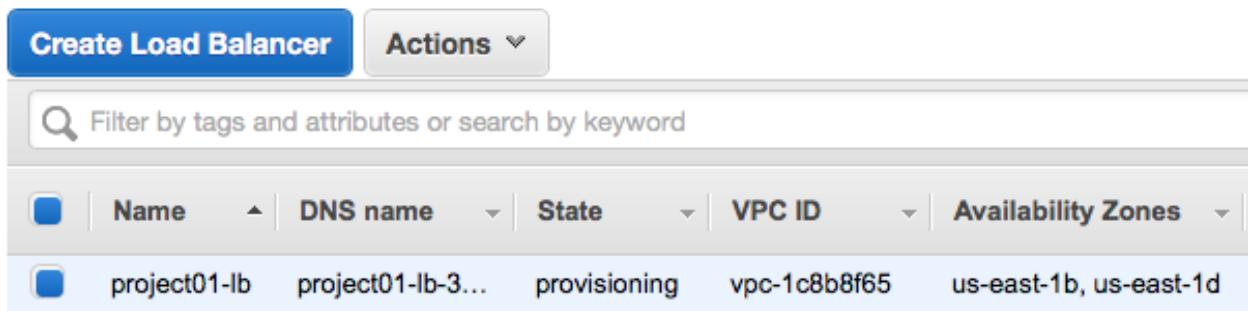
To deregister instances, select one or more registered instances and then click Remove.

<input type="checkbox"/>	Instance	Name	Port	State	Security groups	Zone
<input type="checkbox"/>	i-05ced0d39023bdfd1	ec2_01	8080	running	launch-wizard-1	us-east-1d
<input type="checkbox"/>	i-06137dbb1cdbf19b8	ec2_01	8080	running	launch-wizard-1	us-east-1b

Alvos do load balancer

Agora essas duas instâncias são “alvos” do *load balancer*, ou seja, toda requisição que atingí-lo será redirecionada para uma das duas instâncias, tentando manter uma carga balanceada entre elas.

Para concluir com o processo de configuração do *load balancer*, clique no botão *Next* para ir para a página com o resumo das configurações realizadas. E para finalizar, clique em *Create*. Em alguns instantes o *load balancer* será criado e aparecerá na lista de seu console de administração:



The screenshot shows the AWS Load Balancer console. At the top, there's a blue header bar with 'Create Load Balancer' and 'Actions' dropdown. Below it is a search bar with the placeholder 'Filter by tags and attributes or search by keyword'. The main area has a table with columns: Name, DNS name, State, VPC ID, and Availability Zones. A single row is visible: 'project01-lb' with 'project01-lb-3...' under DNS name, 'provisioning' under State, 'vpc-1c8b8f65' under VPC ID, and 'us-east-1b, us-east-1d' under Availability Zones.

	Name	DNS name	State	VPC ID	Availability Zones
<input checked="" type="checkbox"/>	project01-lb	project01-lb-3...	provisioning	vpc-1c8b8f65	us-east-1b, us-east-1d

Load balancer criado

Quando o *load balancer* é selecionado, sua seção de administração e monitoramento é exibida na parte inferior da página. Nesse momento, o dado mais importante a ser observado é o **DNS name**, que é o endereço no qual a aplicação `aws_project01` deverá ser acessada, passando pelo *load balancer* para que ele possa redirecionar as requisições de forma igual para cada instância em execução.

De posse desse DNS do *load balancer* faça o seguinte teste:

- Acesse cada instância através de um cliente SSH e visualize os logs de cada uma delas;
- Configure o Postman para acessar as instâncias da aplicação `aws_project01` com base nesse endereço: `http://<dns do load balancer>:8080/api/test/dog/{name}`;
- Faça sucessivas requisições com o Postman, colocando valores diferentes para o parâmetro `name` do serviço `test` da aplicação.

O resultado esperado desse teste é que cada requisição feita seja tratada por uma instância, de forma alternada, provando que o *load balancer* está fazendo seu trabalho de **balancear a carga de requisições entre as duas instâncias** que foram configuradas.

As instâncias ainda podem ser acessadas diretamente, através do endereço DNS que pode ser visualizado em cada uma delas no console de administração do EC2, porém em uma situação real, elas somente poderiam ser acessadas através do *load balancer* que foi criado, para que ele possa dividir as requisições entre elas.

Se esse fosse um serviço a ser exposto na Internet, seria o DNS do *load balancer* que deveria ser divulgado para que outros pudessem acessá-lo externamente, ou seja, ele é o ponto de entrada na infraestrutura da AWS para a aplicação `aws_project`.

Em uma situação real, caso fossem necessárias mais instâncias da aplicação `aws_project01`, elas poderiam ser criadas e adicionadas na configuração do *load balancer* para que ele passasse a enxergá-las e redirecionar as requisições para todas as instâncias de forma equilibrada.

A utilização de *load balancer* é uma estratégia de extrema importância e necessidade, pois garante uma estabilidade e disponibilidade maior a uma aplicação.



As instâncias EC2 e o *load balancer* que foram criados nesse capítulo são recursos custam relativamente caro em comparação a outros tipos de recursos da AWS. Caso esteja utilizando uma conta paga, lembre-se de excluir os recursos caso não vá utilizar por um longo período.

6.12 - Conclusão

Esse capítulo apresentou conceitos chaves para utilização de instâncias EC2 para hospedarem aplicações no ambiente AWS, além disso, também foram apresentados os seguintes conceitos:

- Como criar, acessar e monitorar instâncias EC2;
- Como instalar uma aplicação Java desenvolvida com Spring Boot em uma instância EC2;
- Como colocar a aplicação para ser executada na instância EC2;
- Como criar outras instâncias a partir de um *template* e uma imagem configurados;
- Como criar um *load balancer* para equilibrar as requisições entre várias instâncias de uma mesma aplicação.

O próximo capítulo introduzirá alguns conceitos iniciais sobre a criação de recursos AWS utilizando **CloudFormation**, uma forma muito utilizada para que tal tarefa seja feita de forma controlada, prática e automatizada.

7 - Criando recursos AWS com CloudFormation

No capítulo anterior foram criados os seguintes recursos principais para a execução da aplicação `aws_project01` na AWS:

- Duas instâncias EC2, a partir de um *template* que contém uma imagem já configurada com a aplicação;
- Um *Application Load Balancer* para balancear o tráfego de requisições de forma equilibrada entre as duas instâncias.

Embora esse trabalho tenha sido fácil, principalmente com a utilização de um *template* para a criação da instância EC2, ele teve que ser feito de forma manual, ou seja, através do console de administração da AWS.

O processo de criação de recursos na AWS feito de forma manual possui algumas desvantagens:

- **Erros** podem ser cometidos, principalmente quando vários parâmetros devem ser digitados pelo operador;
- O **tempo** para a criação de recursos depende da velocidade do operador;
- Por mais que haja um **processo documentado** do que deva ser feito, não é garantido que tudo seja feito corretamente.

Felizmente a AWS possui um serviço chamado **AWS CloudFormation**, onde é possível criar um documento que descreve tudo o que deve ser feito para a criação de recursos. Esse documento então é interpretado e executado pela AWS de forma automática, através de parâmetros ou não.

Através desse documento, que são chamados de *AWS CloudFormation templates*, é possível:

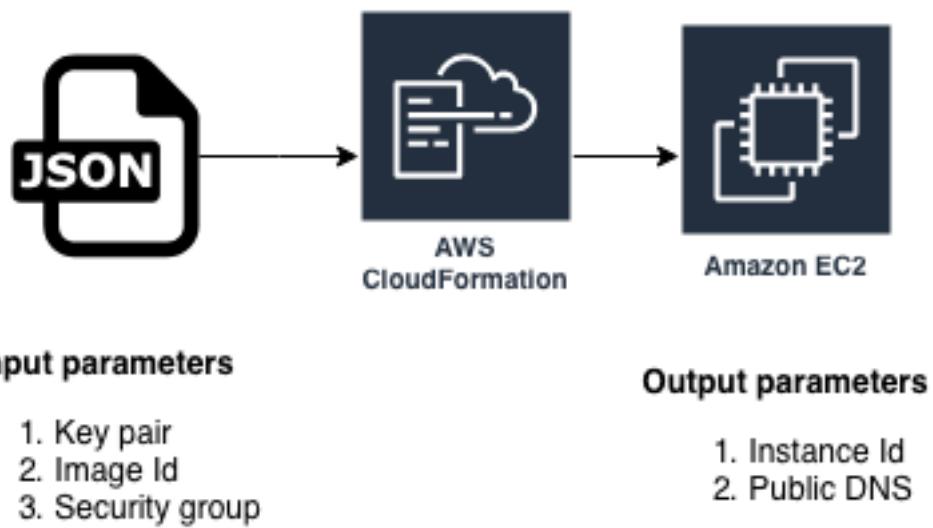
- Especificar qualquer parâmetro de entrada para a criação de um recurso;
- Definir regras de dependências de outros recursos;
- Definir parâmetros de saída após a criação dos recursos;
- Criar versões diferentes do *template*;
- Permitir a revisão do *template* utilizando ferramentas de repositórios de código;
- Automatizar o processo de criação automática de recursos AWS.

Uma vez que o *template* é criado, ele pode ser submetido ao console do AWS CloudFormation para ser executado e então ter os recursos criados. Dessa forma, todos os recursos serão criados automaticamente, sem que haja a necessidade da intervenção do operador.

A técnica de se utilizar o AWS CloudFormation com os *templates* agiliza o processo de criação de novos ambientes de forma automatizada, além de permitir a **réplica de ambiente inteiros**, apenas com a mudança de alguns parâmetros.

O mais importante de tudo é que **esse serviço é gratuito**, pois a cobrança é feita somente sobre os recursos que são criados com o AWS CloudFormation.

O intuito desse capítulo é criar um *template* para o AWS CloudFormation para a criação de uma instância para a execução da aplicação `aws_project01`.



Proposta do capítulo

Nesse *template*, serão configurados os parâmetros para a criação da instância EC2, utilizando sua estrutura padronizada. Também serão criados parâmetros de saída para serem visualizados após a criação dessa instância.

Durante o processo de criação do *template* ao longo desse capítulo, serão detalhadas várias características do serviço AWS CloudFormation e como seu processo é estruturado.

7.1 - O que são templates

Os *templates* do AWS CloudFormation são arquivos texto em formato JSON ou YAML, com seções específicas para descrever o que é necessário de ser criado para a execução de uma aplicação ou um serviço, além de definir dependências durante o processo de criação de tais recursos.



Esse livro se concentra na criação de *templates* de CloudFormation em formato JSON.

Ele também pode definir parâmetros de entrada, vindos ou não de outros processos de CloudFormation. Esse parâmetros podem ser utilizados para definir características do recurso que está sendo criado.

Também é possível definir parâmetros de saída, gerados durante o processo de criação do recurso. Esses parâmetros podem servir de entrada para outros processos de CloudFormation ou simplesmente serem utilizados para observação ao final do processo.

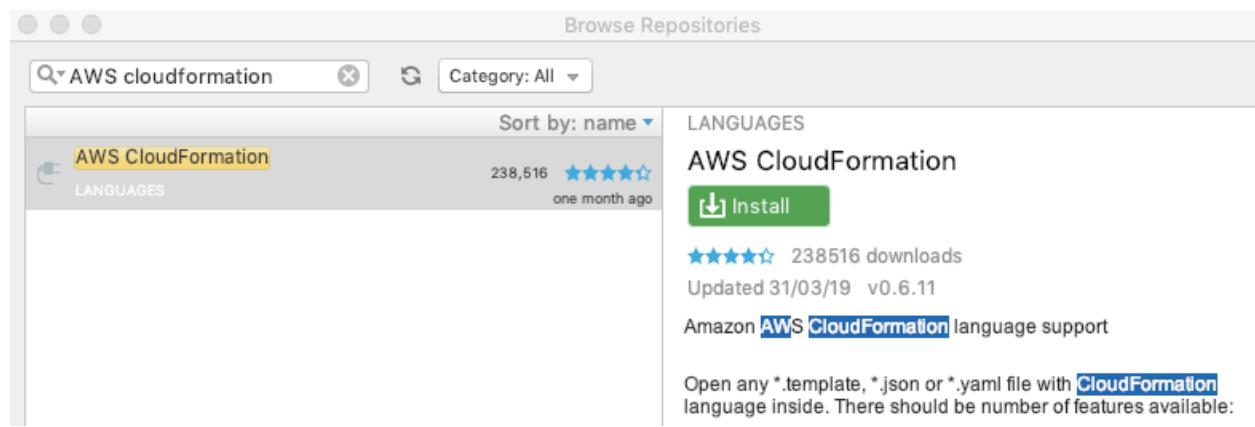
Os *templates* podem ser organizados para criação de um ou mais recursos, dependendo da organização dos processos de CloudFormation.

7.2 - Estrutura básica de um template

A estrutura básica de um template para ser utilizado no AWS CloudFormation possui as seguintes partes principais:

- **Versão do template:** define qual é a versão da AWS para o formato do *template* a ser utilizada. Caso não seja fornecido, a última versão é utilizada;
- **Descrição do template:** texto para detalhar sobre o que se trata o *template*, como por exemplo quais recursos ou serviço será criado;
- **Parâmetros:** aqui podem ser definidos parâmetros de entrada para os recursos que serão criados, como por exemplo tipos de instância EC2;
- **Recursos:** nessa seção é que os recursos que serão criados devem ser especificados;
- **Saída:** aqui podem ser definidos parâmetros de saída com algumas informações dos recursos que foram criados, como por exemplo o DNS público de uma instância EC2.

Um *template* do AWS CloudFormation pode ser criado em qualquer editor de texto, porém é interessante utilizar o IntelliJ IDEA com um plugin desenvolvido para ele, para facilitar a sua construção. Veja na figura a seguir como proceder com sua instalação, acessando o menu *Settings/Preference* -> *Plugins*:



Plugin para AWS CloudFormation

A seguir um exemplo básico de um *template* de CloudFormation, retirado do site da documentação sobre esse serviço da AWS:

```
{  
  "AWSTemplateFormatVersion" : "2010-09-09",  
  "Description" : "A simple EC2 instance",  
  "Resources" : {  
    "MyEC2Instance" : {  
      "Type" : "AWS::EC2::Instance",  
      "Properties" : {  
        "ImageId" : "ami-0ff8a91507f77f867",  
        "InstanceType" : "t1.micro"  
      }  
    }  
  }  
}
```

Nesse exemplo, que não possui as seções para definir parâmetros de entrada e de saída, uma simples instância EC2 é criada, com o tipo t1.micro.

As sessões seguintes desse capítulo mostrarão outros conceitos envolvidos na criação de recursos com o AWS CloudFormation.

7.3 - O que são stacks

Stacks são uma parte fundamental do processo do AWS CloudFormation. Elas são um agrupamento de todos os recursos criados a partir de um *template* e são utilizadas para o gerenciamento do ciclo deles.

Durante o processo de criação de recursos a partir de um *template*, uma *stack* é criada para:

- Obter os parâmetros requeridos pelo *template*;
- Gerenciar os estados do processo de criação dos recursos;
- Efetivamente criar os recursos;
- Gerenciar as dependências entre os recursos criados;
- Cuidar da reversão do processo, caso algum passo não possa ser realizado;
- Exibir os parâmetros de saída que podem ser gerados pelo *template*.

A seguir um exemplo dos eventos de execução de uma stack qualquer:

Events				
Timestamp	Logical ID	Status	Status reason	
2019-06-09 11:55:32 UTC-0300	pcs-ec2-01	CREATE_COMPLETE	-	
2019-06-09 11:55:29 UTC-0300	Ec2Instance	CREATE_COMPLETE	-	
2019-06-09 11:54:36 UTC-0300	Ec2Instance	CREATE_IN_PROGRESS	Resource creation Initiated	
2019-06-09 11:54:34 UTC-0300	Ec2Instance	CREATE_IN_PROGRESS	-	
2019-06-09 11:54:30 UTC-0300	pcs-ec2-01	CREATE_IN_PROGRESS	User Initiated	

Eventos de uma stack

Nesse exemplo, é possível observar todos os eventos gerados pela execução da *stack* de nome *pcs-ec2-01*, para a criação de uma instância EC2.

Os recursos criados pela *stack* podem ser observados na aba Resources, como mostra a figura a seguir:

Resources (1)

Logical ID	Physical ID	Type	Status	Status
Ec2Instance	i-01d17e73d7d6a1ad9	AWS::EC2::Instance	CREATE_COMPLETE	-

Recursos da stack

Essa instância que foi criada pela *stack*, após sua conclusão, já aparece no *dashboard* de instâncias em execução, pronta para o uso.

Um ponto importante de se ressaltar é que os recursos criados pela *stack* estão ligados ao seu ciclo de vida também, ou seja, quando uma *stack* é apagada, todos os recursos que foram criados por ela são apagados também.

7.4 - Construindo um template para criar uma instância EC2

Agora que alguns conceitos importantes já foram apresentados, é hora de construir um *template* capaz de criar uma instância EC2. Ele deverá ter as seguintes características:

- Parâmetros de entrada:
 - O par de chaves para dar acesso via SSH a instância;
 - A identificação da imagem a ser utilizada para a construção da instância;
 - A identificação do grupo de segurança para definir as regras de acesso a instância.
- Parâmetros de saída:
 - Identificação única da instância;
 - DNS público para acesso a instância.
- Recursos e suas propriedades:
 - Tipo do recurso, que nesse caso será uma EC2;
 - Tipo da instância;
 - Região e zona onde a instância será criada;

- Nome da instância;
- Grupo de segurança que pertencerá;
- Par de chaves para acesso via SSH;
- Identificação da imagem base para a construção da instância EC2.

Repare que as 3 últimas propriedades serão definidas por parâmetros de entrada, assim como outras propriedades também poderiam, como o nome da instância e seu tipo. Essa técnica permite que o *template* do CloudFormation fique mais genérico, permitindo ser parametrizado e reaproveitado em outras situações.

7.4.1 - Construindo o *template*

Para começar a construir o *template*, crie um novo arquivo (salve no seu computador) chamado ec2.json, começando com suas seções principais:

```
{
  "AWSTemplateFormatVersion": "2010-09-09",
  "Description": "Create an EC2 instance running from an image.",

  "Parameters": {
  },

  "Resources": {
  },

  "Outputs": {
  }
}
```

O primeiro campo, de nome `AWSTemplateFormatVersion`, se refere à versão do *template* a ser construído. Caso ele seja omitido, a última versão será utilizada.

O segundo campo, de nome `Description`, é apenas uma descrição que aparecerá nas informações da *stack* que for criada a partir desse *template*.

As demais seções serão detalhadas a seguir.

7.4.2 - Definindo os parâmetros de entrada

A seção `Parameters` define os parâmetros de entrada para a criação da *stack*. Esses dados de entrada podem ser utilizados para a definição de valores e propriedades dos recursos que serão criados.

Cada parâmetro deve seguir o seguinte formato para a sua definição:

```

"InputParameterName": {
    "Description" : "Input parameter description",
    "Type": "Input parameter type",
    "Default": "default value"
}

```

- **InputParameterName**: nome que pode ser escolhido pelo desenvolvedor, para melhor definir o significado do parâmetro de entrada;
- **Description**: descrição do parâmetro de entrada que aparecerá na aba **Parameters** da *stack*;
- **Type**: tipo do parâmetro, que deve ser escolhido de acordo com o tipo do valor que será passado;
- **Default**: valor padrão caso ele não seja passada no momento da execução do *template* pelo CloudFormation.

A seção de parâmetros do *template* possui algumas regras:

- Pode haver no máximo 60 parâmetros de entrada;
- O nome deve ser único dentro do *template* e composto por caracteres alfanuméricos;
- Todos os parâmetros devem possuir um tipo, que podem ser basicamente:
 - String;
 - Número;
 - Lista de números ou strings;
 - Tipos específicos da AWS.

Todos os tipos de valores que podem ser utilizados nos parâmetros estão [aqui³](#).

Todas as regras para definição dos parâmetros, bem como todos os atributos que eles podem ter estão definidos nesse endereço [aqui⁴](#).

Como citado no início dessa seção, o exemplo a ser construído para a criação da instância EC2, deverá receber os seguintes parâmetros:

- **Par de chaves** para dar acesso via SSH à instância;
- **Identificação da imagem** a ser utilizada para a construção da instância;
- **Identificação do grupo de segurança** para definir as regras de acesso à instância.

Seguindo a especificação do AWS CloudFormation, a seção **Parameters** do *template* deveria ficar da seguinte forma:

³<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html#parameters-section-structure-properties-type>

⁴<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html>

```

"Parameters": {
  "KeyPair": {
    "Description" : "The EC2 Key Pair to access the instance using SSH",
    "Type": "String"
  },
  "ImageId": {
    "Description" : "The image ID to create the instance",
    "Type": "String"
  },
  "SecurityGroup": {
    "Description" : "The security group name",
    "Type": "String"
  }
},

```

Aqui são definidos os 3 parâmetros citados anteriormente, com os seguintes nomes:

- KeyPair
- ImageId
- SecurityGroup

Todos os parâmetros são do tipo `String` e possuem nomes declarativos, bem como descrições apropriadas.

Esses parâmetros irão definir valores para propriedades específicas do recurso EC2 que será declarado na próxima seção.

O arquivo final, que será construído ao longo das próximas seções, ficará semelhante ao criado [aqui](#)⁵

7.4.3 - Definindo o recurso e suas propriedades

A seção `Resources` do *template* do CloudFormation pode conter uma lista de recursos a serem criados, todos referenciados por um nome. Sua estrutura básica é da seguinte forma:

⁵https://github.com/siecola/aws_book_cfn/blob/master/ec2.json

```

"Resources": {
  "ResourceName": {
    "Type": "AWS::EC2::Instance",
    "Properties": {
      "InstanceType": "t2.micro",
    }
  }
},

```

Como pode ser observado, cada recurso deve possuir um nome para ser unicamente referenciado no *template*, aqui exemplificado apenas como `ResourceName`.

Dentro da especificação de cada recurso devem ser especificados seu tipo, pelo campo **Type** e sua lista de propriedades, no campo **Properties**. Esse último pode variar dependendo do recurso e algumas podem assumir um valor padrão, caso não sejam especificadas.

A **lista completa dos tipos de recursos** que podem ser criados pode ser encontrada neste endereço [aqui⁶](#). Dentro de cada recurso encontram-se os subtipos de cada um deles, como por exemplo para a criação de uma [instância EC2⁷](#).

E por fim nessa página encontram-se todos as propriedades que podem ser configuradas com um detalhamento completo de cada uma delas.

Para o exemplo desse capítulo, as seguintes propriedades devem ser configuradas:

- Tipo do recurso, que nesse caso será uma EC2;
- Tipo da instância;
- Região e zona onde a instância será criada;
- Nome da instância;
- Grupo de segurança que pertencerá;
- Par de chaves para acesso via SSH;
- Identificação da imagem base para a construção da instância EC2.

Isso significa que a seção `Resources` do *template* deverá ficar como no trecho a seguir:

⁶<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html>

⁷<https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-ec2-instance.html>

```

"Resources": {
  "Ec2Instance": {
    "Type": "AWS::EC2::Instance",
    "Properties": {
      "InstanceType": "t2.micro",
      "AvailabilityZone": "us-east-1a",
      "SecurityGroups": [
        {
          "Ref": "SecurityGroup"
        }
      ],
      "KeyName": {
        "Ref": "KeyPair"
      },
      "ImageId": {
        "Ref": "ImageId"
      },
      "Tags" : [
        {"Key" : "Name", "Value" : "pcs_ec2_01"}
      ]
    }
  }
},

```

No trecho anterior, que definiu o recurso a ser criado, alguns valores estão fixos:

- **Type**: obviamente ele define o tipo do recurso a ser criado, que nesse caso é uma instância EC2;
- **InstanceType**: essa propriedade define o tipo da instância a ser criada. Nesse caso foi escolhido o tipo `t2.micro`;
- **AvailabilityZone**: a região e a zona escolhida para a criação da instância;
- **Nome da instância**: configura o nome da instância.

As demais propriedades estão sendo preenchidas com valores retirados dos parâmetros de entrada, que são:

- **SecurityGroups**: identificação do grupo de segurança para definir as regras de acesso à instância;
- **KeyName**: par de chaves para dar acesso via SSH à instância;
- **ImageId**: identificação da imagem a ser utilizada para a construção da instância.

Esse são valores de recursos ou configurações que já foram criados e deverão ser fornecidos no momento da execução da *stack*, que será criada por esse *template*. No momento da submissão da *stack*, na seção 7.5, será detalhado como obter esses valores.

7.4.4 - Definindo os parâmetros de saída

A seção `Outputs` de um *template* define valores contextuais aos recursos que foram criados pela *stack* e podem ser utilizados para:

- Parametrizar a execução de outra *stack*;
- Prover valores a serem utilizados por outra aplicação;
- Permitir a consulta de tais valores na aba `Outputs` no console de administração da *stack*.

O desenvolvedor do *template* pode definir qualquer parâmetro de saída que desejar, desde que respeite o limite máximo de 60 parâmetros.

Bons exemplos de parâmetros de saída que podem ser utilizados durante a criação de uma EC2 são:

- O endereço do DNS público criado pela AWS que possibilita o acesso a ela;
- A identificação única da instância criada.

Esses dois parâmetros de saída são de extrema importância de serem exportados, para que o administrador possa pelo menos localizar a instância que foi criada, caso possua várias.

A sintaxe com os parâmetros mínimos para definir um parâmetro de saída é da seguinte forma:

```
"ParameterName": {  
    "Description": "The parameter description",  
    "Value": {  
        "Ref": "The resource reference"  
    }  
}
```

A seguir, a definição de cada um atributos:

- **ParameterName**: nome declarativo do parâmetro de saída;
- **Description**: descrição sobre o que se trata;
- **Value**: valor do parâmetro de saída, contendo uma referência a um dos recursos criados pela *stack* ou um de seus atributos.

Sendo assim, para definir tais parâmetros, a seção `Outputs` do *template* deverá ficar como no trecho a seguir:

```
"Outputs": {
    "InstanceId": {
        "Description": "The InstanceId",
        "Value": {
            "Ref": "Ec2Instance"
        }
    },
    "PublicDnsName" : {
        "Description" : "Public DNS address",
        "Value" : { "Fn::GetAtt" : [ "Ec2Instance", "PublicDnsName" ] }
    }
}
```

E isso conclui a construção do *template* para a criação de uma instância EC2!

7.5 - Criando e gerenciando stacks

Tendo o *template* construído, agora é necessário executá-lo no [console](#)⁸ do AWS CloudFormation, para que o recurso desejado seja criado. Para isso, acesse a opção Create Stack desse console:

⁸<https://console.aws.amazon.com/cloudformation/home>

The screenshot shows the AWS CloudFormation Stacks page. At the top, it says "CloudFormation > Stacks". Below that, it displays "Stacks (0)". There are two main buttons: "Create stack" and a search icon. A search bar is present below the search icon. A dropdown menu for filtering by status is set to "Active". A toggle switch for "View nested" is turned on. To the right of the status filter, there is a page navigation section with a left arrow, the number "1", and a right arrow. The main content area below the filters displays the message "No stacks" and "No stacks to display".

No stacks

No stacks to display

Create stack

Iniciando o processo de criação de uma stack



É importante observar que o usuário que está com a sessão aberta do console do AWS CloudFormation deve possuir permissões para a criação dos recursos. O usuário administrador da conta já possui tais privilégios.

Na primeira tela do processo, clique na opção Choose File para subir o arquivo do *template* que foi criado:

Template source

Selecting a template generates an Amazon S3 URL where it will be stored.

 Amazon S3 URL Upload a template file**Upload a template file** **ec2.json**

JSON or YAML formatted file

Escolhendo o arquivo do template

Clique no botão Next para passar para a próxima etapa. Nesse momento, o arquivo do *template* é interpretado pelo CloudFormation, o que faz com que os parâmetros de entrada, configurados no *template*, já sejam exibidos para o seu preenchimento:

Stack name

Stack name

Enter a stack name

Stack name can include letters (A-Z and a-z), numbers (0-9), and dashes (-).

Parameters

Parameters are defined in your template and allow you to input custom values when you create or update a stack.

ImageId
The image ID to create the instance

KeyPair
The EC2 Key Pair to access the instance using SSH

SecurityGroup
The security group name

Definindo parâmetros da stack

Nessa tela, defina o nome que deseja para a *stack* que será criada. Além disso, os parâmetros **ImageId**, **KeyPair** e **SecurityGroup** também devem ser configurados, como será explicado a seguir.

Para o parâmetro **ImageId**, utilize o AMI ID da imagem criada na seção [6.9.1 do capítulo anterior](#). Esse valor pode ser localizado no console AWS para instâncias EC2, na seção **Images -> AMIs**, como mostrado na figura a seguir:

The screenshot shows the AWS EC2 Dashboard. On the left sidebar, under the 'IMAGES' section, the 'AMIs' link is highlighted with a red box. The main content area displays a table of AMIs. The columns are labeled 'Name' and 'AMI ID'. A search bar at the top right says 'Owned by me'. A red box highlights the 'AMI ID' column header and the specific value 'ami-06f5469e2c6e1b2d1' for the row where the AMI name is 'aws_project01'.

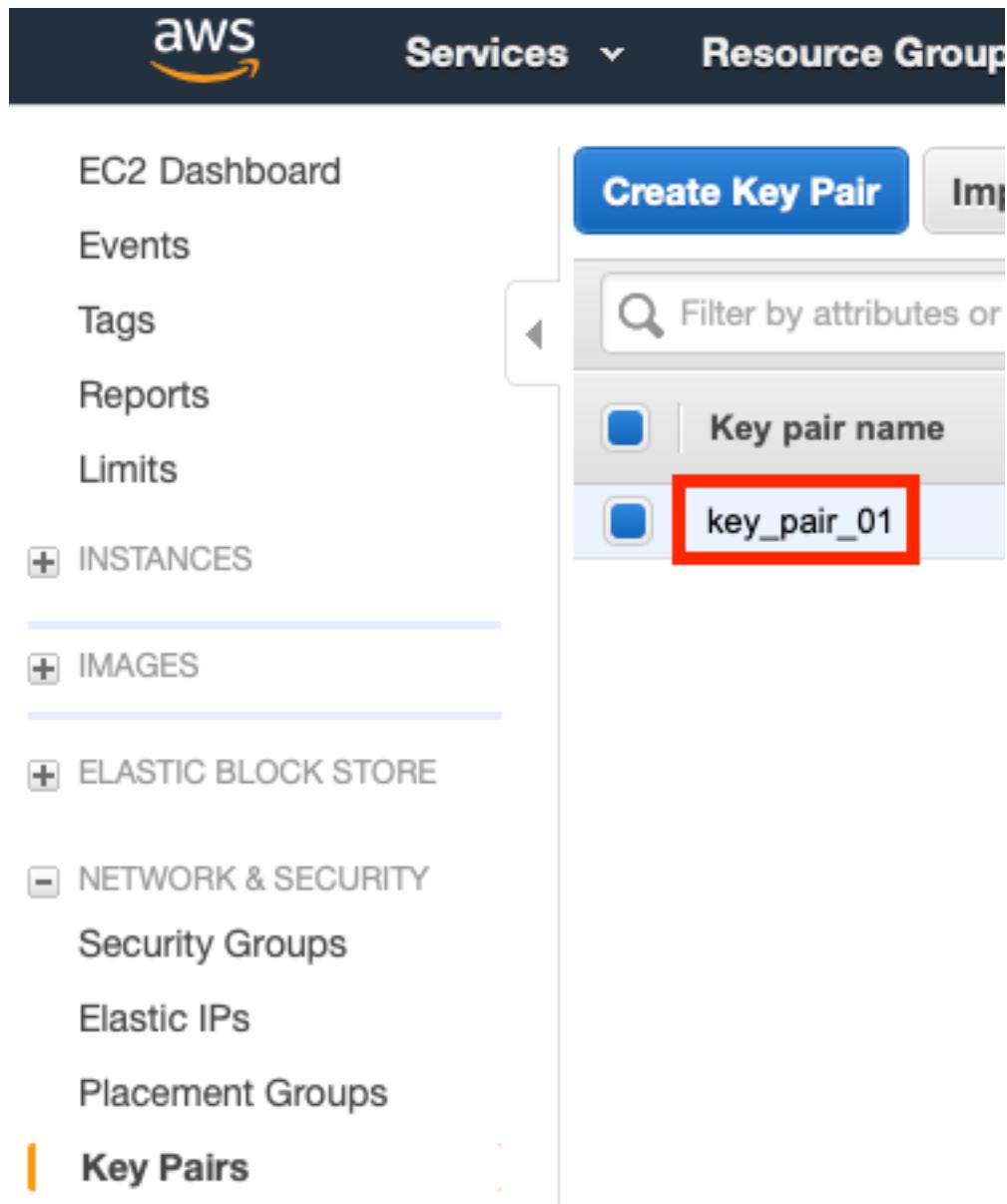
Coletando a identificação da imagem

Estando nessa tela, copie o valor do campo **AMI ID** e coloque no parâmetro **ImageId** na tela de criação da **stack**.

Dessa forma a instância será criada a partir dessa imagem, significando que:

- O sistema operacional será o escolhido durante a criação dessa imagem, ou seja, será utilizado o Ubuntu Server 18.04 LTS;
- A aplicação `aws_project01` já estará instalada;
- Os scripts de inicialização já estarão criados e configurados para executar a aplicação `aws_project01`, assim que o sistema operacional acabar de subir.

Ainda dentro do console AWS para instâncias EC2, localize a seção **Network & Security -> Key Pairs** para buscar o parâmetro **KeyPair**:



Coletando o nome para o par de chaves de acesso

Estando nessa tela, copie o valor do campo `Key pair name` e coloque no parâmetro `KeyPair` na tela de criação da `stack`.

Utilizando o mesmo par de chaves criado no capítulo anterior, será possível acessar a instância que for criada nessa `stack`.

Por fim, para preencher o nome do grupo de segurança, acesse a opção `Security Groups`, no mesma seção selecionada anteriormente:

The screenshot shows the AWS CloudFormation 'Create New Stack' wizard, Step 3: Set stack security group. The left sidebar shows navigation links like EC2 Dashboard, Events, Tags, Reports, Limits, INSTANCES, IMAGES, ELASTIC BLOCK STORE, and NETWORK & SECURITY. Under NETWORK & SECURITY, 'Security Groups' is selected. The main area has a 'Create Security Group' button and an 'Actions' dropdown. A search bar says 'Filter by tags and attributes or search by keyword'. A table lists security groups: 'sg-07c73f2081ad98214' (Group Name: 'launch-wizard-1') and 'sg-3eb4bc4e' (Group Name: 'default'). A red box highlights the 'Group Name' column for the first row. Below the table is a note: 'Select a security group above'.

Coletando o nome do grupo de segurança

Estando nessa tela, copie o valor do campo `Group Name` e coloque no parâmetro `SecurityGroup` na tela de criação da `stack`.

Isso fará com que a instância a ser criada tenha as mesmas regras de segurança aplicadas na instância que foi criada no capítulo anterior, ou seja, **acesso às portas TCP 8080 e 22**.

Tendo todos os parâmetros necessários para a criação da `stack` preenchidos, clique no botão `Next` para passar para a próxima tela.

Nessa segunda tela da criação da `stack` não é necessário fazer nada para esse exemplo, mas é possível:

- Criar `tags` para os recursos que serão criados;
- Definir outro papel de uma permissão do IAM para a criação dos recursos;
- Criar configurações de notificações sobre o andamento do processo da `stack`.

Deixe toda as configurações dessa página como estão e clique no botão `Next`. Nessa última página, revise as configurações realizadas:

Template

Template URL
<https://s3-external-1.amazonaws.com/cf-templates-10wop8jiyyofv-us-east-1/2019160Mqo-ec2.json>

Stack description
Create an EC2 instance running from an image.

[Estimate cost](#)

Step 2: Specify stack details

Parameters (3)

Search parameters

Key	Value
ImageId	ami-06f5469e2c6e1b2d1
KeyPair	key_pair_01
SecurityGroup	launch-wizard-1

Revisão da stack

Perceba que o *template* foi automaticamente armazenado em um local no S3, para que possa ser processado.

Para finalizar, clique em Create stack e acompanhe o processo da *stack*:

The screenshot shows the AWS CloudFormation console with the stack named 'pcs-ec2-01'. The 'Events' tab is selected. A search bar at the top says 'Search events'. Below is a table with the following data:

Timestamp	Logical ID	Status	Status reason
2019-06-09 15:14:32 UTC-0300	pcs-ec2-01	CREATE_COMPLETE	-
2019-06-09 15:14:29 UTC-0300	Ec2Instance	CREATE_COMPLETE	-
2019-06-09 15:13:57 UTC-0300	Ec2Instance	CREATE_IN_PROGRESS	Resource creation Initiated
2019-06-09 15:13:55 UTC-0300	Ec2Instance	CREATE_IN_PROGRESS	-
2019-06-09 15:13:50 UTC-0300	pcs-ec2-01	CREATE_IN_PROGRESS	User Initiated

Eventos da stack

A aba Events mostra todos os passos executados pelo CloudFormation durante o processo de criação da *stack*.

Quando o status chegar em CREATE_COMPLETE, significa que os recursos definidos no *template* foram criados corretamente e já estão disponíveis para serem utilizados. Caso algo dê errado, o CloudFormation cuida de reverter todo o processo e também de apagar o que foi criado.

Repare na aba Outputs que os parâmetros de saída definidos no *template* são exibidos com base nas informações que foram retiradas do recurso que foi criado:

pcs-ec2-01

Stack info | Events | Resources | **Outputs** | Parameters

Outputs (2)

Search outputs

Key	Value	Description
InstanceId	i-0bcdcc7c7b76b1bc61	The InstanceId
PublicDnsName	ec2-34-239-146-44.compute-1.amazonaws.com	Public DNS address

Parâmetros de saída da stack

A instância criada, com a identificação única exibida nessa aba, pode ser visualizada no *dashboard* de instâncias EC2 do console da AWS, como mostra a figura a seguir:

Launch Instance | Connect | Actions ▾

Filter by tags and attributes or search by keyword

	Name	Instance ID	Instance Type	Availability Zone	Instance State
<input type="checkbox"/>	pcs_ec2_01	i-0bcdcc7c7b76b1bc61	t2.micro	us-east-1a	running
<input type="checkbox"/>	ec2_01	i-05ced0d39023bdf1	t2.micro	us-east-1d	stopped
<input type="checkbox"/>	ec2_01	i-06137dbb1cdbf19b8	t2.micro	us-east-1b	stopped

Instância criada pela stack

Perceba que o tipo da instância e a zona de disponibilidade correspondem ao que foi passado na criação do recurso, dentro do *template*:

- Tipo da instância: t2.micro
- Zona de disponibilidade: us-east-1a

Como tudo foi preparado para que essa instância seja idêntica a que foi criada no capítulo anterior, basta utilizar o DNS público, exibido na aba *Outputs* da *stack*, para acessar o serviço de teste que foi criado na aplicação *aws_project01*. Isso prova que a instância criada é realmente uma réplica, a partir da imagem escolhida e passada como parâmetro para a execução da *stack*.

7.6 - Atualizando uma stack

Imagine que a instância EC2 que foi criada nessa *stack*, por alguma razão, não atende às necessidades de execução da carga de requisições e tenha que ser trocada por uma maior. O que fazer?

Felizmente o AWS CloudFormation possui um mecanismo que possibilita alterações na *stack* e consequentemente em seus recursos, sem que ela tenha que ser apagada e criada novamente.

Para demonstrar essa característica, altere o tipo da instância no *template* para *t2.medium*:

```
"Resources": {  
    "Ec2Instance": {  
        "Type": "AWS::EC2::Instance",  
        "Properties": {  
            "InstanceType": "t2.medium",
```

Agora acesse o console do AWS CloudFormation, selecione a *stack* criada anteriormente e selecione a opção *Update*, localizada no canto superior direito:

Na tela seguinte, selecione a opção *Replace current template* para carregar um novo arquivo, clicando no botão *Choose File*, dentro da opção *Upload a template file*. Selecione o arquivo do *template* que foi alterado e clique em *Next*.

Na próxima tela, existe a opção de alteração de algum parâmetro de entrada, que também é uma forma de atualizar a *stack*. Porém, como esse não é o caso, clique no botão *Next* para passar para o próximo passo.

Da mesma forma, é possível alterar outras configurações da *stack* nessa nova tela, mas novamente não será necessário, por isso clique em *Next* para passar para a tela com a revisão da *stack* a ser alterada.

Nessa última tela, em sua parte inferior, é exibido um relatório simplificado do que será alterado, como pode ser visto na figura a seguir:

Changes (1)				
Action ▼	Logical ID ▲	Physical ID	Resource type ▼	Replacement
Modify	Ec2Instance	i-0bcdcc7c7b76b1bc61 ↗	AWS::EC2::Instance	Conditional

Previsão de alterações da stack

Veja que o CloudFormation percebeu que a alteração no *template* afetará o instância EC2.

Para finalizar, clique em *Update* para disparar a atualização da *stack*.

Na tela de *stacks* do console do CloudFormation é possível observar o andamento da atualização da *stack*:

pcs-ec2-01					
Stack info	Events	Resources	Outputs	Parameters	Template
Events					
<input type="text"/> Search events					
Timestamp	Logical ID	Status	Status reason		
2019-06-09 15:37:26 UTC-0300	pcs-ec2-01	⌚ UPDATE_IN_PROGRESS	User Initiated		
2019-06-09 15:31:16 UTC-0300	pcs-ec2-01	☑ CREATE_COMPLETE	-		

Atualização em andamento

Perceba que o status da *stack* está em `UPDATE_IN_PROGRESS`. Quando o processo concluir, o status passará a `UPDATE_COMPLETE`, significando que todo o processo de atualização foi concluído.

Com a finalização do processo, a instância passou do tipo `t2.micro` para o tipo `t2.medium`, como especificado na alteração feita no *template* da *stack*.

	Name	Instance ID	Instance Type	Availability Zone	Instance State
<input type="checkbox"/>	pcs_ec2_01	i-0bcd7c7b76b1bc61	t2.medium	us-east-1a	running
<input type="checkbox"/>	ec2_01	i-05ced0d39023bdf1	t2.micro	us-east-1d	stopped
<input type="checkbox"/>	ec2_01	i-06137dbb1cdbf19b8	t2.micro	us-east-1b	stopped

Instância atualizada



A interrupção, ou não, do serviço provido pelo recurso criado pela *stack* obviamente depende da alteração feita no *template*.

Repare que não foi necessário apagar a instância criada anteriormente, pois ela possui a mesma identificação única. Apenas seu tipo foi alterado.

7.7 - Apagando uma stack

Quando os recursos criados por uma *stack* não são mais necessários, é possível apagá-la, e consequente, apagar todos os recursos que foram criadas por ela.

Esse recurso é interessante, principalmente quando deseja-se excluir todo um sistema e seus recursos, garantindo que nada fique para trás consumindo horas de uma conta da AWS.



Não é possível reverter o processo de exclusão de uma *stack*, na tentativa de reaver os recursos que foram apagados por tal ação.

Para proceder com a exclusão da *stack* que foi criada, selecione-a no console do AWS CloudFormation e clique na opção Delete, localizada no canto superior direito

Confirme a exclusão da *stack* para que ela e seus recursos sejam apagados.

The screenshot shows the AWS CloudFormation console with the stack named 'pcs-ec2-01'. The 'Events' tab is selected. A search bar at the top says 'Search events'. Below is a table with columns: 'Timestamp', 'Logical ID', 'Status', and 'Status reason'. There are two rows:

Timestamp	Logical ID	Status	Status reason
2019-06-09 15:40:44 UTC-0300	Ec2Instance	DELETE_IN_PROGRESS	-
2019-06-09 15:40:42 UTC-0300	pcs-ec2-01	DELETE_IN_PROGRESS	User Initiated

Processo de exclusão da stack

Após a finalização do processo, a instância EC2 criada pela *stack* entra no estado *Terminated*, o que significa que ela será totalmente excluída em alguns instantes.

Futuramente, outra *stack* pode ser criada a partir do mesmo *template*, tendo os mesmos parâmetros ou não.

7.8 - Conclusão

O assunto sobre CloudFormation é bem extenso, por diversas razões, como:

- Existem várias recursos da AWS que podem ser criados com *templates* de CloudFormation;
- Dentro de cada recurso que pode ser criado, existem muitas propriedades que podem ser configuradas;
- Ainda há a possibilidade de *stacks* dependerem de outras, lendo parâmetros de saída para configurar parâmetros de entrada.

A ideia desse capítulo foi introduzir o assunto sobre CloudFormation. Ele voltará a ser discutido nesse livro com outros exemplos envolvendo outros recursos da AWS, para que o conceito seja fixado e mais aprofundado.

O próximo capítulo lida com uma questão que está com muita evidência: **execução de aplicações em containers Docker**. Esses conceitos serão muito importantes para o capítulo 9, onde o serviço **Amazon Elastic Container Service** será apresentado.

8 - Executando aplicações em containers Docker

No capítulo 6 foi gerado um arquivo JAR executável da aplicação `aws_project01`, para que ela fosse executada em uma instância EC2, criada nesse mesmo capítulo.

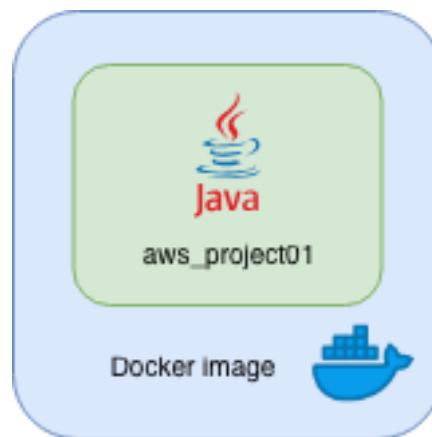
Também foi construído um *template* para a criação de outras instâncias com a mesma configuração necessária para que essa aplicação fosse executada, ou seja, ficou fácil criar novas instâncias para rodar a aplicação `aws_project01`, como foi demonstrado no capítulo 7 onde tudo foi feito utilizando CloudFormation.

Embora essa abordagem seja útil e até mesmo fácil em várias situações, ainda existe uma outra opção para execução de aplicações: utilizar **containers Docker**.

Essa estratégia traz algumas vantagens, como por exemplo:

- Melhor **gerenciamento de recursos** quando várias aplicações devem compartilhar uso de CPU e memória;
- Executar a aplicação **sempre sobre um mesmo ambiente**, evitando problemas de compatibilidade entre máquinas de desenvolvimento e produção;
- Facilitar e agilizar processos de criação de novas instâncias em cenários de configuração de **escalonamento horizontal**.

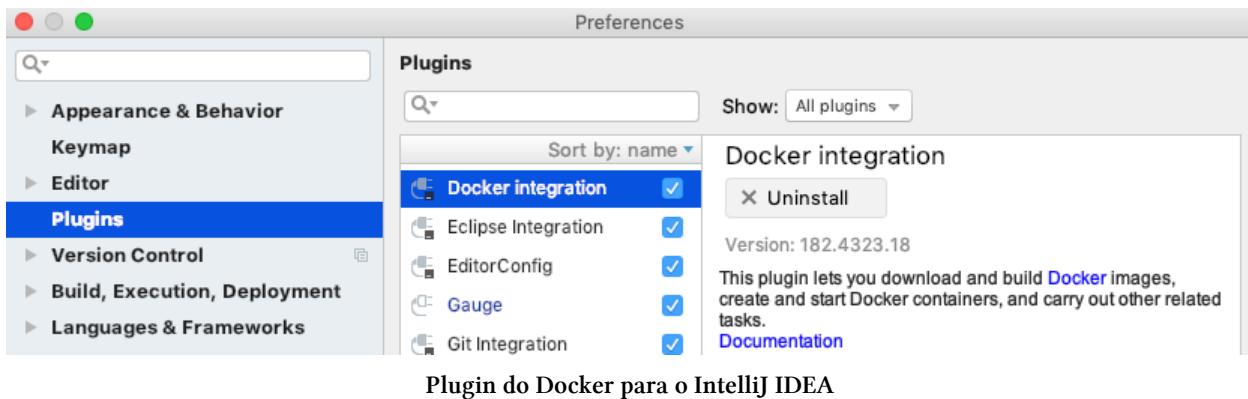
O intuito desse capítulo é fazer com que a aplicação `aws_project01` execute em um *container Docker*, para que no próximo capítulo o serviço **AWS Elastic Container Service** possa ser utilizado para rodá-la utilizando técnicas avançadas de gerenciamento de *container*, através desse serviço da AWS.



Aplicação `aws_project01` rodando em uma imagem Docker

Para isso também será necessário publicar a imagem gerada no DockerHub, assim o AWS ECS poderá baixá-la e executá-la.

O mais interessante é que não é necessário modificar o código fonte da aplicação, mas apenas adicionar algumas configurações ao processo de *build*. E para ajudar nesse processo, o IntelliJ IDEA possui um plugin que facilita todo o trabalho, fazendo com que as ações e comandos possam ser executadas através de sua interface gráfica. Para isso, verifique se ele está instalado, através de sua tela de configurações, na seção Plugins, como mostra a figura a seguir:

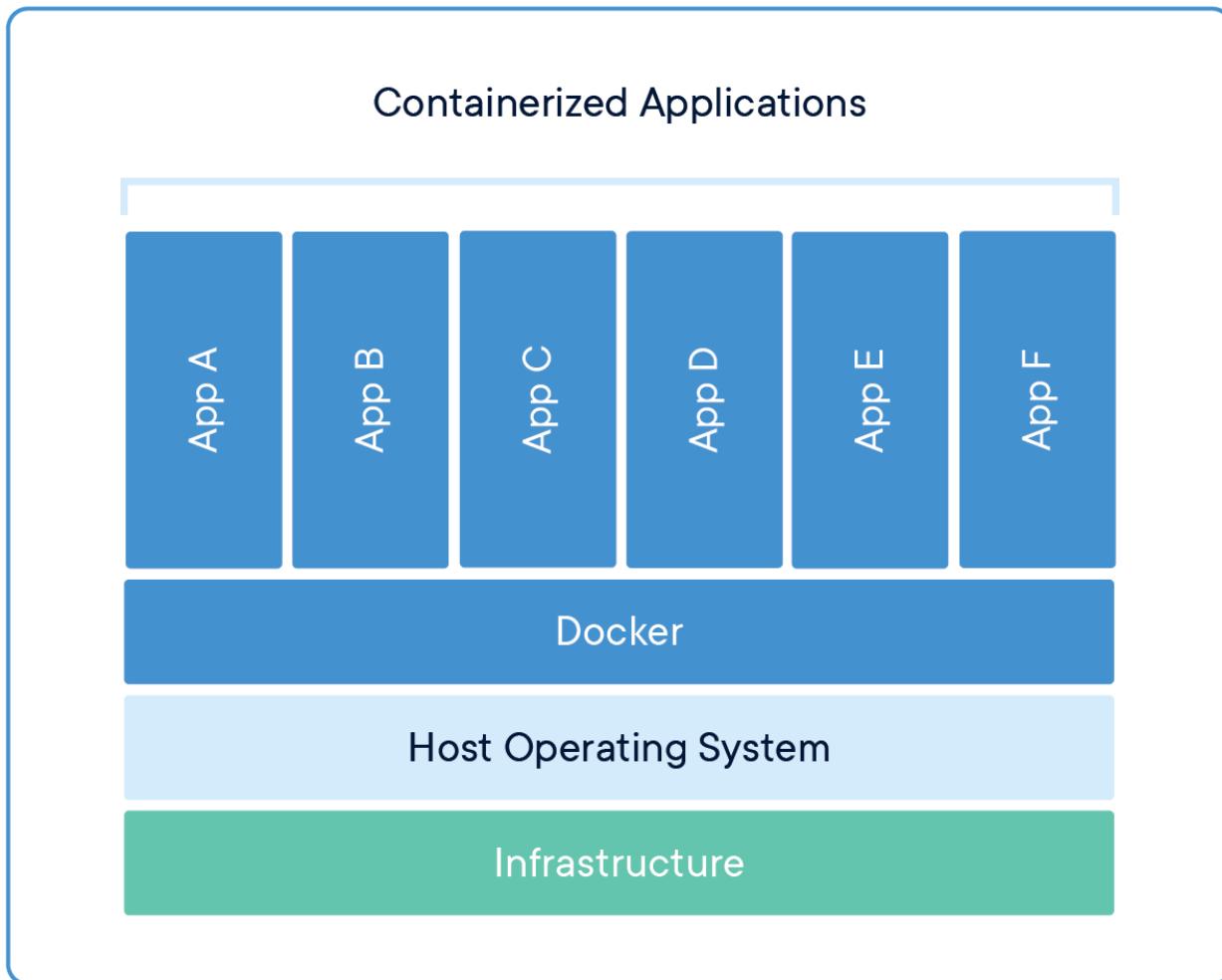


Caso esse plugin não esteja instalado, localize-o nessa mesma tela e execute a sua instalação.

Os conceitos envolvidos para se trabalhar com *containers* Docker é muito extenso. A ideia desse capítulo é apresentar alguns pontos chaves mínimos para fazer com que a aplicação `aws_project01` possa usufruir dessa tecnologia. Ao mesmo, preparar o leitor para o capítulo seguinte, onde o serviço AWS ECS será utilizado com base com o *container* que for criado aqui.

8.1 - Como funcionam containers Docker

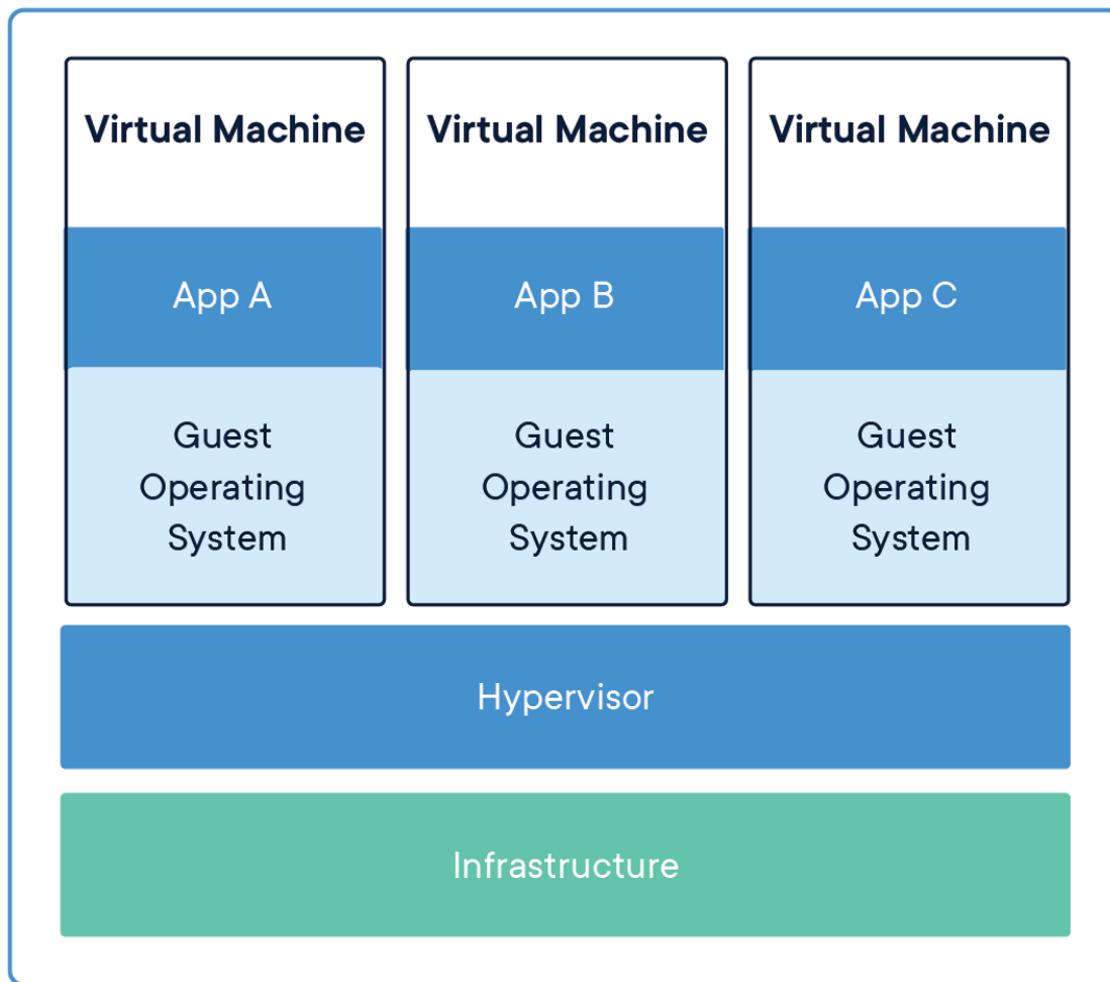
Um *container* Docker “virtualiza” o sistema operacional hospedeiro ao invés do hardware. Isso faz com que cada aplicação rodando dentro de um *container* tenha seu próprio conjunto de recursos, ou seja, do ponto de vista dessa aplicação, ela está sendo executando em um sistema operacional, sem o conhecimento de outras aplicações que possam estar sendo executadas em outros *containers*.



Como funciona o Docker. Fonte: www.docker.com

Os *containers* são executados compartilhando o kernel do sistema operacional, cada um com seu espaço de memória isolado. Essa abordagem faz com que o gerenciamento de recursos como uso de CPU, memória e rede possam ser controlados de forma única para cada *container*, sem que um interfira no outro.

Ainda sobre esse ponto vista, *containers* precisam de menos recursos para serem executados do que máquinas virtuais, que “virtualizam” o hardware para um sistema operacional completo e dessa forma, provê tudo o que é necessário para uma ou mais aplicações serem executadas dentro dessa máquina virtual.



Como funcionam as máquinas virtuais. Fonte: www.docker.com

As máquinas virtuais compartilham recursos de hardware e cada uma possui seu próprio sistema operacional, consumindo mais recursos em comparação com *containers*.

Containers e máquinas virtuais possuem finalidades distintas, ou seja, a escolha de qual caminho tomar depende muito mais do objetivo final do que simplesmente por comparações entre as duas tecnologias.

8.2 - Criando imagens Docker

Como explicado no início desse capítulo, a ideia é fazer com que a aplicação `aws_project01` seja preparada para ser executada dentro de um *container* Docker. Com isso ela poderá ser utilizada para o próximo capítulo, onde o serviço AWS ECS, que lida justamente com *containers*, será apresentado.

Os passos para criação da imagem Docker para a aplicação `aws_project01` consistem em:

- Criar um arquivo chamado `Dockerfile`, que contém instruções sobre como a imagem deve ser montada e executada;
- Adaptar o arquivo `build.gradle` para as tarefas de geração da imagem Docker;
- Utilizar o plugin do IntelliJ IDEA para a geração e publicação da imagem.

Para começar, abra o projeto `aws_project01` no IntelliJ IDEA e crie o arquivo `Dockerfile` na raiz da projeto, no mesmo nível da pasta `src`, com o seguinte conteúdo:

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG DEPENDENCY=target/dependency
COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY ${DEPENDENCY}/META-INF /app/META-INF
COPY ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java", "-cp", "app:app/lib/*", "br.com.siecola.aws_project01.AwsProject01A\pplication"]
```

Dentro desse arquivo, é importante destacar dois pontos:

- a primeira linha, com a instrução `FROM`, define a imagem base para a montagem da imagem para a aplicação `aws_project01`. Ela já contém tudo o que é necessário para execução de uma aplicação Java com o OpenJDK 8;
- a última linha, com a instrução `ENTRYPOINT`, define o ponto de entrada que deve ser executado quando a imagem subir. Aqui é necessário definir o caminho completo para a classe principal da aplicação, onde está o método `main`.

Agora abra o arquivo `build.gradle`, para a criação da tarefa de geração da imagem Docker da aplicação. Para começar, é necessário realizar algumas configurações:

a) Adicione o plugin para geração da imagem Docker

Dependendo da versão do *template* do projeto criado pelo Spring Initializr, será necessário adicionar a seguinte seção no topo do arquivo `build.gradle`:

```

buildscript {
    ext {
        springBootVersion = '2.1.2.RELEASE'
    }
    repositories {
        mavenCentral()
        maven {
            url "https://plugins.gradle.org/m2/"
        }
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:${springBootVersion}")
        classpath('gradle.plugin.com.palantir.gradle.docker:gradle-docker:0.13.0')
    }
}

```



Se tiver dúvidas de como deve ficar o arquivo `build.gradle`, consulte esse [exemplo⁹](#), respeitando as versões de seu projeto.

Para finalizar, na seção onde os *plugins* estão configurados, adicione um novo, como no trecho a seguir:

```
apply plugin: 'com.palantir.docker'
```

Esse *plugin* será necessário para que as tarefas de geração da imagem Docker possam ser executadas pelo IntelliJ IDEA.

b) Configure a variável group

Ainda no arquivo `build.gradle` há uma seção com as seguintes configurações:

```

group = '<login do desenvolvedor no DockerHub'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

```

A variável `group` deve ser configurada com o login do **DockerHub** do desenvolvedor, pois é onde essa imagem será publicada ao final do processo.

⁹https://github.com/siecola/aws_project01/blob/master/build.gradle

c) Nome e versão do executável da aplicação

Para definir o nome e a versão do executável da aplicação, **substitua** a seção `bootJar`, como no trecho a seguir:

```
bootJar {  
    baseName = 'aws_project01'  
    version = '0.1.0'  
}
```

Essa configuração definirá o nome e versão do executável da aplicação.

d) Tarefas de criação da imagem Docker

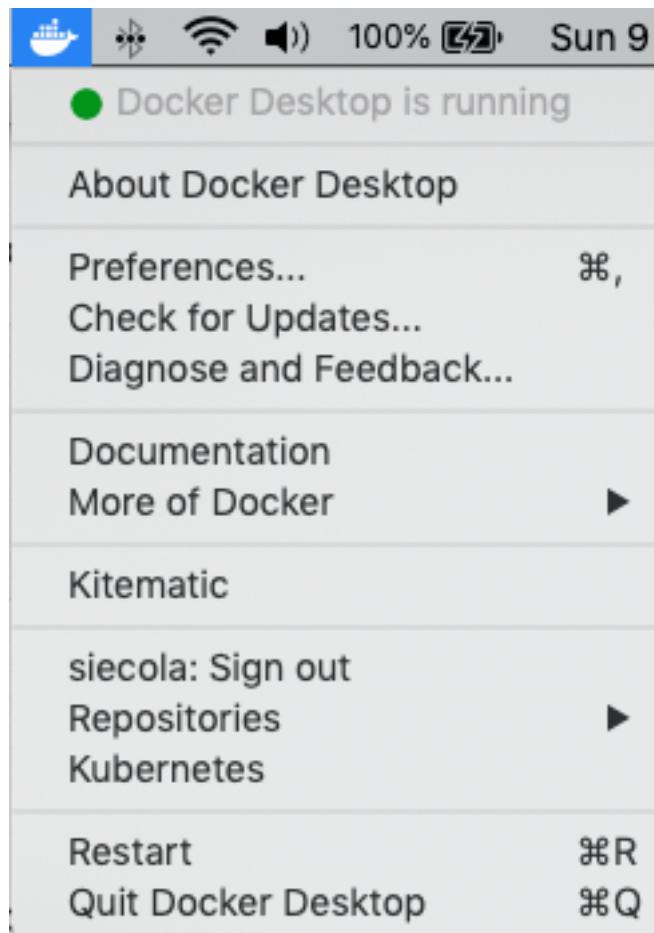
Por último, para definir as tarefas de criação da imagem Docker, adicione os trecho a segui no arquivo `build.gradle`:

```
task unpack(type: Copy) {  
    dependsOn bootJar  
    from(zipTree(tasks.bootJar.outputs.files.singleFile))  
    into("build/dependency")  
}  
docker {  
    name "${project.group}/${bootJar.baseName}"  
    tags "${bootJar.version}"  
    copySpec.from(tasks.unpack.outputs).into("dependency")  
    buildArgs(['DEPENDENCY': "dependency"])  
}
```

Esse trecho utilizará as configurações feitas anteriormente, para a geração da imagem Docker da aplicação `aws_project01`.

A configuração `tags` será utilizada para gerar versões de imagens de acordo com a versão da aplicação.

Pronto! Agora tudo está preparado para o IntelliJ IDEA gerar a imagem Docker, mas para isso é necessário executar a aplicação do Docker Desktop, instalada na seção 2.3.b do capítulo 2.

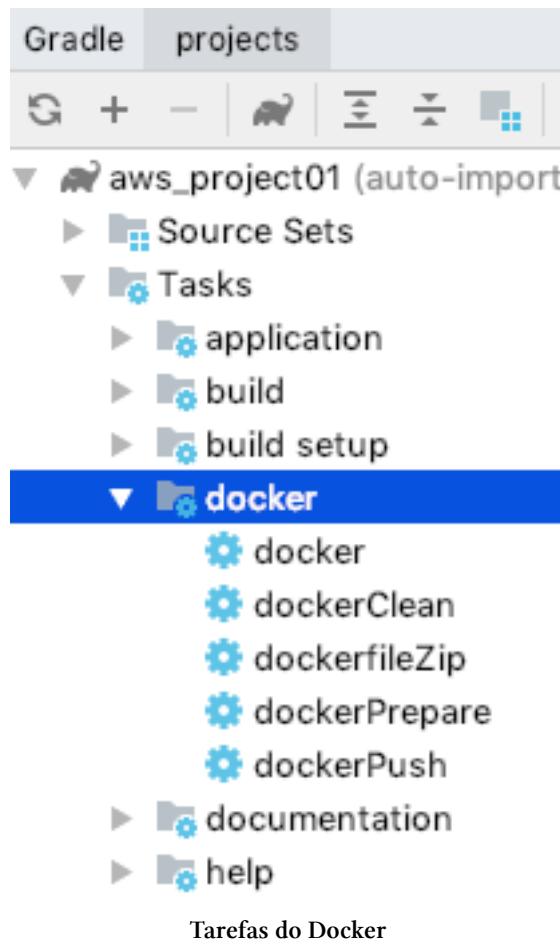


Docker em execução na máquina de desenvolvimento

É necessário manter essa aplicação em execução para que o IntelliJ IDEA possa publicar a imagem Docker gerada no DockerHub mais adiante.

Depois do Docker Desktop iniciar na máquina de desenvolvimento, proceda com os passos a seguir.

Tendo preparado tudo no projeto, para gerar a imagem Docker utilizando o IntelliJ IDEA, acesse o menu **View** -> **Tool Windows** -> **Gradle** e em seguida expanda a seção docker, como na figura a seguir:



Essas tarefas serão utilizadas no projeto `aws_project01`, para geração e publicação da imagem Docker. A seguir uma breve descrição de algumas delas:

- **dockerClean**: limpa os arquivos gerados pelas outras tarefas;
- **docker**: gera a imagem Docker da aplicação;
- **dockerPush**: publica a imagem Docker gerada.

Portanto, para gerar uma imagem Docker da aplicação, dê um duplo clique na tarefa `docker`, que o IntelliJ IDE começar o processo, como mostra o log a seguir:

```
19:33:46: Executing task 'docker'...

> Task :dockerClean UP-TO-DATE
> Task :compileJava
> Task :processResources
> Task :classes
> Task :bootJar
> Task :unpack
> Task :dockerPrepare
> Task :docker

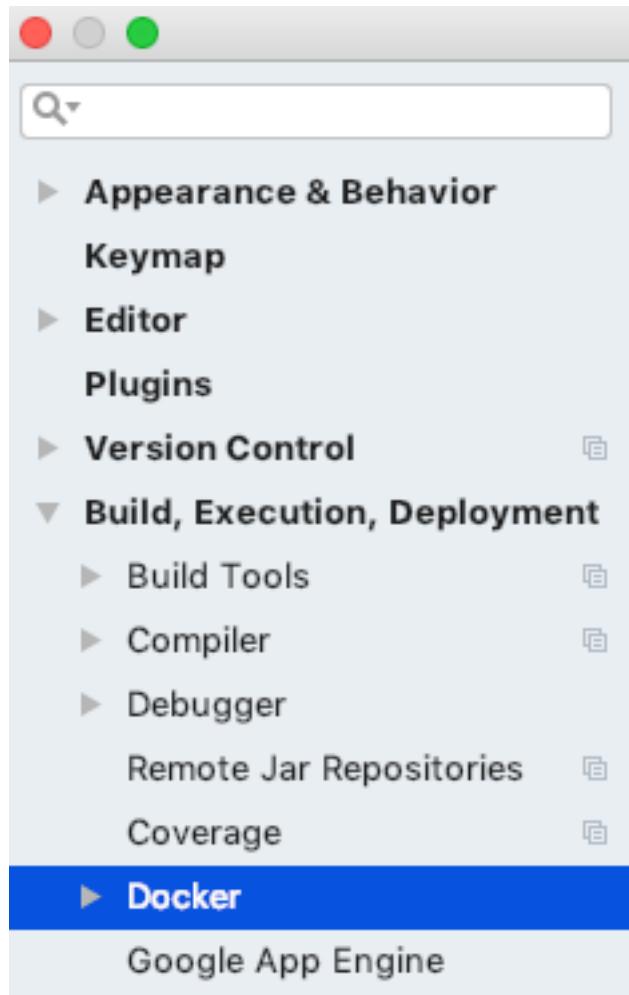
BUILD SUCCESSFUL in 1s
7 actionable tasks: 6 executed, 1 up-to-date
19:33:48: Task execution finished 'docker'.
```

A partir de agora a aplicação pode ser executada dentro de um **container**, através de sua imagem, como será mostrado na seção seguinte.

8.3 - Executando aplicações em containers Docker

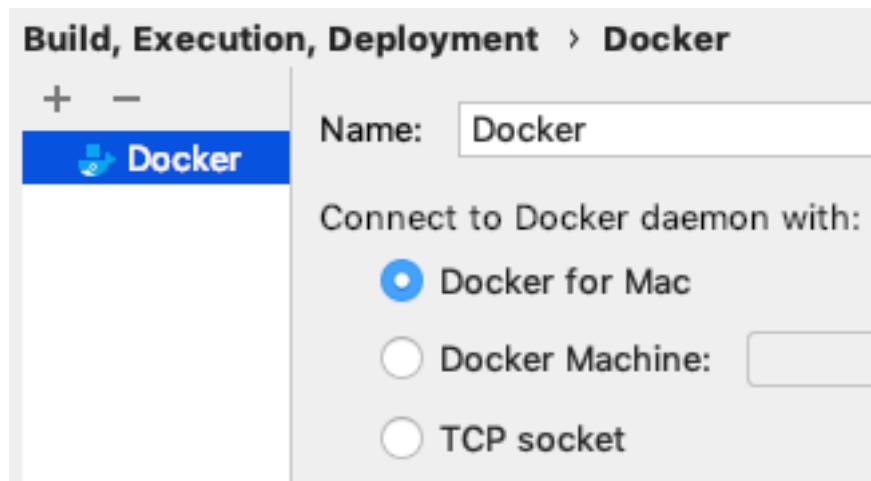
Agora que a imagem Docker da aplicação `aws_project01` foi gerada, é possível executá-la localmente. Essa estratégia é interessante para verificar seu total funcionamento dentro de um *container*. Felizmente, o IntelliJ IDEA também possui todas as ferramentas para isso.

Para começar, é necessário conectar o IntelliJ IDEA ao Docker da máquina de desenvolvimento. Isso pode ser feito acessando suas configurações, no menu `Preferences\Settings -> Build, Execution, Deployment`, na seção Docker, como pode ser visto na figura a seguir:



Conectando o IntelliJ no Docker local

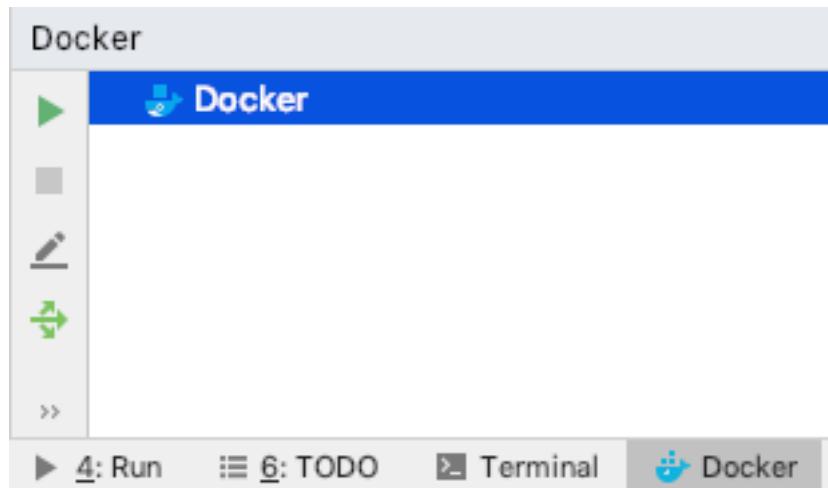
Nessa tela, clique no botão + localizado na parte superior para adicionar uma configuração como mostra a figura a seguir:



Adicionando uma configuração do Docker

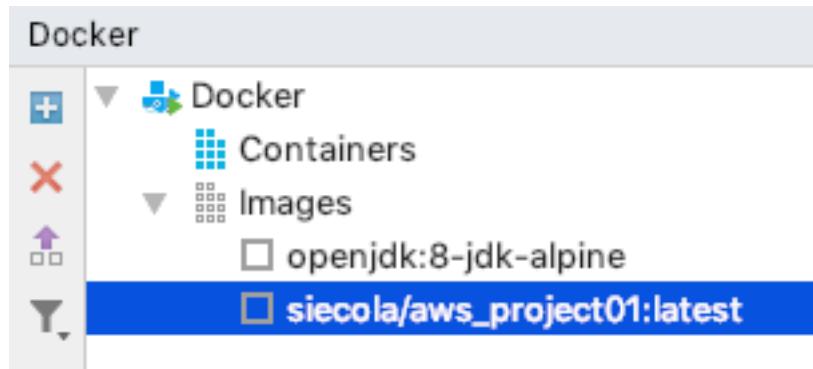
Obviamente, essa tela pode variar de acordo com o sistema operacional utilizado.

Para finalizar, clique no botão OK para salvar a configuração e fechar a tela. Em seguida acesse o menu View -> Tool Windows -> Docker para abrir o gerenciador de imagens e *containers* Docker no IntelliJ:



Gerenciador de imagens e containers

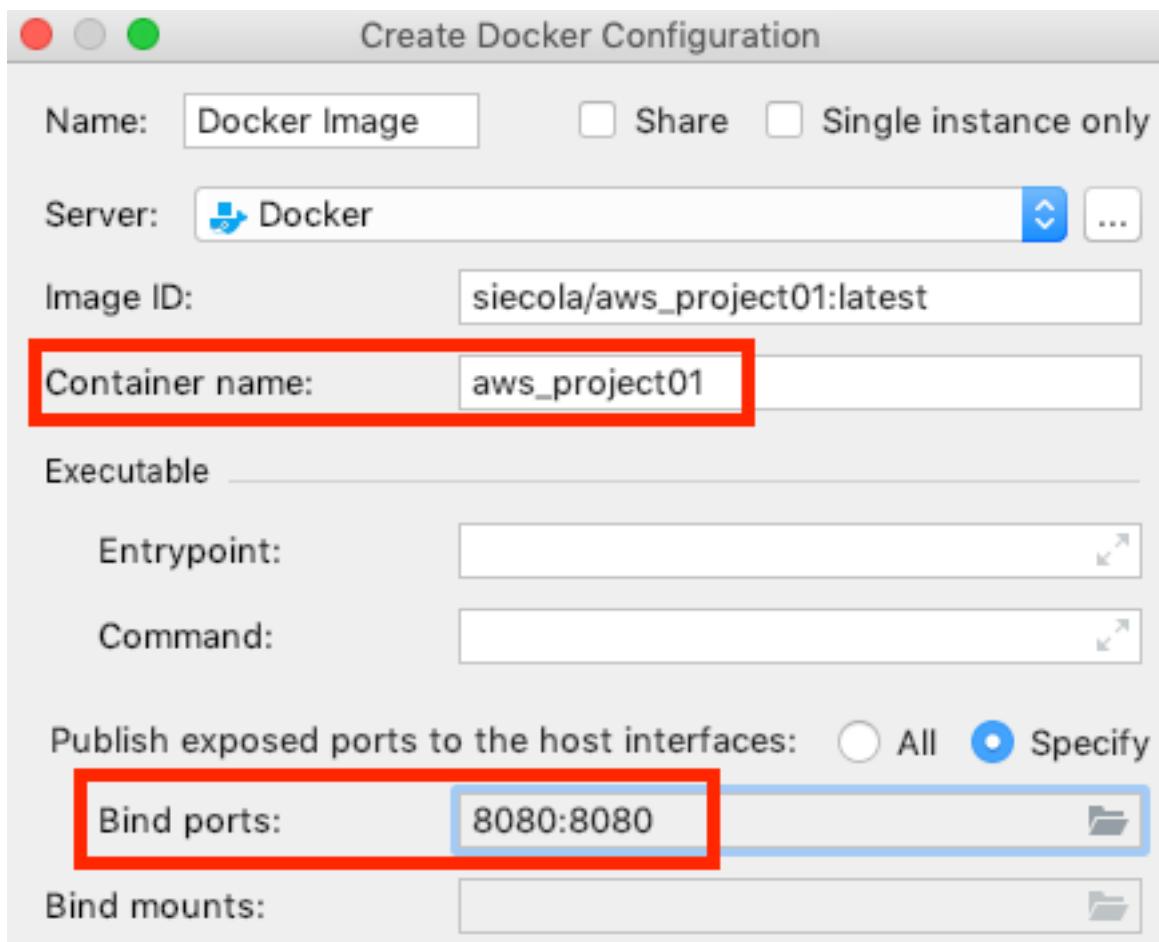
Pressione o botão *play* localizado no canto superior esquerdo da tela para conectar o IntelliJ ao Docker local:



Imagens no Docker local

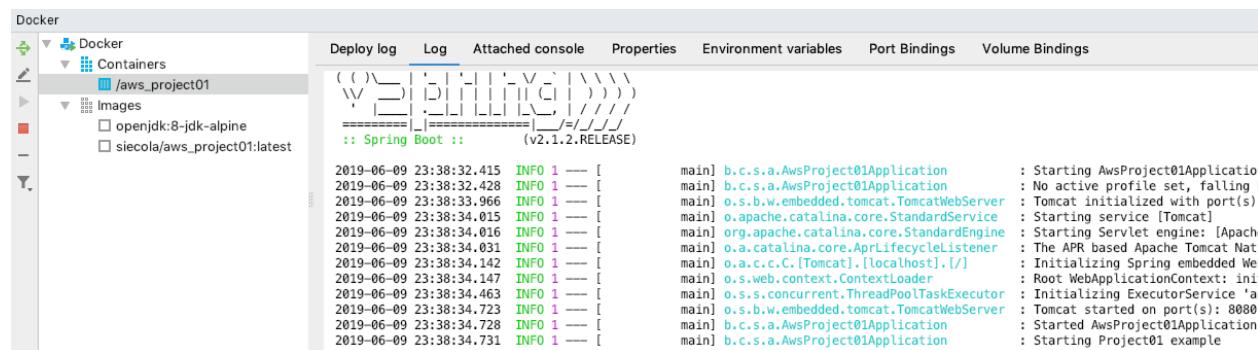
Perceba que a imagem da aplicação `aws_project01`, que foi gerada na seção anterior, já é exibida, bem como a imagem base que foi utilizada, a OpenJDK 8.

Para criar um *container* para executar a imagem do projeto `aws_project01`, clique com o botão direito e selecione a opção `Create container -> Create...`. Na tela que aparecer, configure o nome do *container* e o mapeamento de portas, conforme a figura a seguir:



Configurando o container

Em seguida clique em Run para criar o *container* e já executá-lo. Isso fará com que a aplicação entre em execução e seus logs serão exibidos, como mostra a figura a seguir:



Container em execução

Nesse momento, é possível acessar a aplicação através do Postman, da mesma forma que ele estivesse sendo executada de dentro do IntelliJ IDEA, como foi feito no capítulo 5, porém a aplicação está sendo executando dentro de um *container* Docker, capaz de ser executada em qualquer serviço gerenciador

de *containers*, como o AWS ECS, que será visto no próximo capítulo.

Para acompanhar os logs de execução da aplicação, acesse a aba Attached console.

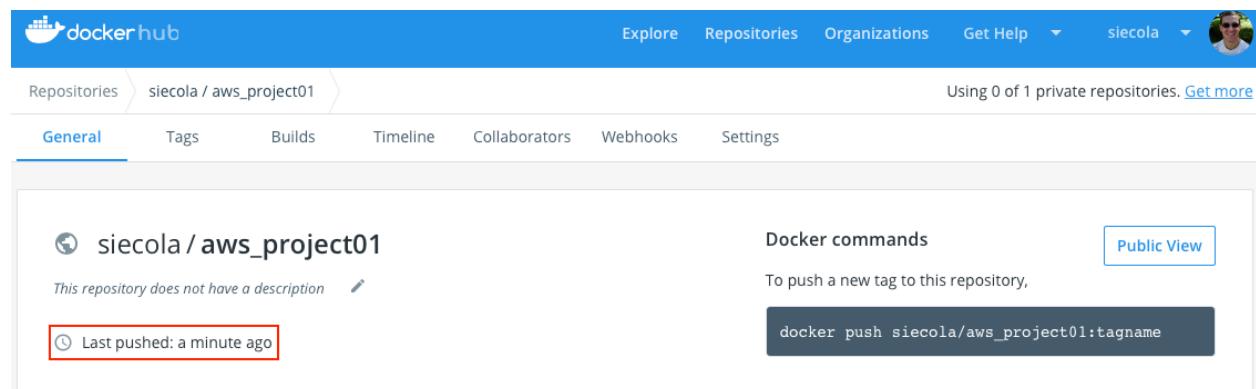
8.4 - Docker Hub

Agora que a imagem Docker da aplicação `aws_project01` foi criada, é possível publicá-la no DockerHub. Isso permitirá que o AWS ECS possa acessá-la para executá-la em um *container* na AWS. Porém, isso também é interessante para que outras pessoas possam baixa-la e executá-la em suas máquinas.

Para começar, é necessário criar um repositório no DockerHub para acolher a imagem do projeto `aws_project01`. Para isso, vá até o site <https://hub.docker.com>, logue com sua conta e acesse opção Create Repository. Nessa tela, digite `aws_project01` para o nome do repositório, torne-o público e aperte o botão Create.

Depois que o repositório público for criado no DockerHub, será possível subir a imagem criada na seção anterior para ele.

Para publicar a imagem no DockerHub, basta acessar o menu View -> Tool Windows -> Gradle e executar a tarefa `dockerPush0.1.0`, dentro da seção docker. Nesse momento o IntelliJ começa o processo de publicação da imagem e ao final, ela estará publicada no DockerHub:



The screenshot shows the DockerHub interface for the repository `siecola / aws_project01`. At the top, there are navigation links for Explore, Repositories, Organizations, Get Help, and a user profile. Below the header, it says "Using 0 of 1 private repositories. [Get more](#)". The main content area has tabs for General, Tags, Builds, Timeline, Collaborators, Webhooks, and Settings. Under the General tab, it displays the repository name and a note: "This repository does not have a description". It also shows the last push time: "Last pushed: a minute ago". To the right, there's a "Docker commands" section with a "push" button containing the command `docker push siecola/aws_project01:tagname`. There's also a "Public View" button. The entire screenshot is labeled "Imagen publicada no DockerHub" at the bottom.

Dessa forma, qualquer pessoa pode executar a aplicação em sua máquina local, através do comando a seguir:

```
docker run siecola/aws_project01:0.1.0
```

Repare que é possível especificar a versão da aplicação através do nome da tag existente no repositório da aplicação. Nesse caso, a versão que foi gerada é a **0.1.0**.

Obviamente, a aplicação ainda é bem simples, mas ideia aqui é mostrar como ela pode ser executada dentro de um *container* Docker, sem a necessidade de nenhuma configuração ou instalação adicional, além do Docker, obviamente.



É importante lembrar que o código do projeto está no GitHub e pode ser acessado [aqui¹⁰](#).

8.5 - Conclusão

Esse foi um capítulo introdutório sobre como criar uma imagem Docker para a aplicação `aws_project01`, além de executá-la na máquina local e publica-la no DockerHub.

Trabalhar com *containers* é algo de extrema importância para uma arquitetura com microsserviços, principalmente num ambiente de *cloud computing*.

No próximo capítulo será apresentado o **AWS Elastic Container Service**, um serviço da AWS para gerenciamento de *containers*, onde a imagem Docker criada para a aplicação `aws_project01` será executada em um *container* gerenciado por esse serviço.

¹⁰https://github.com/siecola/aws_project01

9 - Amazon Elastic Container Service

No capítulo anterior a aplicação `aws_project01` foi preparada para ser executada dentro de um *container* Docker e uma imagem foi gerada e publicada no DockerHub, o repositório de imagens público do Docker.

Essa imagem será utilizada neste capítulo para demonstrar como o serviço Elastic Container Service da AWS funciona como gerenciador e orquestrador de *containers* Docker, utilizando clusters de instâncias EC2.

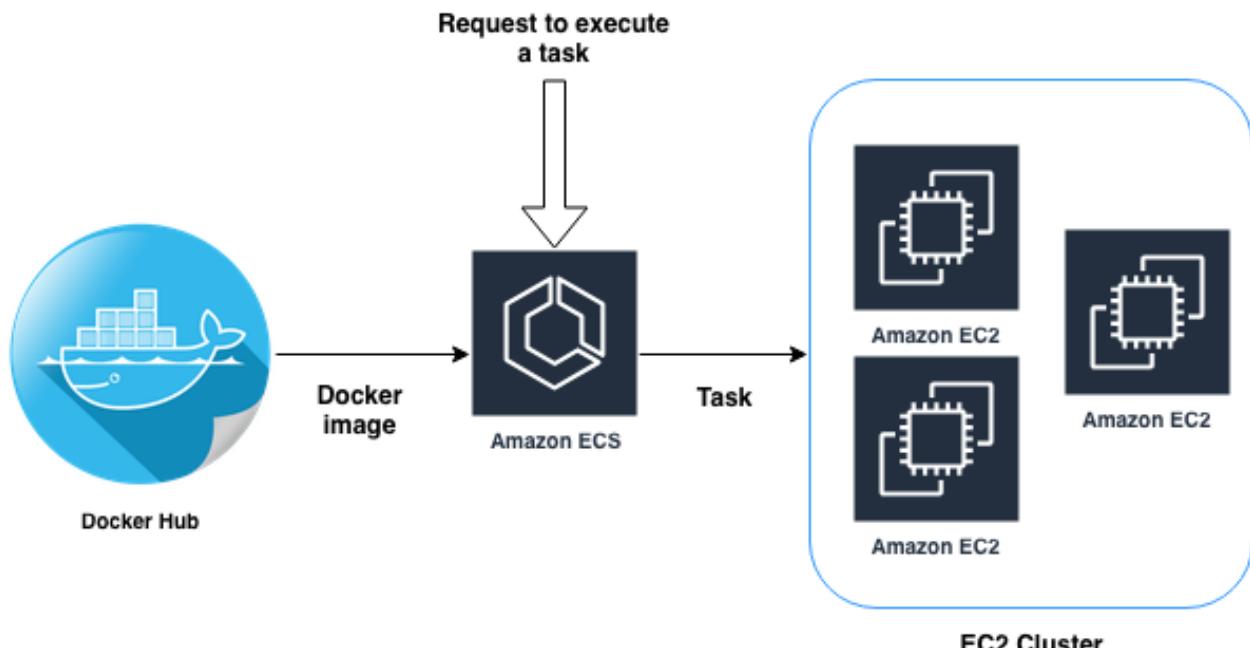


Diagrama de execução de uma tarefa

O AWS ECS permite a execução de *containers* Docker através de definições de tarefas, que podem ser controladas por serviços capazes de monitorá-las durante toda a sua execução.

Como a aplicação `aws_project01` já foi preparada para ser executada em um *container* Docker, **não será necessário criar e gerenciar instâncias EC2** para sua execução, tão pouco preocupar-se em alocar uma quantidade fixa delas para atender uma demanda mínima de requisições.

A seguir, algumas vantagens de se utilizar o AWS ECS para execução de aplicações baseadas em *containers* Docker:

- Não é necessário criar e gerenciar instâncias EC2;
- Um mesmo *cluster* de instâncias EC2 pode ser utilizado para execução de aplicações distintas;

- **Várias instâncias de uma aplicação** podem ser executadas no *cluster*, escalando a partir de parâmetros como consumo de CPU por tarefa sendo executada;
- A versão de uma aplicação pode ser trocada, sem afetar os serviços por ela oferecido.

A fonte da imagem Docker utilizada pelo ECS pode ser configurada para uma das seguintes opções:

- **Amazon EC2 Container Registry**: um serviço da AWS para armazenamento de imagens Docker;
- **Docker Hub**: o repositório gratuito da Docker para armazenamento de imagens Docker.

Nesse livro será utilizado o Docker Hub, como repositório da imagem Docker que será executada.

A seguir serão detalhados os componentes principais do AWS ECS, a partir da criação de um *cluster* para execução da aplicação `aws_project01`, utilizando sua imagem Docker que foi gerada no capítulo anterior.



Os recursos que serão criados nesse capítulo são passíveis de cobrança, dependendo da conta utilizada AWS pelo leitor, incluindo as instâncias EC2 e o *application load balancer*. As cobranças podem ser feitas por horas de utilização, assim como a quantidade de requisições realizadas.

9.1 - Cluster

Um *cluster* é um grupo lógico de instâncias EC2 preparadas para a execução de *containers* Docker. Ele contém as seguintes características:

- Podem contar várias instâncias EC2 de um tipo especificado durante a sua criação;
- Todas as instâncias são alocadas na mesma região;
- As instâncias podem ser compartilhadas para execução de tarefas distintas;
- Todas as instâncias são capazes de executar imagens Docker.

Vários *clusters* podem ser criados, afim de promover uma separação lógica entre os tipos de suas aplicações, como por exemplo: *cluster* para o *back-end* e *cluster* para o *front-end*.

9.1.1 - O que é o ECS Agent

O ECS Agent está presente em todas as instâncias de um *cluster* ECS. Ele é o responsável por gerenciar o estado de cada uma dessas instâncias. Além disso ele estabelece a comunicação entre o ECS e o *daemon* do Docker dentro da instância EC2.

É através do ECS Agent que uma imagem Docker entra em execução, quando solicitada pelo ECS.

9.1.2 - Criando um ECS cluster

Para começar o processo de criação de um *cluster*, acesse seu console na AWS, através da opção Services → ECS, como mostra a figura a seguir:

The screenshot shows the AWS navigation bar with 'Services' and 'Resource Groups' dropdowns. Below the navigation bar, the 'Amazon ECS' section is expanded, showing 'Clusters' (which is highlighted with an orange border), 'Task Definitions', and 'Account Settings'. To the right of this sidebar, the main content area is titled 'Clusters'. It contains a brief description: 'An Amazon ECS cluster is a regional grouping of one instance type.' Below the description are two buttons: a blue 'Create Cluster' button and a grey 'Get Started' button. At the bottom of the main content area, the text 'Criando um cluster' is visible.

Clique no botão Create Cluster para começar com o processo e escolher os parâmetros do *cluster* de instâncias EC2 que será utilizado para prover o serviço responsável por executar a aplicação aws_project01.

A primeira página exibe os *templates* disponíveis para a criação do *cluster*. Escolha a opção EC2 Linux + Networking, que irá criar os seguintes recursos:

- O *cluster* para gerenciar as instâncias EC2;
- Uma VPC e uma subrede para estabelecer a comunicação entre com as instâncias EC2;
- Um grupo para abrigar as configurações de *auto scaling*.

Para continuar, clique no botão Next step. A tela seguinte exibe várias opções que podem ser configuradas para a criação do *cluster*. Porém, para o exemplo desse capítulo serão necessários configurar apenas os seguintes parâmetros:

- Nome do *cluster*
- Tipo das instâncias EC2 dentro do *cluster*
- Número de instâncias EC2 desse tipo
- Porta TCP a ser liberada para o tráfego de entrada.

Obviamente os demais parâmetros podem ser configurados, caso um usuário avançado tenha conhecimento dessas configurações adicionais.

Então, para começar a alterar esses parâmetros nessa página, digite o nome do *cluster* a ser criado, como mostra a figura a seguir:

Configure cluster

Cluster name* i

Create an empty cluster

Configurando o nome do cluster

As próximas opções que devem ser configuradas são o tipo e a quantidade das instâncias que serão gerenciadas por esse *cluster*:

EC2 instance type* i

Manually enter desired instance type

Enable T2 unlimited i

Number of instances* i

Tipo e quantidade de instâncias EC2

Para essas opções, escolha o tipo **t2.micro**, o mesmo utilizado no capítulo 6 para execução da aplicação `aws_project01`. Ainda, configure para que o *cluster* possa ter 2 instâncias desse tipo.

A última configuração a ser feita nessa página é a porta que deve ser liberada para tráfego de entrada. Esse parâmetro está localizado na seção *Networking*. Ela será a mesma utilizada no capítulo 6 pela aplicação `aws_project01`, ou seja, a 8080:



Porta de entrada

Tendo realizado essas configurações, clique no botão *Create* para efetivamente criar o *cluster*. Após alguns segundos, os recursos serão criados e um relatório como o da figura a seguir será exibido:

Instance type	t2.micro
Desired number of instances	2
Key pair	
ECS AMI ID	ami-0c09d65d2051ada93
VPC	vpc-036f74eada12d341b
Subnet 1	subnet-008f2c14585c2ec6a
Subnet 1 route table association	rtbassoc-000975a2ecac8eab2
Subnet 2	subnet-010480fae238b235a
Subnet 2 route table association	rtbassoc-05dd01fdf85c7fbf2
VPC Availability Zones	us-east-1a, us-east-1b, us-east-1c, us-east-1d, us-east-1e, us-east-1f
Security group	sg-006ee2d49de6b4db6
Internet gateway	igw-0acd91948adc61262
Route table	rtb-0e1cd9d549181ba85
Amazon EC2 route	EC2Co-Publi-1KRH6XTM5OKF6
Virtual private gateway attachment	EC2Co-Attac-1OZ2VBKRAY4MF
Launch configuration	EC2ContainerService-cluster-01-EcsInstanceLc-6BNSM3GWUTC1
Auto Scaling group	EC2ContainerService-cluster-01-EcsInstanceAsg-1U22PYPMXB26A

Recursos criados

Clique no botão *View Cluster*, localizado no canto superior dessa última, para visualizar o *cluster* que foi criado utilizando o console da AWS:

The screenshot shows the AWS Elastic Container Service (ECS) console interface. The top navigation bar has tabs for 'Services', 'Tasks', 'ECS Instances' (which is highlighted with an orange border), 'Metrics', and 'Scheduled Tasks'. Below the navigation is a blue button labeled 'Scale ECS Instances' and a dropdown menu labeled 'Actions'. A status filter section shows 'Status: ALL ACTIVE DRAINING'. A search bar below it says 'Filter by attributes (click or press down arrow to view filter options)'. The main content area is a table titled 'Instâncias do cluster' (Cluster Instances). The table has columns: 'Container Instance' (with checkboxes), 'EC2 Instance', and 'Availability Zone'. Two rows are present: one for an instance in 'us-east-1a' and another for an instance in 'us-east-1b'.

	Container Instance	EC2 Instance	Availability Zone
<input type="checkbox"/>	0e6b9c42-37d5-49a2-b38...	i-07dbd48dc90...	us-east-1a
<input type="checkbox"/>	4188a844-3a1c-4372-82cf...	i-0c2716e95ea...	us-east-1b

Essas são as instâncias que foram criadas e que serão gerenciadas pelo *cluster*. Ele as utilizará para a execução de tarefas, organizando da melhor forma para acomodar cada execução de acordo com os requisitos de CPU e memória necessários. Essas instâncias também podem ser monitoradas pelo console da AWS, na seção EC2.

Apesar do *cluster* já ter sido criado, ainda não há nenhuma aplicação em execução. Isso será feito na seção seguinte, com a criação de tarefas.

9.2 - Tarefas

As tarefas são responsáveis por instalarem o *container* Docker nas instâncias EC2 do *cluster*. Elas possuem configurações como:

- **vCPU:** número inteiro que define quantas unidades de CPU a tarefa terá alocada para sua execução;
- **Memória:** o quanto de memória ficará reservado para a execução da tarefa.

Também é possível controlar a quantidade de tarefas em execução em um *cluster*.

9.2.1 - Criando uma definição de tarefa

A definição de uma tarefa é um *template* que serve para executar uma tarefa. Com ela é possível definir:

- A imagem Docker a ser definida para cada *container*;
- A quantidade de memória e CPU de cada *container*;
- Requisitos de rede e portas de comunicação.

Para melhor entendimento, é possível fazer uma comparação entre uma imagem Docker e uma definição de uma tarefa:

- A imagem Docker é uma **foto do código-fonte**, bem como suas dependências, em um determinado momento do tempo;
- A definição de uma tarefa é uma **foto da configuração para a execução de uma imagem Docker**, em um determinado momento do tempo.

Como o propósito aqui é executar a aplicação `aws_project01` dentro do *cluster* criado anteriormente, utilizando a imagem Docker criada no capítulo 8, será necessário então criar uma definição de uma tarefa. Para isso, dentro do console da AWS para ECS, escolha a opção **Task Definitions**, localizado no canto superior esquerdo. Nessa página, clique no botão **Create new Task Definition**.

Na primeira página de criação de definição de tarefa, escolha a opção **EC2** e clique em **Next step**.

Nessa segunda página é onde tudo será configurado na definição da tarefa, incluindo a definição do *container* Docker para a execução da aplicação `aws_project01`, utilizando sua imagem que foi gerada no capítulo 8.

Na primeira seção dessa página, configure o nome da task e o modo de operação da rede para **Bridge**, como mostra a figura a seguir:

Configure task and container definitions

A task definition specifies which containers are included in your task and how they interact with each other. volumes for your containers to use. [Learn more](#)

Task Definition Name* task-01 i

Requires Compatibilities* EC2

Task Role Select a role... ▼ ⟳ i
Optional IAM role that tasks can use to make API requests to authorized AWS services. Create an Amazon Elastic Container Service Task Role in the [IAM Console](#) ↗

Network Mode Bridge ▼ i
If you choose <default>, ECS will start your container using Docker's default networking mode, which is Bridge on Linux and NAT on Windows.
<default> is the only supported mode on Windows.

Nome da definição da tarefa

Na primeira vez que um *cluster* é criado, será também criado um *IAM role*, ou seja, um papel que a tarefa irá assumir, com permissões básicas para acessar os serviços da AWS. Esse papel poderá ser customizado e também poderá ser reutilizado para criação de outros *clusters* no futuro.

Na terceira seção dessa página, chamada de *Task size*, é possível configurar os requisitos de CPU e memória que a tarefa irá necessitar. Obviamente isso depende da aplicação que ela irá executar. No caso da aplicação `aws_project01`, as configurações mostradas na figura a seguir são suficientes:

Task size

[?](#)

The task size allows you to specify a fixed size for your task. Task size is required for tasks using the Fargate launch type and is optional for the EC2 launch type. Container level memory settings are optional when task size is set. Task size is not supported for Windows containers.

Task memory (MiB) The amount of memory (in MiB) used by the task. It can be expressed as an integer using MiB, for example 1024, or as a string using GB, for example '1GB' or '1 gb'.

Task CPU (unit) The number of CPU units used by the task. It can be expressed as an integer using CPU units, for example 1024, or as a string using vCPUs, for example '1 vCPU' or '1 vcpu'.

Task memory maximum allocation for container memory reservation



Requisitos de CPU e memória da tarefa



É importante ressaltar que os valores de CPU e memória escolhidos para a tarefa devem ser compatíveis com as instâncias EC2 disponíveis no *cluster* onde ela será executada, ou seja, não se pode escolher mais memória ou unidades de CPU do que uma instância EC2 *cluster* possui.

O ECS irá alocar a tarefa no *cluster* escolhido, baseado nos requisitos de CPU e memória dessa tarefa.

A próxima seção, chamada de **Container Definitions**, se refere, como o próprio nome diz, às definições do *container* Docker que a tarefa irá executar. Clique no botão **Add container** para abrir a janela de configurações.

Na janela que se abrir, na seção **Standard**, configure o nome do container, bem como o endereço da imagem Docker a ser buscada no DockerHub, semelhante a da figura a seguir:

▼ Standard

Container name*	aws_project01
Image*	siecola/aws_project01:0.1.0

Imagen para o container Docker



Troque a primeira parte do endereço da imagem para o seu usuário do DockerHub, onde a imagem foi publicada no capítulo 8.

Ainda nessa seção, configure as portas TCP que deverão ser liberadas para o *container*, como mostra a figura a seguir:

Port mappings	Host port	Container port	Protocol
	8080	8080	tcp ▾

+ Add port mapping

Porta do container



O ideal seria definir uma porta aleatória para o *host*. A ideia deixar esse valor fixo é facilitar a didática no decorrer desse e de outros capítulos.

A última configuração a ser feita nessa página está localizada na seção *Storage and logging*. Aqui é necessário habilitar a injeção dos logs da *container* no CloudWatch, como na figura a seguir:

The screenshot shows the 'Log configuration' section of the AWS CloudWatch Logs interface. It includes a checked checkbox for 'Auto-configure CloudWatch Logs'. Below it, the 'Log driver' dropdown is set to 'awslogs'. Under 'Log options', there are three key-value pairs: 'awslogs-group' with value '/ecs/task-01', 'awslogs-region' with value 'us-east-1', and 'awslogs-stream-prefix' with value 'ecs'. There is also a 'Add key' button and a 'Logs do container' section.

Logs do container

É necessário habilitar a opção Auto-configure CloudWatch Logs, bem como selecionar o *driver* awslogs e deixar as demais caixas de configuração como mostra a figura anterior.

O CloudWatch é um serviço da AWS que será detalhado no capítulo 10.

Depois de feitas todas essas configurações nessa página, clique no botão Add para adicionar esse *container* à definição da tarefa que está sendo criada.

Repare que agora existe uma configuração de *container* associada à definição da tarefa que está sendo criada, como mostra a figura a seguir:

Container Definitions

The screenshot shows the 'Container Definitions' table. It has columns for 'Container Name', 'Image', 'Har...', 'CPU Units', 'GPU', and 'Essential'. A single row is present with the name 'aws_project01', image 'siecola/aws_project01:0.1.0', and essential status 'true'.

Container Name	Image	Har...	CPU Units	GPU	Essential
aws_project01	siecola/aws_project01:0.1.0	--/--			true

Definição do container

Veja que a imagem utilizada contém a versão da aplicação que foi gerada no capítulo 8, ou seja, a versão 0.1.0.

Essas são as configurações mínimas necessárias para a criação da definição da tarefa, por isso clique no botão Create.

Após alguns instantes a definição da tarefa é criada e aparece na lista do menu Task Definitions.

Caso alguma configuração tenha que ser alterada nessa definição de tarefa, como a alteração da versão da imagem do Docker a ser utilizada, é necessário criar uma nova nova revisão dela, ou seja, a primeira revisão da definição da tarefa não pode ser alterada. Esse processo será demonstrado mais adiante nesse capítulo.

9.2.2 - Executando uma tarefa

Agora que uma definição de tarefa foi criada, é possível executá-la. O processo de executar uma tarefa consiste, do ponto de vista do ECS, em:

- Encontrar instâncias EC2 dentro do *cluster* capazes de executar a tarefa em termos de quantidade de CPU e memória disponíveis;
- Definir como as tarefas serão distribuídas pelas instâncias EC2 do *cluster*;
- Instruir o ECS Agent e o *daemon* Docker para executar a imagem Docker definida na tarefa para ser executada nas instâncias EC2 escolhidas para tal.

Para isso, volte ao menu que mostra todos os *clusters* que foram criados:

The screenshot shows the AWS ECS console interface. On the left, there is a sidebar with various links: 'Amazon ECS' (selected), 'Clusters' (highlighted with a red box), 'Task Definitions', 'Account Settings', 'Amazon EKS', 'Clusters' (under EKS), 'Amazon ECR', 'Repositories', 'AWS Marketplace', 'Discover software', and 'Subscriptions'. The main content area is titled 'Clusters' and contains the following text: 'An Amazon ECS cluster is a regional grouping of one instance type.' Below this is a 'Create Cluster' button and a 'Get Started' button. At the bottom, there is a 'View' dropdown set to 'list' and a 'card' button. A list of clusters is shown, with 'cluster-01 >' highlighted with a red box. To the right of 'cluster-01' are two status indicators: 'CloudWatch monitoring' and 'Default Monitoring' (with a checked checkbox). Below the cluster list, the text 'Listando os clusters' is visible.

Como mostra a figura anterior, existe o *cluster* de nome *cluster-01*. Clique nele para abrir sua página de gerenciamento.

Nessa página, na aba Tasks, é possível iniciar uma nova tarefa para ser executada por esse *cluster*. Para isso, clique no botão Run new task.

Na página de configuração da tarefa, escolha o tipo EC2 para a instância e a definição da tarefa criada na seção anterior, além do *cluster* onde ela deve ser executada, como mostra a figura a seguir:

Run Task

Select the cluster to run your task definition on and the

Launch type FARGATE EC2

Task Definition task-01:1 ▾

Cluster cluster-01 ▾

Number of tasks 1

Preparando a tarefa para ser executada

Ainda nessa página, na seção de opções avançadas, é possível observar que o papel de nome `ecsTaskExecutionRole` está sendo utilizado para definir as permissões que a tarefa possui ao ser executada. Esse papel será alterado em capítulos mais adiante para inclusão de novas permissões.

▼ Advanced Options

Task Overrides

Task Role - current [ecsTaskExecutionRole](#)

Task Role - override

None

Optional IAM role that tasks can use to make API requests to authorized AWS services. Create an Amazon Elastic Container Service Task Role in the [IAM Console](#)



Task Execution Role [ecsTaskExecutionRole](#)

- current

Task Execution Role

- override

None

Container Overrides

▶ aws_project01

Papel para execução da tarefa

No final dessa seção, é possível observar que a definição do *container* de nome `aws_project01`, criada na seção anterior, será utilizada para a execução dessa tarefa. Isso significa que a aplicação `aws_project01` será executada a partir de sua imagem Docker hospedada no DockerHub.

Agora que tudo já foi configurado para a execução da tarefa, clique no botão Run Task localizado no canto inferior direito da página. Com essa ação, o processo de execução da tarefa se inicia, sendo exibido no console de administração do ECS, como pode ser visto na figura a seguir:

The screenshot shows the AWS Elastic Container Service (ECS) Tasks page. At the top, there are tabs: Services, Tasks (which is selected and highlighted in orange), ECS Instances, Metrics, Scheduled Tasks, and Tags. Below the tabs are buttons for 'Run new Task' (blue), 'Stop' (gray), 'Stop All' (gray), and 'Actions'. A status message says 'Desired task status: Running' (blue button) and 'Stopped' (gray). There is a 'Filter in this page' input field and a 'Launch type' dropdown set to 'ALL'. A table lists tasks with columns: Task (checkbox), Task definition, Container instance, and Last status. One row is highlighted with a red box around the 'Task' column value '26569267-0da4-4391-8...'. The table header includes a checkbox column and the column names.

	Task	Task definition	Container instance	Last status
<input type="checkbox"/>	26569267-0da4-4391-8...	task-01:1	9e1a81fb-df1d-47c2...	RUNNING

Tarefa em execução

Clique no link com a identificação da tarefa, como ressaltado na figura anterior, para abrir a página com informações da tarefa em execução.

Na página que detalha a tarefa que está em execução, é possível observar alguns detalhes, como mostra a figura a seguir:

Details	Tags	Logs
Cluster	cluster-01	
Container instance	9e1a81fb-df1d-47c2-b47b-66c75cee0c8f	
EC2 instance id	i-00e2e72663d5299fa	
Launch type	EC2	
Task definition	task-01:1	
Group	family:task-01	
Task role	ecsTaskExecutionRole	
Last status	RUNNING	
Desired status	RUNNING	
Created at	2019-07-28 15:13:16 -0300	
Started at	2019-07-28 15:13:22 -0300	

Detalhes da tarefa em execução

Alguns desse detalhes são:

- Nome do *cluster* onde a tarefa está em execução;
- Identificação da instância EC2 onde a aplicação está em execução;
- Nome da definição da tarefa;
- Papel que a tarefa assumiu durante sua execução.

É possível clicar em cada um desses valores para poder obter informações adicionais sobre eles.

Na parte inferior dessa página é possível ver detalhes do *container* da aplicação, como pode ser observado na figura a seguir:

Containers

	Name	Container Id	Status	Image
▼	aws_project01	3f2fe990-30d4...	RUNNING	siecola/aws_project01:0.1.0

Details

Network bindings

Host Port	Container Port	Protocol	External Link
8080	8080	tcp	54.173.229.16:8080

Detalhes do container em execução

Repare que a imagem Docker que está sendo utilizada é a versão 0.1.0, que foi criada e publicada no DockerHub no capítulo anterior.

Também é possível notar o endereço externo para acesso à aplicação, que nesse caso é o 54.173.229.16:8080. Esse endereço pode ser utilizado para acessar o *controller* de teste através do Postman, da seguinte forma:

The screenshot shows a REST client interface. At the top, the URL `54.173.229.16:8080/api/test/dog/matilde` is entered into the address bar, which is highlighted with a red box. Below the address bar, the method is set to `GET`. Under the `Headers` tab, there are seven entries. The `Body` tab is selected, showing the response content: `{Name: matilde}`. Below the body, there are tabs for `Body`, `Cookies`, `Headers (3)`, and `Test Results`. The `Test Results` tab is currently inactive. At the bottom, there are buttons for `Pretty`, `Raw`, `Preview`, and `Auto`.

Acessando a aplicação em execução

Ainda nessa página, na aba Logs, é possível ver os logs de execução da aplicação `aws_project01`:

The screenshot shows the AWS CloudWatch Logs interface for the `aws_project01` application. The `Logs` tab is selected. At the top, there is a `Filter logs` input field and a time range selector with options `All`, `30s`, `5m`, `1h`, and `6h`. The table below has columns for `Timestamp (UTC+00:00)` and `Message`. The log entries are:

Timestamp (UTC+00:00)	Message
2019-07-28 15:16:56	2019-07-28 18:16:56.729 INFO 1 --- [nio-8080-exec-1] b.c.s.a.controller.TestController : Is matilde? true
2019-07-28 15:16:56	2019-07-28 18:16:56.722 INFO 1 --- [nio-8080-exec-1] b.c.s.a.controller.TestController : Test controller - name: matilde
2019-07-28 15:16:56	2019-07-28 18:16:56.426 INFO 1 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 307 ms
2019-07-28 15:16:56	2019-07-28 18:16:56.118 INFO 1 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'

Logs da aplicação em execução

Porém, no capítulo seguinte será mostrado uma forma melhor de observar os logs das aplicações executadas dessa maneira.



Antes de prosseguir, pare a execução da tarefa que foi iniciada nessa seção.

9.3 - Definição de serviços

A execução de tarefas pode ser interessante se um trabalho tiver que ser executado de forma manual, mas para aplicações que necessitam ficar constantemente em execução, por exemplo pra servir telas, ler mensagens de filas ou tratar eventos, é necessário um nível de controle e gerenciamento superior. Para isso existe a opção de criação de serviços dentro de um *cluster*, que são capazes de:

- Gerenciar o processo de execução de tarefas;
- Permitir que aplicações permaneçam em execução enquanto o serviço está em execução;
- Monitorar as tarefas, verificando seu status de execução;
- Reiniciar tarefas caso elas fiquem em estado inconsistente;
- Escalar horizontalmente a aplicação instanciando mais tarefas para atender a, por exemplo, uma demanda maior de processamento ou requisições.

Essencialmente uma definição de serviço possui:

- Qual *cluster* ele será executado;
- Qual a definição da tarefa a executar e monitorar;
- Quantas tarefas serão executadas.

Também é possível definir estratégias para executar as tarefas dentro do *cluster*, como por exemplo colocar execuções em zonas de disponibilidade diferentes. Da mesma forma é possível definir que tarefas iguais não sejam executadas em uma mesma instância EC2, ainda que dentro da mesma zona de disponibilidade.

Essa seção detalha o processo de criação de um serviço para ser executado dentro do *cluster-01*.

9.3.1 - Criando um application load balancer

Embora seja opcional, é interessante criar um *application load balancer* capaz de balancear as requisições de entrada entre as várias instâncias que podem ser executadas da aplicação *aws_project01*. Esse *load balancer* pode ser escolhido durante o processo de criação serviço, como será visto mais adiante.

Para a criação do *load balancer* a ser utilizado aqui, proceda da mesma forma como foi feito na seção 11 do capítulo 6, porém ressaltando alguns pontos importantes durante o processo de sua criação:

- Configure a porta 8080 como a de entrada;
- Selecione a VPC utilizada pelas instâncias EC2 do *cluster*;
- Selecione todas as zonas de disponibilidade em que as instâncias EC2 do *cluster* estão;
- Selecione o mesmo grupo de segurança utilizado pelas instâncias EC2 do *cluster*;
- Selecione todas as instâncias EC2 do *cluster*.



Todas essas informações podem ser obtidas no console que exibe as instâncias EC2. Basta clicar em qualquer instância utilizada pelo *cluster* e localizar as informações na aba de descrição.

Portanto, crie um *application load balancer* para ser utilizado pelo serviço a ser executado no `cluster-01`, antes de criar o serviço propriamente dito. Isso pode ser feito no console de instâncias EC2, na seção Load Balancing.

9.3.2 - Criando um serviço para executar uma tarefa

Agora que a definição da tarefa e o *application load balancer* já foram criados, é possível criar um serviço para gerenciar as tarefas em execução da aplicação. Para isso, volte ao console de gerenciamento do `cluster-01` e entre na aba Services:

Cluster : cluster-01

Get a detailed view of the resources on your cluster.

Status	ACTIVE
Registered container instances	2
Pending tasks count	0 Fargate, 0 EC2
Running tasks count	0 Fargate, 0 EC2
Active service count	0 Fargate, 0 EC2
Draining service count	0 Fargate, 0 EC2

Services **Tasks** **ECS Instances** **Metrics** **Logs**

Create **Update** **Delete** **Actions ▾**

Filter in this page **Launch type** ALI

Service Name

Serviços do cluster

Nessa tela, clique no botão Create para iniciar o processo de criação de um novo serviço no cluster-01.

A primeira seção de configuração do serviço solicita algumas informações como:

- Tipo das instâncias a serem utilizadas;
- Qual definição de tarefa a ser executada, além de sua revisão;
- Qual o *cluster* escolhido para a execução da tarefa;
- Nome do serviço;
- Número de tarefas desejado para execução.

Veja como deve ficar essa primeira seção de configuração do novo serviço:

Launch type FARGATE EC2 i

Task Definition Family i

Revision i

Cluster i

Service name i

Service type* REPLICA DAEMON i

Number of tasks i

[Criando o serviço no cluster](#)

Aqui o número de tarefas significa quantas instâncias da aplicação `aws_project01` serão executadas. A ideia é demonstrar o funcionamento do *application load balancer* balanceando as requisições entre essas duas instâncias.

Passe para a próxima página clicando no botão `Next step` no canto inferior direito da página.

Nessa página é necessário configurar o tipo do平衡器 e carga a ser utilizado pelo serviço. Nesse caso, selecione o tipo `Application Load Balancer` e além de escolher o que foi criado na seção anterior. Veja como deve ficar essa seção de configuração:

Load balancer type*	<input type="radio"/> None Your service will not use a load balancer.
	<input checked="" type="radio"/> Application Load Balancer Allows containers to use dynamic host port mapping (multiple tasks per instance). Multiple services can use the same listener port on a single host based routing and paths.
	<input type="radio"/> Network Load Balancer A Network Load Balancer functions at the fourth layer of the OpenShift model. After the load balancer receives a request, it selects a target using a default rule using a flow hash routing algorithm.
	<input type="radio"/> Classic Load Balancer Requires static host port mappings (only one task allowed per container). Routing and paths are not supported.

Service IAM role	<input type="button" value="Create new role"/>	
Load balancer name	cluster-01-lb	

Configurando o balanceador de carga

Repare aqui também será criado um novo papel com permissões para o serviço, caso nenhum ainda tenha sido criado.

Na seção `Container to load balance`, clique no botão `Add to load balancer` para realizar as configurações adicionais do balanceador de carga:

Container to load balance

Container name : port

Configurando o平衡ador de carga

Essa ação abrirá as configurações que devem ser feitas em relação ao *container* e o平衡ador de cargas que foi criado.

Felizmente tudo já foi criado durante o processo de criação do平衡ador de cargas, então é necessário apenas selecionar a porta:protocolo além do grupo de destino onde o平衡ador de carga irá trabalhar, como mostra a figura a seguir:

Container to load balance

aws_project01 : 8080

Production listener port*

Production listener protocol* HTTP

Target group name

Target group protocol HTTP 

Configurações adicionais do平衡ador de carga

As demais configurações dessa página se referem ao *container* a ser executado. Elas já foram carregadas da definição da tarefa, logo não é necessário alterar mais nada. Por isso, clique no botão *Next step*.

O próximo passo, que é opcional, se refere às configurações de **escalonamento horizontal automático** do serviço. Essa funcionalidade pode ser utilizada para fazer com que o serviço crie novas tarefas da aplicação automaticamente, baseado em parâmetros como consumo de CPU e memória, fazendo com que a aplicação possua mais instâncias e seja capaz de, por exemplo, tratar um número maior de requisições simultâneas.

Para habilitar essa funcionalidade no serviço que está sendo criado, selecione a opção `Configure Service Auto Scaling to adjust your service's desired count`. Isso fará com que o menu de configuração do *auto scaling* apareça.

A primeira sessão se refere às quantidades de tarefas que devem ser executadas. São elas:

- **Número mínimo de tarefas:** isso garante que haverão no mínimo a quantidade de tarefas definido por esse valor;
- **Número desejado de tarefas:** esse é o número desejado de tarefas desde o início da execução do serviço;
- **Número máximo de tarefas:** esse é o número máximo de tarefas que podem ser criadas durante o processo de escalonamento vertical das tarefas.

Obviamente essas configurações, assim como todas nessa seção, devem ser muito bem planejadas, pois tarefas de menos podem fazer com que o serviço fique escalando a todo instante e ao mesmo tempo, deve-se escolher um número máximo de tarefas que seja suportado pelo *cluster* em termos de quantidade de CPU e memória disponíveis.

Veja uma boa configuração, somente para efeitos didáticos, para a configuração do serviço que está sendo criado:

Minimum number of tasks

2

Automatic task scaling policies you set cannot reduce the minimum number of tasks below 2.

Desired number of tasks

2

Maximum number of tasks

4

Automatic task scaling policies you set cannot increase the maximum number of tasks above 4.

Número de tarefas a serem escalonadas

A próxima se refere às políticas de escalonamento do serviço, assim como os alarmes que farão com que novas instâncias sejam criadas ou destruídas.



Novamente, em uma aplicação real, é necessário fazer um estudo de como essas configurações devem ser feitas. Aqui elas estão sendo criadas apenas com efeito didático e demonstrativo.

A ideia é criar um alarme quando a CPU ultrapassar uma média de 85% durante 5 minuto. Quando esse alarme for disparado, o serviço deverá criar uma nova instância da tarefa, fazendo com que uma nova instância da aplicação seja executada. Para isso, selecione a opção Step scaling e configure a seção Scale out como mostra a figura a seguir:

Automatic task scaling policies

The screenshot shows the configuration of an automatic task scaling policy. The policy type is set to 'Step scaling'. The name is 'ScaleOutPolicy'. It is configured to execute when a new alarm is created ('Create new Alarm' is selected). The alarm name is 'highCPU', the metric is 'CPUUtilization', and the threshold is set to average 85% utilization for 1 consecutive period of 5 minutes. A 'Save' button is at the bottom.

Scaling policy type Target tracking Step scaling

Scale out (increase desired count) Delete X

Policy name* ScaleOutPolicy

Execute policy when Create new Alarm Use an existing Alarm

This wizard uses ECS metrics for new alarms. To scale your service with other metrics, create your alarms in the [CloudWatch console](#). and then refresh the alarm list here.

Alarm name highCPU

ECS service metric CPUUtilization

Alarm threshold Average of CPUUtilization >= 85
for 1 consecutive periods of 5 minu...

Save

Alarme de utilização de CPU

Para finalizar essa parte, clique no botão Save e passe para a seção Scale in. Essa seção é responsável por gerenciar quando uma tarefa deve ser destruída, quando ela estiver com a CPU abaixo de um valor médio, como mostra a figura a seguir:

Scale in (decrease desired count)

Policy name* ScaleInPolicy

Execute policy when Create new Alarm
 Use an existing Alarm

This wizard uses ECS metrics for new alarms. To scale your service with other metrics, create your alarms in the [CloudWatch console](#), and then refresh the alarm list here.

Alarm name lowCPU

ECS service metric CPUUtilization

Alarm threshold Average of CPUUtilization <= 5
for 1 consecutive periods of 5 minu...

Save

Alarme de utilização de CPU

Novamente, clique no botão Save para salvar esse alarme. E por fim, clique no botão Next step para ir para a próxima página.

Revise tudo o que foi criado e clique no botão Create Service para iniciar o processo de criação do serviço. Dentro de alguns segundos o serviço será criado. Ao final, clique no botão View Service localizado no canto inferior direito para visualizar o serviço em execução.

Na página de visualização do serviço, na aba Tasks, perceba que existem duas tarefas em execução, ou seja, existem duas instâncias da aplicação aws_project01 sendo executadas, isso porque a configuração do serviço exige que pelo menos duas tarefas fossem criadas desde o início.

Task status: **Running** Stopped

Filter in this page

Task	Task Definition
9e1a0b9e-167d-4553-b2d9-e52d7eb...	task-01:22
bc789431-ebba-4655-bcf6-bab1b91...	task-01:22

Tarefas do serviço em execução

Para acessar o *endpoint* de teste da aplicação `aws_project01` basta buscar o DNS do *application load balancer* que foi criado para esse serviço. Isso pode ser feito no console de instâncias EC2, na seção Load Balancers, como mostra a figura a seguir:

The screenshot shows the AWS EC2 Dashboard with the 'Load Balancers' section selected. The 'cluster-01-lb' load balancer is listed with its DNS name: 'cluster-01-lb-774295546.us-east-1.elb.amazonaws.com'. The 'DNS name' field is highlighted with a red box.

DNS do application load balancer

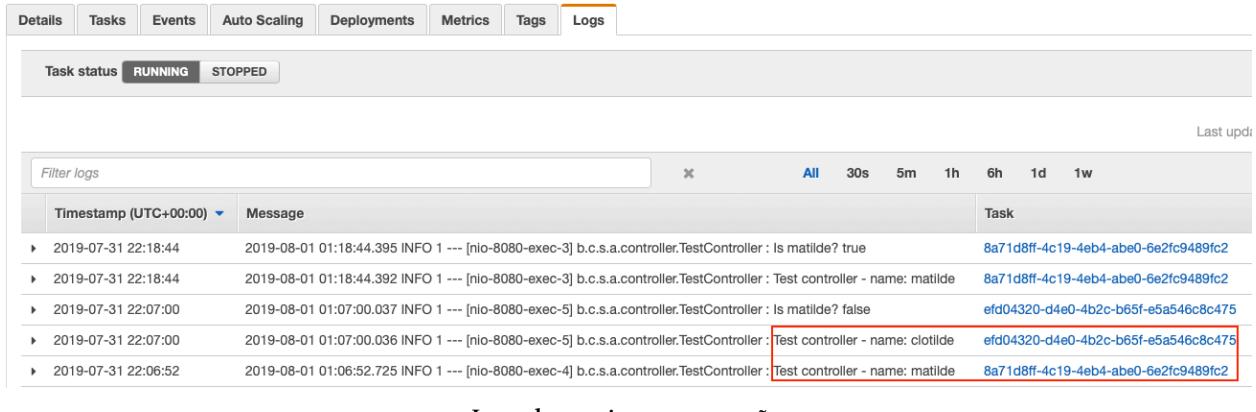
Com esse endereço, basta configurar o Postman para realizar, por exemplo, a seguinte requisição:

The screenshot shows the Postman application interface. A GET request is being made to the URL: 'cluster-01-lb-774295546.us-east-1.elb.amazonaws.com:8080/api/test/dog/matilde'. The 'Headers' tab is selected, showing '(7)' header entries. The 'Body' tab is also visible. The response body shows the result: '1 Name: matilde'.

Acessando o serviço pelo Postman

Isso fará com que a requisição seja tratada por uma das tarefas em execução no serviço. Faça outras

requisições, alterando o nome do último parâmetro e veja, na aba Logs do console do *cluster* que cada requisição é tratada por uma tarefa, de forma alternada, como mostra a figura a seguir:



The screenshot shows the AWS ECS service logs interface. At the top, there are tabs for Details, Tasks, Events, Auto Scaling, Deployments, Metrics, Tags, and Logs, with Logs selected. Below the tabs, a Task status filter shows RUNNING and STOPPED. A timestamp filter is set to UTC+00:00. The main area displays log entries with columns for Timestamp, Message, and Task. The log entries show alternating task IDs (8a71d8ff-4c19-4eb4-abe0-6e2fc9489fc2 and efd04320-d4e0-4b2c-b65f-e5a546c8c475) processing requests for 'matilde' and 'clotilde'. The last two entries for 'matilde' are highlighted with a red box.

Timestamp (UTC+00:00)	Message	Task
2019-07-31 22:18:44	2019-08-01 01:18:44.395 INFO 1 --- [nio-8080-exec-3] b.c.s.a.controller.TestController : Is matilde? true	8a71d8ff-4c19-4eb4-abe0-6e2fc9489fc2
2019-07-31 22:18:44	2019-08-01 01:18:44.392 INFO 1 --- [nio-8080-exec-3] b.c.s.a.controller.TestController : Test controller - name: matilde	8a71d8ff-4c19-4eb4-abe0-6e2fc9489fc2
2019-07-31 22:07:00	2019-08-01 01:07:00.037 INFO 1 --- [nio-8080-exec-5] b.c.s.a.controller.TestController : Is matilde? false	efd04320-d4e0-4b2c-b65f-e5a546c8c475
2019-07-31 22:07:00	2019-08-01 01:07:00.036 INFO 1 --- [nio-8080-exec-5] b.c.s.a.controller.TestController : Test controller - name: clotilde	efd04320-d4e0-4b2c-b65f-e5a546c8c475
2019-07-31 22:06:52	2019-08-01 01:06:52.725 INFO 1 --- [nio-8080-exec-4] b.c.s.a.controller.TestController : Test controller - name: matilde	8a71d8ff-4c19-4eb4-abe0-6e2fc9489fc2

Logs do serviço em execução



A aba Metrics, exibe gráficos de consumo de CPU e memória do serviço.

9.4 - Aumentando o número de instâncias do cluster

Eventualmente pode ser necessário aumentar o número de instâncias do *cluster*, sem prejudicar os serviços que estão em execução. Para isso, navegue até o `cluster-01`, na aba ECS Instances. Essa aba mostra todas as instâncias em execução no *cluster*.

Para alterar o número de instâncias EC2 do *cluster*, clique no botão Scale ECS Instances e configure para 4 instâncias no popup que aparecer.

Em alguns instantes as duas novas instâncias EC2 estarão disponíveis no *cluster*.

Essa mesma tela pode ser utilizada para diminuir a quantidade de instâncias em execução, porém isso fará que com que as tarefas que estiverem em execução nelas sejam finalizadas.

9.5 - Atualizando o serviço com uma nova definição de tarefa

O ECS possui uma funcionalidade que permite a atualização de um serviço para uma nova definição de tarefa, por exemplo para atualizar a imagem do Docker com uma nova versão da aplicação ou para alterar configurações de sua execução. A grande vantagem é que esse método permite que a aplicação em execução não seja interrompida durante a troca da definição da tarefa. O ECS também

garante que, caso algo de errado aconteça na instalação da nova definição da tarefa, a definição antiga continue funcionando, sem nenhuma interrupção na aplicação.

Para exemplificar esse conceito, crie um novo *endpoint* na aplicação `aws_project01`, dentro da classe `TestController`, como no trecho a seguir:

```
@GetMapping("/dogcolor/{name}")
public ResponseEntity<?> dogColor(@PathVariable String name) {
    log.info("Dog color - name: {}", name);

    return ResponseEntity.ok("Always black!");
}
```

Em seguida, altere a versão da aplicação para **0.2.0**, no arquivo `build.gradle`, como no trecho a seguir:

```
bootJar {
    baseName = 'aws_project01'
    version = '0.2.0'
}
```

Agora abra a aba `Gradle` do IntelliJ IDEA, localizado no canto superior direito, e execute a tarefa `dockerPush0.2.0`. Isso fará com que a nova versão da aplicação, com o novo *endpoint* seja publicado no DockerHub, podendo ser utilizado para a nova versão da tarefa do serviço no ECS.

Para atualizar a definição da tarefa, vá no console do ECS e clique na seção `Task Definitions`. Nessa tela, selecione a tarefa `task-01` e clique no botão `Create new revision`. Nessa página, vá até a seção `Container Definitions` para localizar onde o *container* a ser utilizado pela tarefa foi definido e clique nele, como mostra a figura a seguir:

Container Definitions

Add container	
Container Name	Image
 aws_project01	siecola/aws_project01:0.1.0

Definição do container da tarefa

Na página que abrir, altere a versão da imagem do Docker para **0.2.0**, como mostra a figura a seguir:

The screenshot shows a configuration form for a Lambda function. At the top, there is a field labeled "Container name*" containing the value "aws_project01". Below it, another field labeled "Image*" contains the value "siecola/aws_project01:0.2.0", which is highlighted with a red rectangular border. Both fields have a blue outline.

Atualizando a imagem da aplicação

Para prosseguir, clique no botão **Update**, localizando no canto inferior direito dessa página. Perceba que a versão da imagem Docker já foi atualizada na definição do *container*.

Para finalizar a nova definição da tarefa, clique no botão **Create**, no canto inferior direito dessa página. Isso criará uma nova revisão da tarefa, a partir da que já existia, tendo somente a imagem do Docker alterada para a nova versão, com o novo *endpoint*.

Para atualizar o serviço com a nova definição da tarefa, entre dentro do `cluster-01`, vá na aba `Services`, selecione o serviço `service-01` e clique no botão **Update**, como mostra a figura a seguir:

[Clusters](#) > cluster-01

Cluster : cluster-01

Get a detailed view of the resources on your cluster.

Status **ACTIVE**

Registered container instances 2

Pending tasks count 0 Fargate, 0 EC2

Running tasks count 0 Fargate, 2 EC2

Active service count 0 Fargate, 1 EC2

Draining service count 0 Fargate, 0 EC2

Services

Tasks

ECS Instances

Metrics

Create

Update

Delete

Actions ▾

Filter in this page

Launch type ALL



Service Name



service-01

Atualizando o serviço

Na tela que aparecer, na seção Task definition, altere a revisão da tarefa, no campo Revision, para a nova revisão da tarefa que foi criada.

Para prosseguir, clique no botão Skip to review e em seguida no botão Update Service. Isso fará com que o serviço seja atualizado com a nova versão da tarefa e consequentemente com a nova versão da imagem do Docker, com novo *endpoint* criado.

Após alguns instantes, o serviço já estará atualizado e será possível testar o novo *endpoint* com o Postman:

The screenshot shows the Postman interface with a GET request to the specified endpoint. The Headers section contains 7 items. The Body section is selected, showing options for 'none', 'form-data', 'x-www-form-urlencoded', 'raw', 'binary', and 'GraphQL BETA'. The response body displays the text 'Always black!'. The interface includes tabs for Body, Cookies, Headers (4), and Test Results, with the Body tab currently active.

Acessando o novo endpoint

Perceba o quanto é rápido e prático atualizar uma aplicação em execução no ECS. Veja também que a aplicação aws_project01 não ficou “fora do ar” para que sua nova versão fosse atualizada.



É importante lembrar que o código do projeto está no GitHub e pode ser acessado [aqui¹¹](#).

9.6 - Conclusão

Nesse capítulo foram demonstrados conceitos básicos da utilização do serviço ECS da Amazon, bem como suas particularidades como tarefas e serviços, incluindo a apresentação do mecanismo de alterar a versão de uma aplicação sem a interrupção do serviço por ela oferecido.

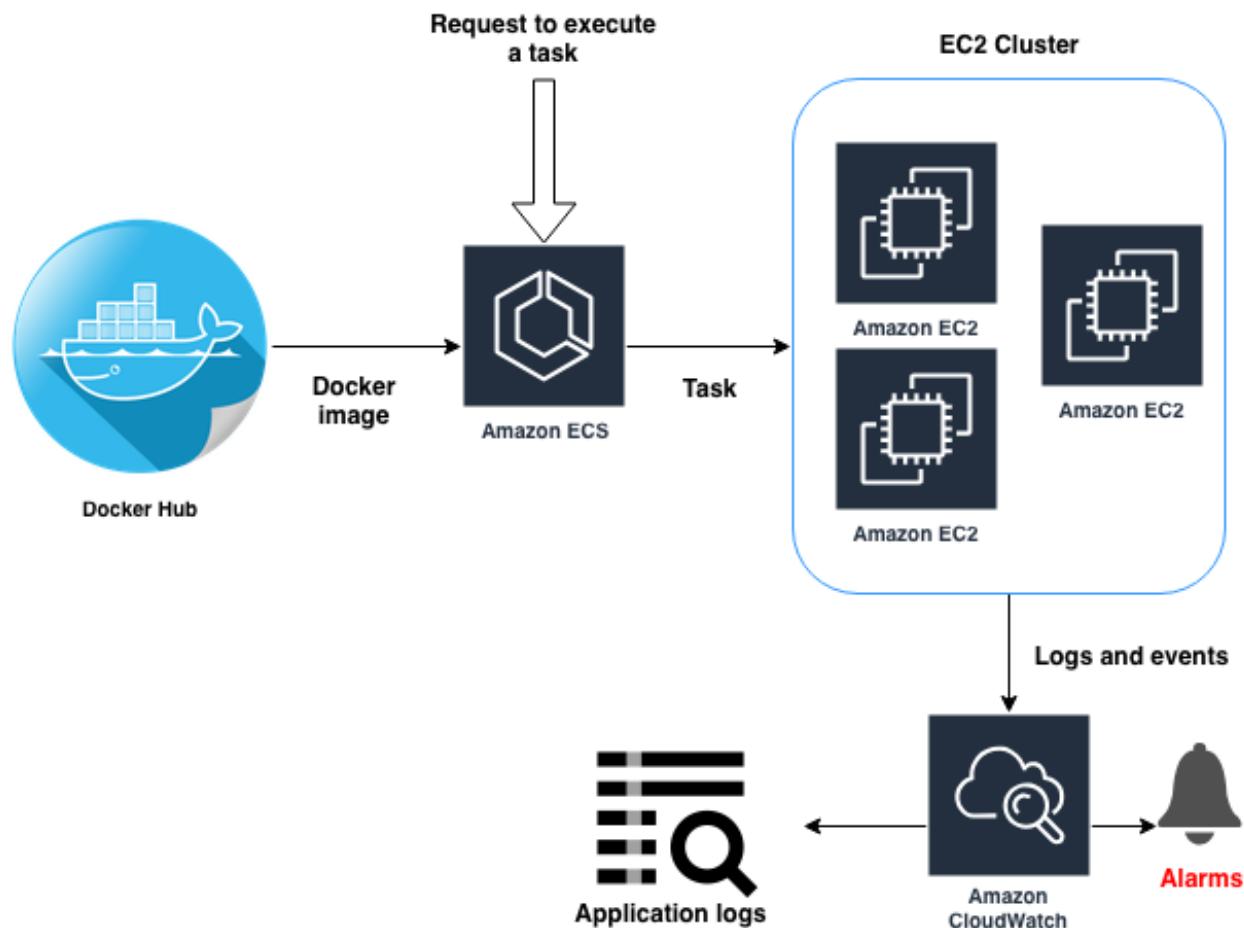
No próximo capítulo será mostrado outro serviço da AWS, o CloudWatch, um poderoso aliado para monitoramento, com visualização de logs, alarmes e eventos.

¹¹https://github.com/siecola/aws_project01

10 - Amazon CloudWatch

O CloudWatch é um serviço da AWS para monitoramento através de logs, métricas, eventos e alarmes, com o mesmo mecanismo onde o usuário é cobrado pelo seu uso, semelhante a outros serviços da AWS.

O intuito deste capítulo é fornecer uma visão básica do que esse serviço pode oferecer, usando a aplicação `aws_project01` como exemplo, sendo executada no *cluster* criado no capítulo anterior. A ideia é, além de visualizar seus logs, também observar suas métricas de execução e gerar eventos e alarmes, de forma a demonstrar como é possível utilizar o CloudWatch como uma importante ferramenta de monitoramento.



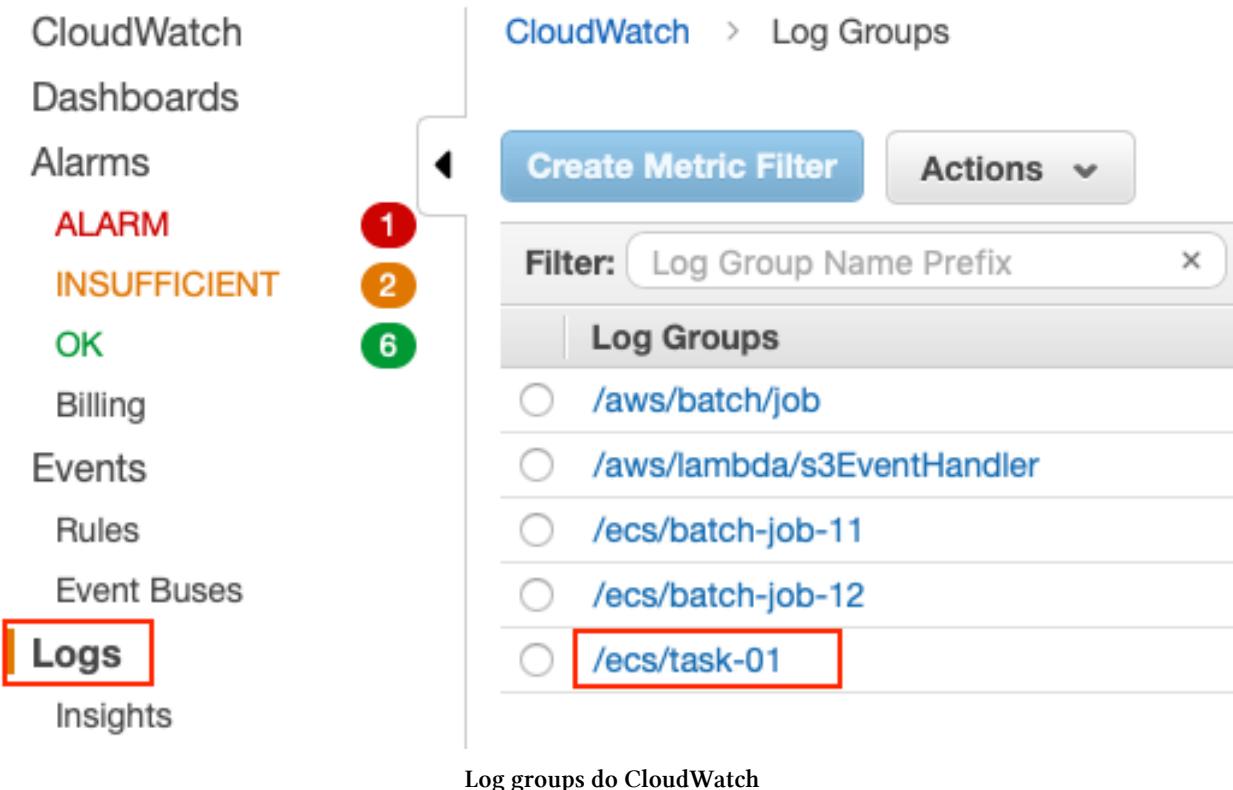
CloudWatch e a aplicação `aws_project01`

Para começar, acesse o console da AWS e vá até o menu Services. Em seguida localize o serviço CloudWatch, para abrir seu console de gerenciamento.

Na página principal do CloudWatch é exibida uma visão geral, com alguns gráficos e tabelas com informações sobre os serviços da conta. No menu esquerdo, existem as opções para de navegação.

10.1 - Visualizando logs da aplicação

O capítulo anterior mostrou como é possível observar os logs de execução da aplicação através da aba Logs do serviço criado no *cluster*. Porém, dentro do CloudWatch é possível observar, pesquisar e filtrar os logs concentrados da aplicação, que ficam divididos por grupos, como mostra a figura a seguir:



Repare que para acessar os grupos de logs, basta acessar o menu lateral, na opção Logs.

Os grupos de logs normalmente são divididos pelo tipo de serviço e em seguida tarefas e/ou aplicações. Na imagem anterior, por exemplo, existe o grupo de logs /ecs/task-01, que representa a task-01 criada na capítulo anterior, com a execução da aplicação aws_project01.

Nessa mesma tela, também é possível filtrar os grupos de logs, na opção Filter.

Clicando-se no grupo de logs /ecs/task-01, uma nova página é aberta com os logs desse grupo, divididos em streams. Aqui também é possível filtrar por um nome de stream específico.

CloudWatch > Log Groups > Streams for /ecs/task-01

The screenshot shows the AWS CloudWatch Log Groups interface. At the top, there are three buttons: "Search Log Group" (blue), "Create Log Stream" (grey), and "Delete Log Stream" (grey). Below these are two input fields: "Filter:" and "Log Stream Name Prefix". Underneath is a section titled "Log Streams" with a checkbox. A list of log stream names is provided, each with a corresponding checkbox:

- ecs/aws_project01/a4d68461-8ec9-4f1e-aac4-960530fe8d65
- ecs/aws_project01/d7cba9bc-c051-4954-b8e3-0b2c20fc83ea
- ecs/aws_project01/42efc8e6-30ef-4848-92a5-3abb62426822
- ecs/aws_project01/cff9072e-c16c-44ea-b79b-4e3b4a1cbf45
- ecs/aws_project01/ddb4f5f5-2cb8-49e8-8a7e-b7209604bedf
- ecs/aws_project01/617d6fb6-6e03-4704-b0ba-28c56b798076

Log streams

Dentro de cada *stream* estão os logs de execução da aplicação `aws_project01`, como pode ser visto na figura a seguir:

CloudWatch > Log Groups > /ecs/task-01 > ecs/aws_project01/efd04320-d4e0-4b2c-b65f-e5a546c8c475

The screenshot shows the AWS CloudWatch Log Events interface. At the top left is a "Filter events" input field. The main area has a table with two columns: "Time (UTC +00:00)" and "Message". The table shows log entries for August 1, 2019:

Time (UTC +00:00)	Message
2019-08-01	
▶ 01:06:20	2019-08-01 01:06:20.934 INFO 1 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
▶ 01:06:21	2019-08-01 01:06:21.034 INFO 1 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 100 ms
▼ 01:07:00	2019-08-01 01:07:00.036 INFO 1 --- [nio-8080-exec-5] b.c.s.a.controller.TestController : Test controller - name: clotilde
2019-08-01 01:07:00.036	INFO 1 --- [nio-8080-exec-5] b.c.s.a.controller.TestController : Test controller - name: clotilde
▼ 01:07:00	2019-08-01 01:07:00.037 INFO 1 --- [nio-8080-exec-5] b.c.s.a.controller.TestController : Is matilde? false
2019-08-01 01:07:00.037	INFO 1 --- [nio-8080-exec-5] b.c.s.a.controller.TestController : Is matilde? false

Logs com detalhes

Cada linha de log pode ser expandida para a exibição de seus detalhes. Repare que existe informações de data e hora de quando o log foi gerado, assim como seu nível.

Nessa tela também é possível filtrar os logs do *stream* selecionado, usando os controles localizados no canto superior direito dessa tela.

10.2 - Visualizando métricas de um serviço de um cluster

O CloudWatch também oferece a possibilidade de visualização de métricas de um serviço em execução em um *cluster*, exibindo dados como consumo de CPU e memória. Para isso, no console do CloudWatch, escolha a opção Metrics no menu lateral esquerdo.

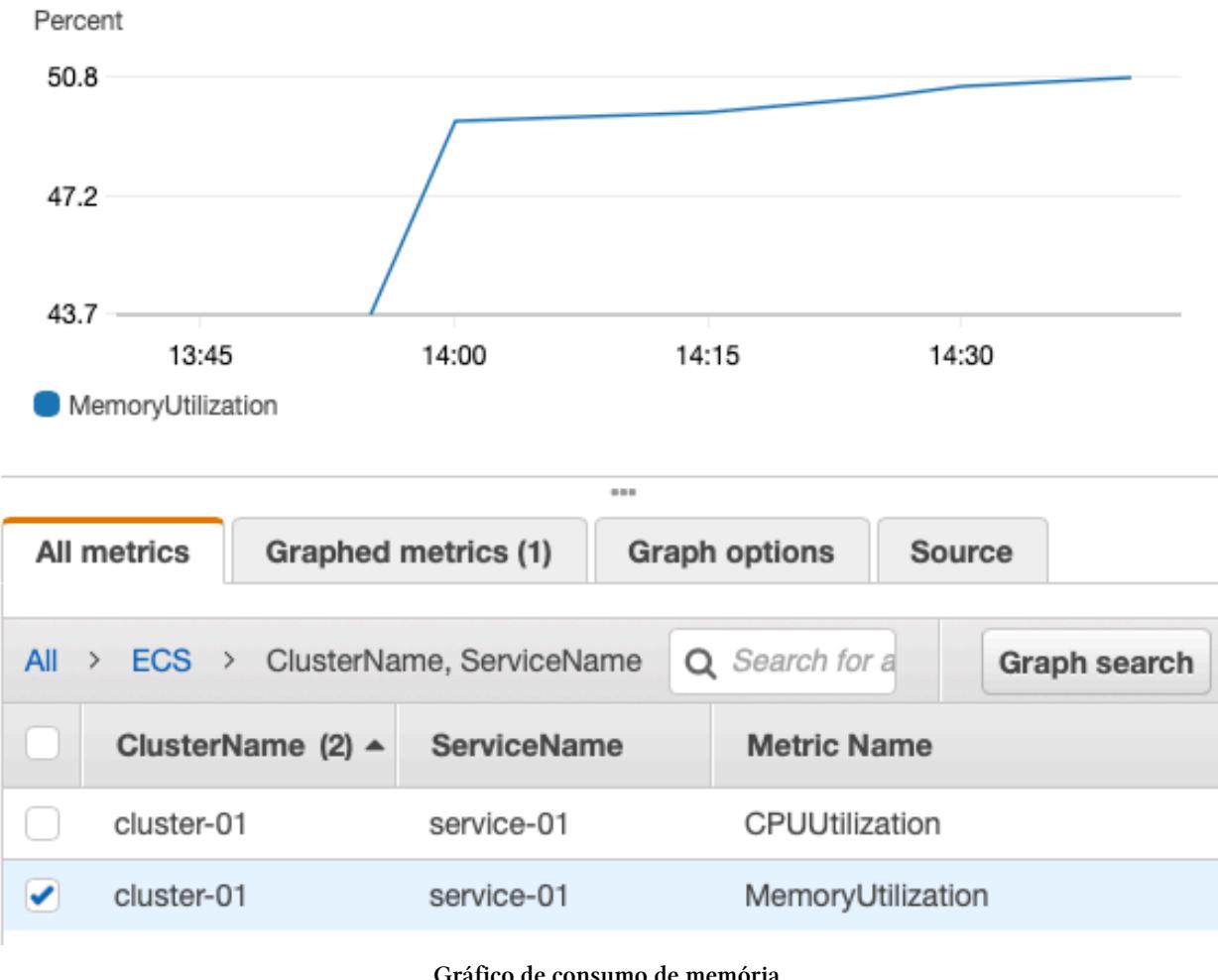
Nessa página existem várias métricas que podem ser visualizadas, como mostra a figura a seguir:

The screenshot shows the AWS CloudWatch Metrics interface. On the left, there's a sidebar with links: Insights, Metrics (which is selected and highlighted with a red box), Settings, Favorites, and a blue link to 'Add a dashboard'. The main content area has a header with tabs: All metrics (selected), Graphed metrics, Graph options, and Source. Below the tabs is a search bar with placeholder text 'Search for any metric, dimension or resource id'. The main content area displays '573 Metrics' and a list of namespaces under 'Custom Namespaces': LogMetrics (1 Metric), AWS Namespaces, ApplicationELB (118 Metrics), EC2 (213 Metrics), Logs (6 Metrics), Billing (11 Metrics), ECS (6 Metrics), and S3 (2 Metrics). The 'ECS' section is also highlighted with a red box. At the bottom, it says 'Métricas do CloudWatch'.

Clique na opção ECS para abrir esse grupo e em seguida clique no grupo `ClusterName`, `ServiceName`. Nessa tela serão exibidas duas métricas relacionadas ao serviço `service-01` do *cluster* `cluster-01`. São elas:

- **CPUUtilization:** esse gráfico mostra o uso de CPU dentro do *cluster*, da aplicação em execução no serviço;
- **MemoryUtilization:** esse gráfico mostra o uso de memória dentro do *cluster*, da aplicação em execução no serviço;

Quando qualquer uma das métricas é selecionada, o gráfico na parte superior da tela exibe os dados dentro de períodos que podem ser configurados de acordo com o desejado, como mostra a figura a seguir:



Também é possível selecionar mais de uma métrica para visualização no gráfico.

10.3 - Criando alarmes

Uma valiosa funcionalidade do CloudWatch é a criação de alarmes que podem ser baseados em vários eventos, como logs gerados pela aplicação informando algum evento importante ou mesmo um erro.

Os eventos podem ser agregados pelo CloudWatch dentro de um período de tempo e só assim gerar um alarme no console.

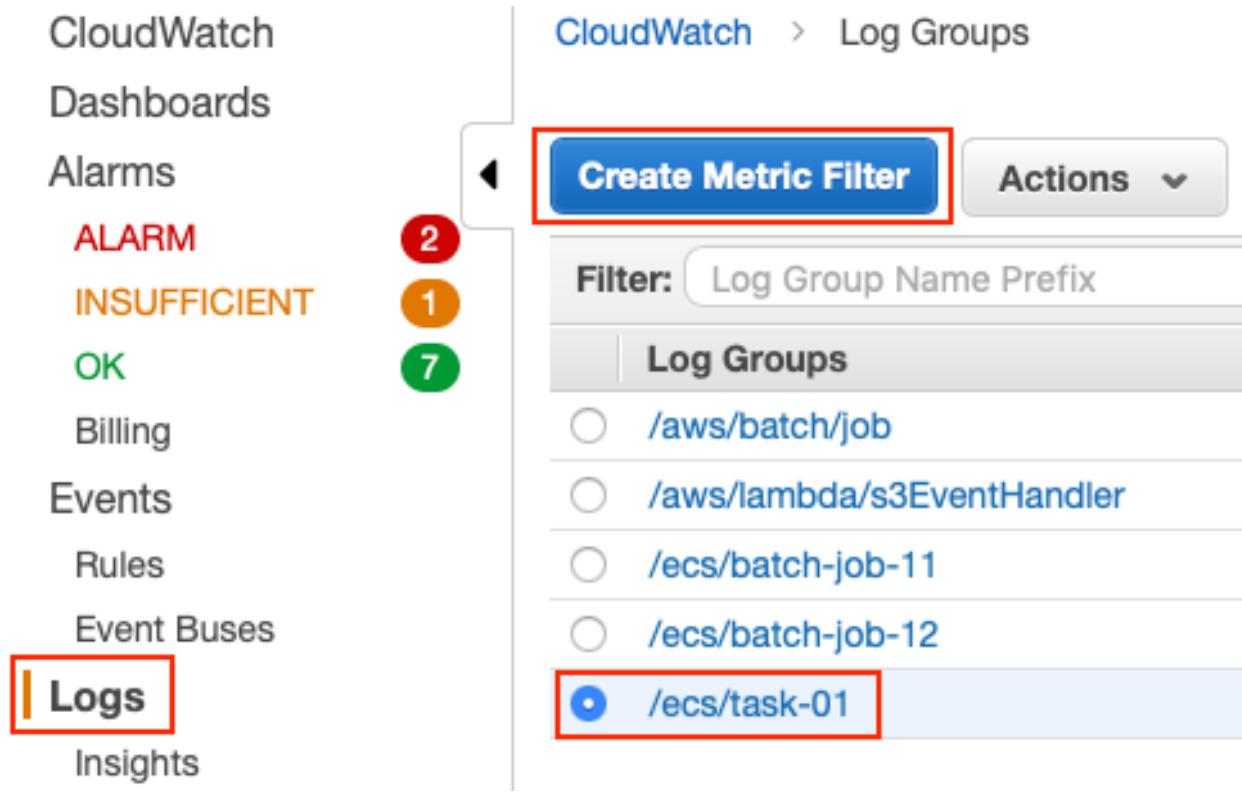
Os alarmes também podem gerar notificações que podem disparar ações em outra aplicação ou serviço da AWS.

A ideia dessa seção é criar um alarme baseado em um log específico da aplicação `aws_project01`. A seguir seus detalhes:

- Criar uma métrica que capture o log `Is matilde? true`;
- Se esse log aparecer mais de 5 vezes em um período de 1 minuto, um alarme deve ser gerado no CloudWatch;
- Quando esse alarme for gerado, um e-mail deve ser enviado para o administrador da conta.

Obviamente, o exemplo aqui é bem simples, mas é capaz de mostrar como o CloudWatch pode ser poderoso para geração de alarmes através de análise de logs, sem que a aplicação tenha que ser modificada para tal.

Para começar, vá na seção Logs do CloudWatch e selecione o grupo de logs `/ecs/task-01`, como mostra a figura a seguir:



Tendo o grupo de logs selecionado, clique no botão `Create Metric Filter` para abrir a tela para criação da métrica baseada em logs.

Na tela que aparecer, configure o campo `Filter Pattern` com o log desejado, como mostra a figura a seguir:

Também é possível testar, através do botão Test Patter, quais eventos foram localizados através do padrão configurado, como mostra a figura anterior.

Para passar ao próximo passo, clique no botão Assign Metric.

Na segunda tela de criação da métrica, apenas configure um nome desejado para a ela, no campo Metric Name, deixando os demais campos com os valores configurados pelo CloudWatch. Em seguida, clique no botão Create Filter.

Após a criação do filtro, repare que há uma opção para a criação de um alarme baseado nele, como mostra a figura a seguir:

Add Metric Filter

✓ Your filter **Is-matilde-true** has been created.

Filter Name: Is-matilde-true	Create Alarm
Filter Pattern: Is matilde? true	
Metric: LogMetrics / IsMatilde	
Metric Value: 1	
Default Value: none	
Alarm: IsMatilde	

Filtro criado

Nessa tela, clique então na opção **Create Alarm** para criar um alarme do CloudWatch baseado nesse filtro. A tela para especificar os detalhes e condições para geração do alarme será exibida. Nela é necessário configurar apenas dois campos para esse exemplo:

- **Período:** esse campo define o período em que o CloudWatch deve avaliar para a geração ou não do alarme;
- **Valor limite:** esse campo determina a quantidade de vezes que o log pode aparecer dentro do período escolhido.

Para realizar a configuração do período, configure o campo **Period** para 1 minuto, como mostra a figura a seguir:

Metric name

Statistic

Period

Período de avaliação do alarme

Da mesma forma, configure o último campo da seção Conditions com o valor 5, como mostra a figura a seguir:

Conditions

Threshold type

Static

Use a value as a threshold

Whenever IsMatilde is...

Define the alarm condition

Greater

> threshold

Greater/Equal

\geq threshold

than...

Define the threshold value

5



Configurando o valor limite

Essas duas configurações, juntamente com as demais com os valores padrões, fazem com que o CloudWatch monitore o log no padrão configurado em períodos de 1 minuto e caso ultrapasse 5 ocorrências, um alarme será gerado.

Para passar para o próximo passo, clique no botão Next dessa página.

Na seção de notificações, pode ser configurado uma ação quando o alarme for acionado, como por exemplo disparar uma mensagem para um endereço de e-mail. Para isso, configure o campo a seguir com o valor NotifyMe:

Send a notification to...



Email (endpoints)

Configurando a ação do alarme

Isso fará com que um e-mail seja enviado ao administrador da conta, quando o alarme for disparado. Passe para a próxima seção, clicando no botão Next. Nessa última página, configure o nome e a descrição para o alarme como desejar. Nesse exemplo, o nome será Matilde name was issued. Para finalizar, clique em Next, revise as configurações e clique em Create alarm para criar o alarme. Agora que o alarme foi criado, ele pode ser observado no CloudWatch, como mostra a figura a seguir:

	Name	State
<input type="checkbox"/>	lowCPU	⚠ In alarm
<input type="checkbox"/>	BillingAlarm	⚠ In alarm
<input type="checkbox"/>	Matilde name was issued	Insufficient data

Alarme criado

Perceba que o alarme que foi criado está no estado Insufficient data. Isso significa que, dentro do seu período de avaliação que foi configurado para 1 minuto, não existem dados para que ele entre no estado de alarme.

Para testar a geração desse alarme, basta fazer múltiplas requisições para o endpoint de teste da aplicação com o valor Matilde, como no exemplo a seguir:

`api/test/dog/matilde`

Fazendo mais do que 5 requisições como essa, dentro de um intervalo de 1 minuto, fará com que o alarme seja acionado, como mostra a figura a seguir:

The screenshot shows the AWS CloudWatch Alarms interface. On the left, a sidebar lists various services: CloudWatch, Dashboards, Alarms (with 3 ALARM notifications), INSUFFICIENT (0), OK (7), Billing, Events, Rules, Event Buses, Logs, and Insights. The 'Alarms' section is selected. The main area is titled 'CloudWatch > Alarms' and shows 'Alarms (10)'. A search bar is present. Below it is a table with columns 'Name' and 'State'. Three rows are highlighted with red boxes and labeled 'In alarm': 'Matilde name was issued', 'lowCPU', and 'BillingAlarm'. At the bottom, the text 'Alarm disparado' (Alarm triggered) is displayed.

Name	State
Matilde name was issued	In alarm
lowCPU	In alarm
BillingAlarm	In alarm

Alarm disparado

Depois que o alarme for disparado, um e-mail de notificação será enviado ao administrador da conta, de acordo com o que foi configurado durante a criação desse alarme.



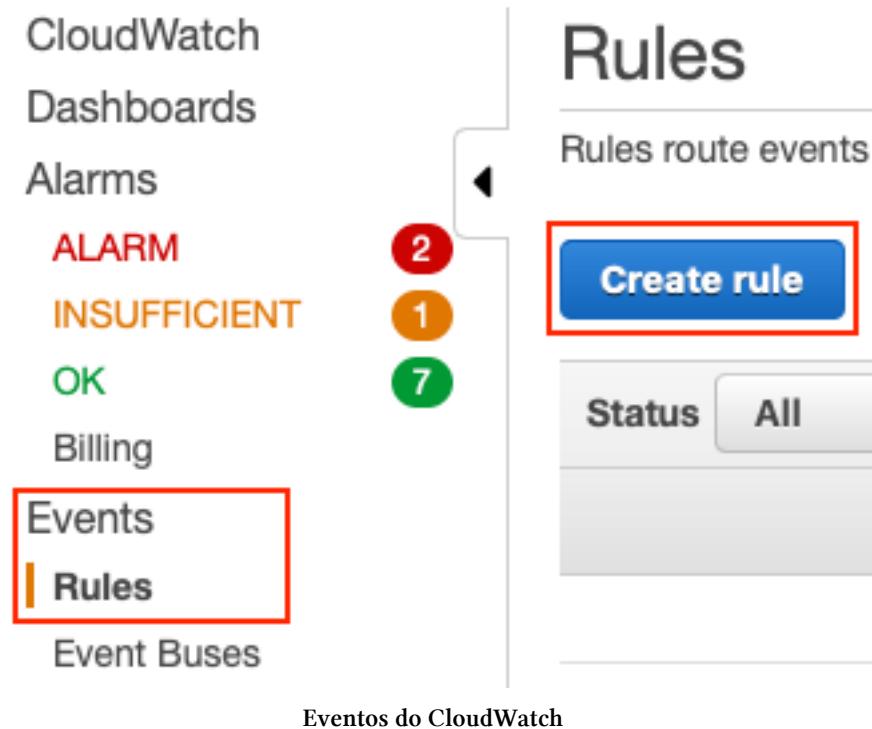
Embora a demonstração tenha sido bem simples, isso evidencia o poder do CloudWatch na configuração de alarmes baseados em logs, sem que a aplicação tenha que ser modificada para tal.

10.4 - Criando e monitorando eventos

Uma outra característica interessante do CloudWatch é a geração de logs baseados em eventos gerados pela infraestrutura, como por exemplo a mudança de estado de uma instância EC2.

Isso pode ser importante para o monitoramento de recursos na infraestrutura da AWS. Além disso, alarmes podem ser gerados a partir dessas notificações para que o administrador seja notificado de tais eventos.

Para demonstrar um exemplo simples, acesse o console do CloudWatch no menu Rules da seção Events, como mostra a figura a seguir:



Em seguida, clique no botão **Create rule** para começar o processo de criação de uma nova regra de eventos.

Como dito, a ideia é gerar um log no CloudWatch a partir de modificações de estados de instâncias EC2. Então para isso selecione o tipo EC2 no campo **Service Name**, assim como o valor **EC2 Instance State-change Notification** no campo **Event Type**, como mostra a figura a seguir:

Event Pattern Schedule

Build event pattern to match events by service

Service Name	EC2
Event Type	EC2 Instance State-change Notification

Configurando a fonte do evento

Agora que a fonte do evento foi configurada, é necessário criar a ação a ser invocada quando ele ocorrer. Isso pode ser feito nessa mesma página, no lado esquerdo, através do botão **Add target**.

Como nesse exemplo é apenas criar um log quando o estado de uma instância EC2 sofrer alguma modificação, escolha a opção **CloudWatch log group** e preencha o nome do grupo de log como na figura a seguir:

Targets

Select Target to invoke when an event matches your rule

CloudWatch log group

Log Group*

/aws/events/ event-01

Select a log group

Ação a ser invocada pelo evento

Isso fará com que seja criada uma mensagem de log nesse grupo, toda vez que o estado de uma instância EC2 for alterada.

Para continuar, clique no botão **Configure details** e atribua um nome que desejar para a definição da regra. Em seguida, clique em **Create rule** para finalizar.

Tudo certo! Agora que a regra foi criada, é possível testá-la. Para isso, vá no console de instâncias EC2 e altere o estado de qualquer instância. Isso fará com que um log seja gerado no CloudWatch, dentro do grupo de logs configurado, que nesse caso foi `aws/events/event-01`, como mostra a figura a seguir:

CloudWatch > Log Groups > /aws/events/event-01 > 019d6168

Filter events		
	Time (UTC +00:00)	Message
	2019-08-04	
▼	18:56:46	{"version":"0","id":"ee23d3fa-0b96-4b9e-931b-35ab410b7754", "detail-type": "EC2 Instance State-change Notification", "source": "aws.ec2", "account": "666336910744", "time": "2019-08-04T18:56:46Z", "region": "us-east-1", "resources": ["arn:aws:ec2:us-east-1:666336910744:instance/i-05ced0d39023bdfd1"], "detail": { "instance-id": "i-05ced0d39023bdfd1", "state": "running" } }

Visualizando os logs do evento

Perceba que na mensagem de log, vários detalhes são informados, incluindo a identificação da instância e seu novo estado, que nesse caso foi running.

Embora esse exemplo também seja bem simples, mas é possível perceber o quanto essa funcionalidade do CloudWatch pode ser interessante, principalmente para monitoramento de infraestrutura.

10.5 - Conclusão

Esse capítulo deu uma visão mais ampla ao CloudWatch, um serviço que muitas vezes é enxergado somente como um agregador de logs.

Com o CloudWatch é possível criar todo um mecanismo de monitoramento e alarmes, muitas vezes sem a necessidade de escrever ou alterar aplicações em execução, apenas configurando valores e parâmetros corretos de métricas e alarmes.

O próximo capítulo mostrará outro serviço da AWS, o Amazon Relational Database Service, utilizado para criação de instâncias gerenciadas de banco de dados.

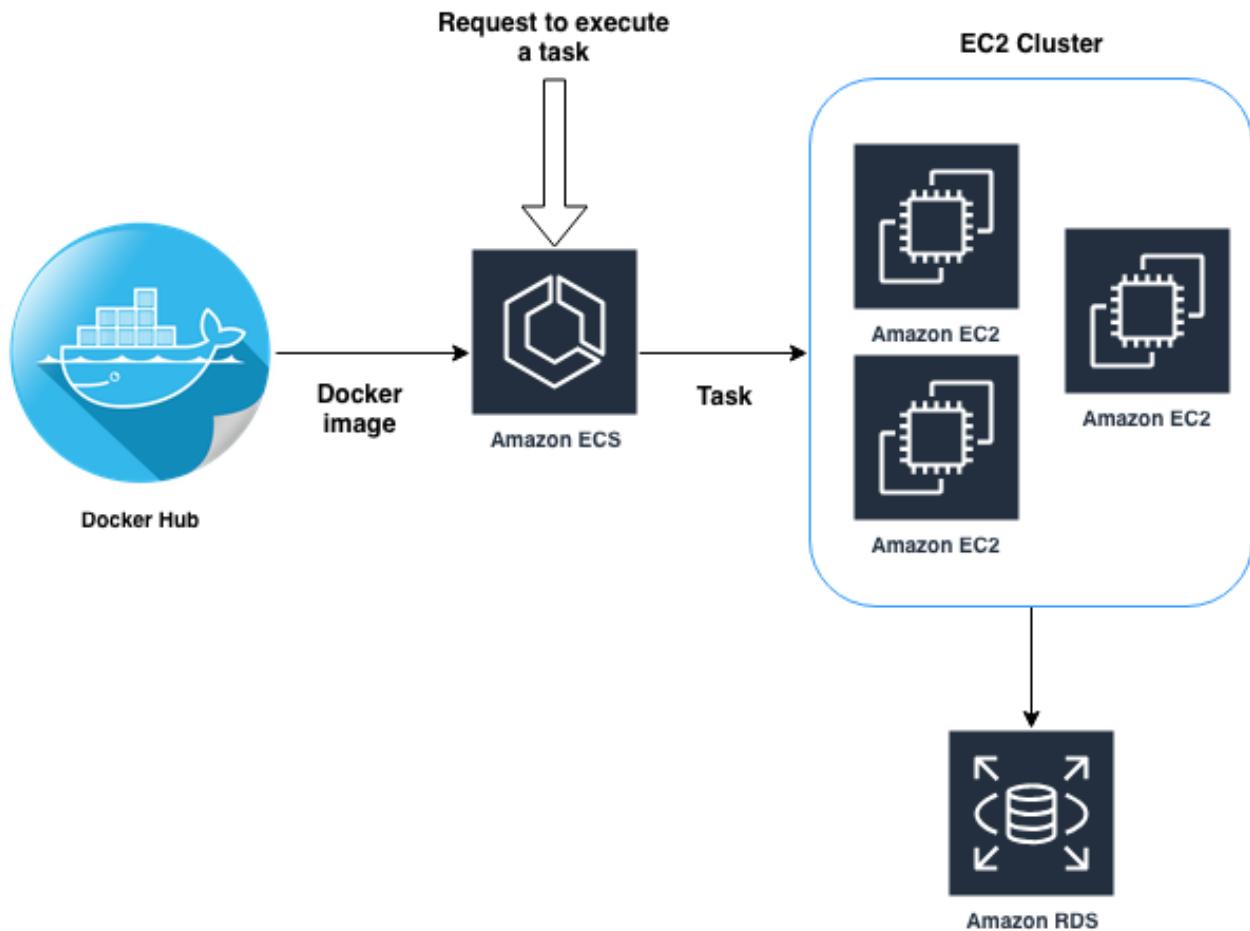
11 - Amazon Relational Database Service

Até o momento a aplicação `aws_project01` não persistia nenhum dado, ou seja, os *endpoints* que foram criados nesse projeto serviram apenas para testes, geração de logs e alarmes.

Este capítulo dá um passo adiante, exemplificando o uso de um banco de dados MySQL nessa aplicação, de tal forma que ela possa explorar um outro serviço da AWS, o **Amazon Relational Database Service** ou simplesmente **RDS** como será chamado aqui.

O RDS é um serviço da AWS que permite a criação de instâncias gerenciadas de banco de dados, sem a necessidade da configurações de servidores, manutenção e criação de regras de backup automáticas.

A ideia aqui é fazer com que a aplicação `aws_project01`, sendo executada de dentro de um serviço no *cluster* criado no capítulo anterior, possa acessar um banco de dados criado no RDS, como mostra a figura a seguir.



Acessando um banco de dados com o RDS

Novamente, o intuito é ser simples e sem complexidade de código, mas ao mesmo tempo mostrar conceitos importantes desse serviço que a AWS oferece.

Além disso, a aplicação `aws_project01` será incrementada com um novo *controller* para expor *endpoints* com operações básicas de persistência de uma nova entidade a ser criada.

As seções seguintes mostram como criar uma instância de um banco de dados MySQL no RDS, bem como configurar a aplicação para que possa acessá-la.

11.1 - Criando um banco de dados MySQL

Para começar a criar uma instância do banco de dados MySQL no RDS, acesse seu console na AWS e accesse a seção *Databases* no menu lateral esquerdo.

Essa é a página que lista todas as instâncias criadas até o momento. Para cria uma nova, clique no botão *Create database*, localizado no canto superior esquerdo.

A página de criação de um novo banco de dados possui vários parâmetros que devem ser

configurados. Para começar, na opção Engine options, selecione a opção MySQL. Isso fará que com o banco de dados a ser criado seja desse tipo.

A próxima configuração é o *template*. Nesse caso, escolha a opção Free tier. Essa opção é razoável para a demonstração que esse capítulo se propõe.

Na seção Settings deve-se configurar o nome da instância a ser criada, assim como as credencias de acesso, como mostra a figura a seguir:

The screenshot shows the 'Settings' section of the AWS RDS creation wizard. The 'DB instance identifier' field is highlighted and contains the value 'aws-project01-db'. A note below the field states: 'The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Can contain letters, numbers, underscores, and hyphens (1 to 15 for SQL Server). First character must be a letter. Can't contain two consecutive underscores or two consecutive hyphens.' The 'Master username' field is also visible, containing 'admin'.

DB instance identifier [Info](#)
Type a name for your DB instance. The name must be unique cross all DB instances owned by your Region.
aws-project01-db

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Can contain letters, numbers, underscores, and hyphens (1 to 15 for SQL Server). First character must be a letter. Can't contain two consecutive underscores or two consecutive hyphens.

▼ **Credentials Settings**

Master username [Info](#)
Type a login ID for the master user of your DB instance.
admin

1 to 16 alphanumeric characters. First character must be a letter
 Auto generate a password
Amazon RDS can generate a password for you, or you can specify your own password

Credenciais de acesso ao banco

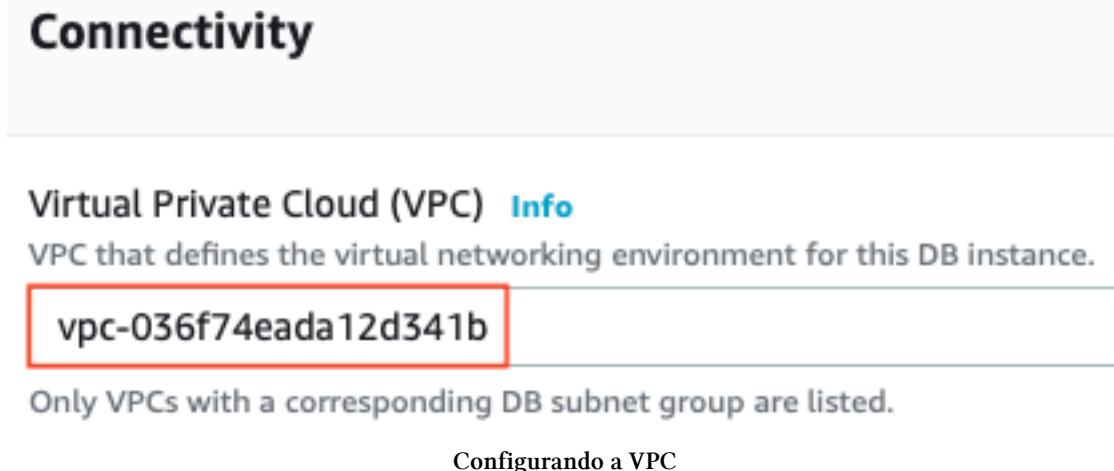
Deixe o nome da instância do banco configurada como aws-project01-db e defina o nome do usuário e sua senha. Lembre-se de guardar essas informações, pois elas serão necessárias para a configuração da aplicação aws_project01 para que ela possa acessar esse banco de dados.

A próxima seção, que a DB instance size oferece a opção de configuração da instância do banco, em termos de sua capacidade de processamento e memória. Instâncias com capacidades maiores são mais caras, mas apresentam um desempenho superior. Nessa configuração, escolha a opção db.t2.micro , suficiente para os testes desse capítulo.

A seção Storage define a capacidade de armazenamento da instância do banco de dados. Deixe essas opções nos valores padrões apresentados pelo console.

A seção Connectivity deve ser configurada com muito cuidado, pois ela define as regras de acesso à instância do banco de dados. Obviamente, tais regras devem permitir o acesso através das instâncias pertencentes ao cluster-01.

O primeiro parâmetro a ser configurado nessa seção é o no campo Virtual Private Cloud (VPC). Aqui selecione a mesma VPC que as instâncias do cluster-01 estão.



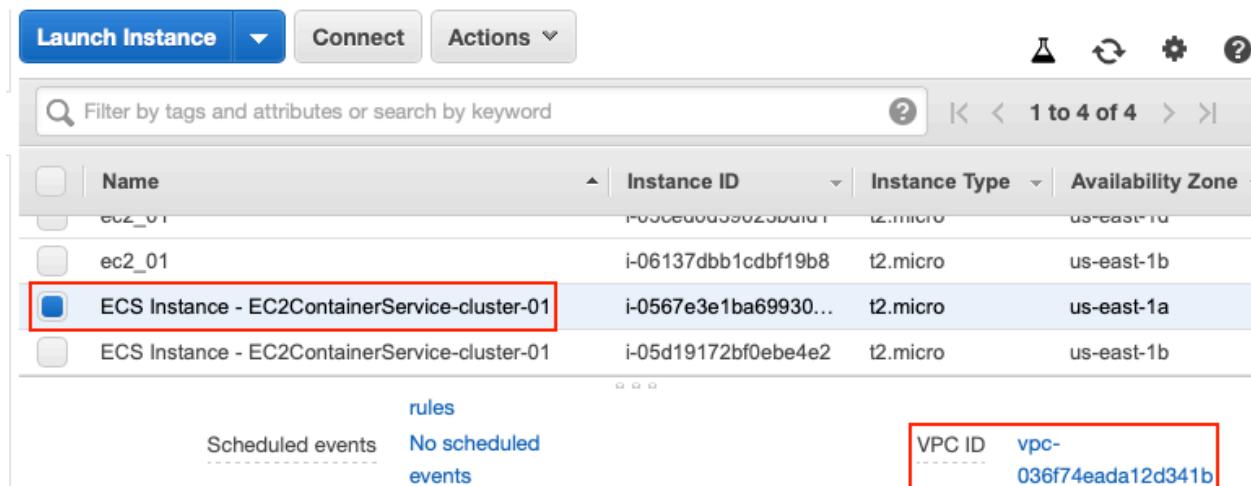
Virtual Private Cloud (VPC) [Info](#)
VPC that defines the virtual networking environment for this DB instance.

vpc-036f74eada12d341b

Only VPCs with a corresponding DB subnet group are listed.

Configurando a VPC

Isso pode ser verificado no console do EC2, clicando-se sobre uma dessas instâncias e verificando o valor do parâmetro VPC ID na aba Descriptions.



Name	Instance ID	Instance Type	Availability Zone
ec2_01	i-05ce000000000000	t2.micro	us-east-1a
ec2_01	i-06137dbb1cdbf19b8	t2.micro	us-east-1b
ECS Instance - EC2ContainerService-cluster-01	i-0567e3e1ba69930...	t2.micro	us-east-1a
ECS Instance - EC2ContainerService-cluster-01	i-05d19172bf0ebe4e2	t2.micro	us-east-1b

rules
Scheduled events No scheduled events
VPC ID vpc-036f74eada12d341b

Verificando a VPC correta

Ainda nessa seção, expanda a subseção de configurações adicionais e realize as seguintes configurações:

- Selecione a opção para a criação de um novo grupo de sub-rede;
- Escolha a opção para permitir acesso público à instância;
- Selecione a opção para utilizar um grupo de segurança da VPC já existente;

- Selecione o mesmo grupo de segurança da VPC que as instâncias do cluster-01 pertencem, que pode ser observado na mesma aba Descriptions exibida na figura anterior, no parâmetro Security groups.

Veja como essa parte da configuração deve ficar:

▼ Additional connectivity configuration

Subnet group [Info](#)

DB subnet group that defines which subnets and IP ranges the DB instance can use in the VPC you selected.

▼

Publicly accessible [Info](#)

Yes

Amazon EC2 instances and devices outside the VPC can connect to your database. Choose one or more VPC security groups that specify which EC2 instances and devices inside the VPC can connect to the database.

No

RDS will not assign a public IP address to the database. Only Amazon EC2 instances and devices inside the VPC can connect to your database.

VPC security group

Choose one or more RDS security groups to allow access to your database. Ensure that the security groups have the correct inbound rules to allow traffic from EC2 instances and devices outside your VPC. (Security groups are required for publicly accessible databases.)

Choose existing

Choose existing VPC security groups

Create new

Create new VPC security group

Existing VPC security groups

[Choose VPC security groups](#)▼

EC2ContainerService-cluster-01-EcsSecurityGroup-60X6SS76TQ99 X

default X

Configurações adicionais de conectividade

Ainda nessa seção é possível configurar a porta TCP de acesso à instância, que por padrão é a 3306. Altere essa configuração apenas se desejar mudar para uma porta diferente.

As configurações adicionais se referem backup, monitoramento, logs e manutenção, mas para esse teste simples, podem ser deixadas em suas opções padrões.

Para finalizar, clique no botão **Create database** no final da página. Após alguns minutos a instância estará pronta para ser utilizada.

Depois que a instância for criada, é necessário liberar a **porta 3306**, ou a que foi configurada no passo anterior, no grupo de segurança no qual as instâncias do **cluster-01** pertencem, como mostra o exemplo a seguir:

Security

VPC security groups

default (sg-045312c10e710c561)

(active)

EC2ContainerService-cluster-01-EcsSec

006ee2d49de6b4db6

(active)

Localizando o grupo de segurança

Clique no grupo de segurança correspondente e vá na aba **Inbound**. Edite suas regras para **adicionar** uma nova para permitir o acesso a essa porta por qualquer endereço IP, como mostra a figura a seguir:

Create Security Group Actions ▾

search : sg-006ee2d49de6b4db6

Group ID: sg-006ee2d49de6b4... Group Name: EC2ContainerService-cluster-0

Security Group: sg-006ee2d49de6b4db6

Description Inbound Outbound Tags

Edit

Type	Protocol	Port Range	Source
Custom TCP	TCP	8080	0.0.0.0/0
MySQL/Aurora	TCP	3306	0.0.0.0/0

Configurando o grupo de segurança

Dessa forma será possível acessar a instância do banco de dados de fora da AWS, para visualização dos dados através de um cliente MySQL.



É de extrema importância ressaltar que políticas de segurança devem ser muito bem planejadas de tal forma que **o banco de dados seja acessível somente pela aplicação**. Para efeitos didáticos, o banco criado aqui será exposto para toda a Internet, algo que **não deve ser feito em um ambiente real de produção**.

Agora que toda a configuração da instância foi feita, é possível acessá-la através de um cliente MySQL, como o MySQL Workbench utilizando o endereço e a porta TCP de acesso à instância, que

pode ser encontrado na aba **Connectivity & security** do console do RDS, como mostra a figura a seguir:

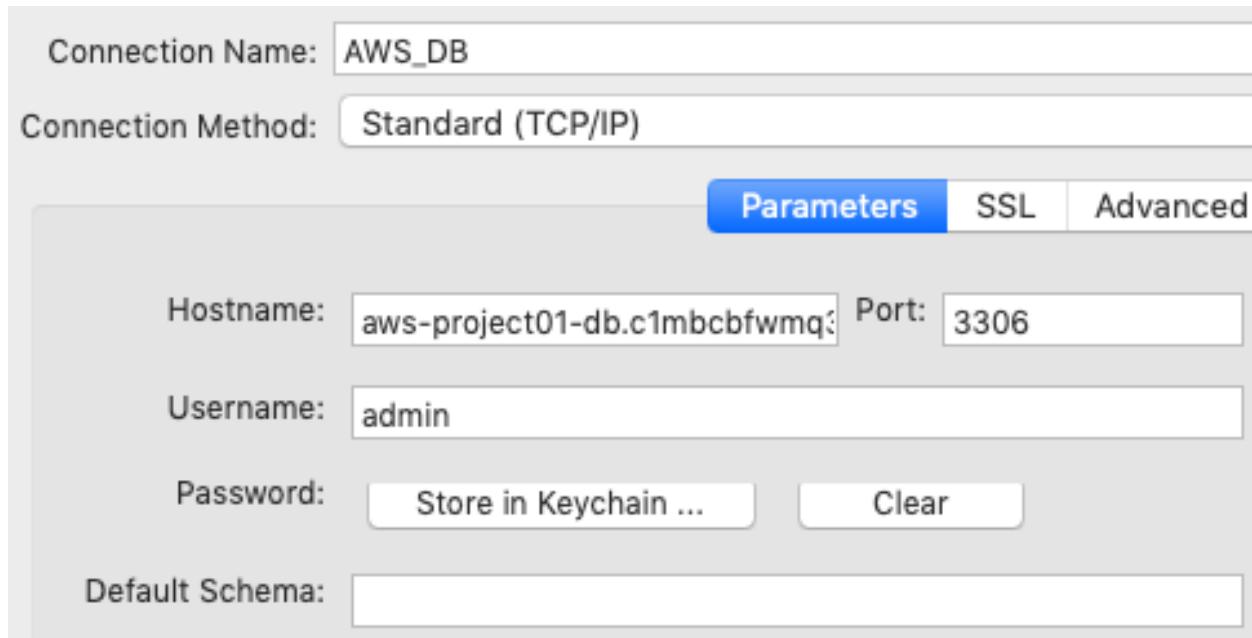
The screenshot shows the 'Connectivity & security' tab selected in the AWS RDS console. Below it, the 'Endpoint & port' section is displayed, containing the 'Endpoint' value 'aws-project01-db.c1mbcbfwmq3e.us-east-1.rds.amazonaws.com' and the 'Port' value '3306'. A note below states 'Endereço de acesso à instância'.

Endpoint	aws-project01-db.c1mbcbfwmq3e.us-east-1.rds.amazonaws.com
Port	3306

Endereço de acesso à instância

As credenciais de acesso, como usuário e senha, devem ser os mesmos utilizados no momento de criação da instância, como descritos nos passos anteriores.

Agora abra o MySQL Workbench, e acesse o menu Database -> Connect to Database. Nessa tela, dê um nome à nova conexão, além de preencher seu endereço de acesso e a porta TCP, como mostrado no exemplo da figura anterior. Veja como deve ficar na figura a seguir:



Configurando o MySQL Workbench

Lembre-se também de configurar o campo `username` com o nome do usuário configurado no momento de criação da instância. A seguir, clique no botão `Test Connection` dessa mesma tela para se conectar ao banco e concluir a configuração, digitando a senha de acesso à instância. Para finalizar, clique no botão `OK`.

Nesse momento a configuração de conexão ao banco já deve aparecer no MySQL Workbench. Para acessá-la, dê um duplo clique.

Agora é necessário criar um `schema`, que será utilizado pela aplicação `aws_project01` para a criação de sua tabela. No MySQL `schema` é uma espécie de agrupamento de tabelas. Para isso, dentro do MySQL Workbench, clique com o botão direito na seção `Schemas` e escolha a opção `Create Schema...`. Na tela que aparecer, digite o nome `aws_project01` e em seguida clique no botão `Apply` localizado no canto inferior direito.

Pronto! Tudo está preparado para que a aplicação `aws_project01` possa criar sua tabela e persistir seus dados utilizando a instância do banco de dados criada no AWS RDS.

11.2 - Configurando a aplicação para acessar o banco de dados MySQL

Configurar uma aplicação Spring Boot para acessar uma base de dados MySQL é muito simples. A seguir os passos necessários:

- Adicionar a dependência do Java Persistence API (JPA) no arquivo `build.gradle`;

- Adicionar a dependência do conector para um banco MySQL no arquivo `build.gradle`;
- Adicionar as configurações do Spring de acesso ao banco de dados no arquivo `application.properties`.



O foco dessa seção não é descrever em detalhes como uma aplicação Spring Boot funciona em conjunto com uma implementação do JPA. Para maiores detalhes, consulte literaturas dedicadas a esse assunto.

Para começar, abra o arquivo `build.gradle` no IntelliJ e adicione as duas dependências a seguir:

```
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
compile group: 'org.mariadb.jdbc', name: 'mariadb-java-client-jre7', version: '1.6.1'
```

A segunda dependência é uma implementação gratuita do conector Java para um banco de dados MySQL.

Em seguida, abra o arquivo `application.properties` e adicione as seguintes configurações nele:

```
spring.jpa.hibernate.ddl-auto=update  
spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBialect  
  
spring.datasource.url=jdbc:mariadb://localhost:3306/aws_project01  
spring.datasource.username=username  
spring.datasource.password=password
```

A linha se refere ao comportamento do Hibernate JPA quando a aplicação iniciar. A opção update instrui o Hibernate a atualizar o schema do banco de dados com novas alterações. Isso significa que quando a aplicação `aws_project01` for iniciada pela primeira vez todas as tabelas mapeadas em suas entidades serão criadas no banco de dados e eventuais mudanças em sua estrutura serão refletidas no schema do banco de dados.



Essa abordagem deve ser evitada em uma ambiente de produção, pois mudanças nas entidades da aplicação serão automaticamente aplicadas ao schema do banco de dados. O ideal é utilizar um mecanismo de migração de banco de dados como o Liquibase ou o Flyway.

A linha 2 configura o dialeto a ser utilizado pelo banco de dados.

As linhas 4, 5 e 6 configuram os dados de acesso ao banco de dados. Eles serão injetados pelo ECS no momento da execução do `container` da aplicação, como será detalhado mais adiante.

Somente com essas alterações a aplicação `aws_project01` já está configurada para acessar o banco de dados criado no RDS.

11.3 - Persistindo entidades no banco de dados

Essa seção discute o que deve ser feito na aplicação `aws_project01` para que ela possa persistir dados em uma tabela do banco de dados criado no AWS RDS. O exemplo que será descrito aqui é bem simples: uma tabela chamada `Product` será criada para o armazenamento de produtos de uma loja, com algumas regras e validações de persistência.

11.3.1 - A entidade de produtos

Como a aplicação `aws_project01` foi configurada para utilizar o Hibernate, que é um poderoso ORM (*Object-Relational Mapping*), é necessário apenas definir uma classe especial no projeto para que ela represente a tabela no banco de dados.

Para começar, crie um novo pacote na aplicação, chamado `model`. Dentro desse pacote, crie uma nova classe chamada `Product`, como mostra o trecho a seguir:

```
package br.com.siecola.aws_project01.model;

import javax.persistence.Entity;

@Entity
public class Product {
```

A anotação `@Entity` instrui o Hibernate que essa é uma classe que representa uma entidade no banco de dados, ou seja, uma tabela com o nome `Product`, que é o nome da classe.

Dentro dessa classe, é necessário definir seus atributos, que representarão as colunas da tabela `Product`. Veja como deve ficar no trecho a seguir:

```
@Entity
public class Product {

    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    private long id;

    private String name;

    private String model;

    private String code;
```

```
private float price;  
  
//getters and setters  
}
```

O atributo `id` possui duas anotações:

- **Id**: indica que o atributo é a chave primária na tabela do banco;
- **GeneratedValue**: define a estratégia de geração do valor da chave primária. Aqui a estratégica adotada é fazer com que o próprio banco de dados defina o valor para esse campo.

A classe `Product`, como está agora, já é suficiente para que o Hibernate possa criar a tabela de mesmo nome, porém ainda é possível adicionar outras características a ela, como:

- Definir que os atributos `name`, `model` e `code` sejam obrigatórios;
- Definir o tamanho dos atributos `name`, `model` e `code`;
- Garantir que o valor do atributo `code` seja único em toda a tabela.

Isso tudo pode ser feito com outras anotações, como no trecho a seguir:

```
@Table(  
    uniqueConstraints = {  
        @UniqueConstraint(columnNames = {"code"})  
    }  
)  
@Entity  
public class Product {
```

Dentro da anotação `Table` é possível definir colunas que devem ter valores únicos em toda tabela. Perceba que nesse caso a coluna de nome `code` possui esse comportamento.

Para definir os tamanhos dos campos `String` e também se são obrigatórios ou não, basta utilizar a anotação `@Column` como no trecho a seguir:

```

@Table(
    uniqueConstraints = {
        @UniqueConstraint(columnNames = {"code"})
    }
)
@Entity
public class Product {

    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    private long id;

    @Column(length = 32, nullable = false)
    private String name;

    @Column(length = 24, nullable = false)
    private String model;

    @Column(length = 8, nullable = false)
    private String code;

    private float price;

    //getters and setters
}

```

O atributo `length` da anotação `Column` define o tamanho do campo, assim como o atributo `nullable` define, nesse exemplo, que o campo é obrigatório.

A classe `Product`, como mostrada no trecho de código anterior, está na sua versão final.

11.3.2 - O repositório de produtos

Um repositório de dados é um padrão utilizado para abstrair as consultas ao banco de dados através de interfaces e métodos objetivos. Felizmente o Spring Data já possui uma implementação básica chamada `CrudRepository`. Através dela é possível definir uma interface própria, com outros métodos contextuais ao modelo no qual ela representa.

A ideia é criar um repositório para a tabela `Product`. Dessa forma será possível realizar consultas a ela sem precisar escrever muito código. Para começar, crie um novo pacote na aplicação `aws_project01`, chamado `repository`. Dentro dele, crie uma nova interface chamada `ProductRepository`, como no trecho a seguir:

```
package br.com.siecola.aws_project01.repository;

import br.com.siecola.aws_project01.model.Product;
import org.springframework.data.repository.CrudRepository;

public interface ProductRepository extends CrudRepository<Product, Long> {

}
```

Perceba que a interface extende de `CrudRepository`. Isso faz com que ela já possua alguns métodos de graça, como por exemplo:

- Localizar um produto pelo seu id;
- Localizar todos os produtos da tabela;
- Salvar ou alterar um novo produto;
- Apagar um produto.

Com esses métodos já é suficiente criar um *controller* com as operações básicas de CRUD, como será visto na próxima seção.

11.3.3 - O controller de produtos

Para expor as funções de CRUD do novo repositório de produtos e permitir que clientes externos possam acessá-las, é necessário criar um novo *controller* no pacote `controller`, com o nome `ProductController`, como no trecho a seguir:

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    private ProductRepository productRepository;

    @Autowired
    public ProductController(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }

}
```

Veja que no construtor da classe `ProductController`, uma instância do repositório de produtos já é injetada para que ela possa ser usada dentro dessa classe.



A ideia aqui é manter o código o mais simples possível, mas em aplicações reais é aconselhável criar uma camada de serviço entre o *controller* e o repositório, para possíveis lógicas de negócio.

É importante notar também que o caminho de acesso a esse *controller* é o definido na anotação `RequestMapping`, que nesse caso está configurado para `api/products`. É com esse endereço base que os *endpoints* de produtos poderão ser acessados.

O primeiro método dessa classe deve ser o que retorna todas os produtos do banco. Veja como dele deve ficar:

```
@GetMapping  
public Iterable<Product> findAll() {  
    return productRepository.findAll();  
}
```

Nesse método, a anotação `GetMapping` foi utilizada para definir o verbo **HTTP GET** como o de acesso à operação do serviço de produtos. Dessa forma, a URL de acesso a essa operação será `/api/products`.

Perceba que o repositório de dados já é utilizado, com um método que veio de `CrudRepository`.

A resposta dessa operação deve ser uma lista de produtos, como no exemplo a seguir:

```
[  
{  
    "id": 1,  
    "name": "Name1",  
    "model": "Model1",  
    "code": "1111",  
    "price": 10.0  
},  
{  
    "id": 2,  
    "name": "Name2",  
    "model": "Model2",  
    "code": "2222",  
    "price": 20.0  
}  
]
```

Para a busca de um produto pela sua identificação única, basta criar um outro método, como a seguir:

```

@GetMapping("/{id}")
public ResponseEntity<Product> findById(@PathVariable long id) {
    Optional<Product> optProduct = productRepository.findById(id);
    if (optProduct.isPresent()) {
        return new ResponseEntity<Product>(optProduct.get(), HttpStatus.OK);
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}

```

Novamente o repositório e produtos foi utilizado, com seu método `findById`, com o parâmetro recebido pela URL da requisição. Aqui a ideia foi utilizar o tipo `ResponseEntity` como resposta do método, que facilita na criação códigos de retorno e mensagens customizadas.

Na linha 5 desse método, é possível observar que o código de retorno foi definido como o HTTP 200 OK, passado como segundo parâmetro na criação do `ResponseEntity`.

Essa operação poderá ser acessada utilizando-se o verbo HTTP GET, através da URL `/api/products/{id do produto}`.

A resposta dessa operação deve ser um único produto, como o exemplo a seguir:

```
{
    "id": 1,
    "name": "Name1",
    "model": "Model1",
    "code": "1111",
    "price": 10.0
}
```

Caso o produto não seja encontrado, o código de retorno será o HTTP 404 Not Found.

Para criação de um novo produto, adicione o seguinte método à classe `ProductController`:

```

@PostMapping
public ResponseEntity<Product> saveProduct(
    @RequestBody @Valid Product product) {
    return new ResponseEntity<Product>(productRepository.save(product),
        HttpStatus.CREATED);
}

```

A anotação `Requestbody` indica que o parâmetro, que é o novo produto, deve vir dentro do corpo da requisição. Já a anotação `Valid` faz as validações necessárias desse corpo, de acordo com as configurações realizadas na classe `Product`, como por exemplo, exigir que o código seja preenchido.

Essa operação poderá ser acessada com o verbo HTTP POST através da URL `/api/products`, tendo o produto a ser criado em formato JSON em seu corpo, como no exemplo a seguir:

```
{
    "name": "Name1",
    "model": "Model1",
    "code": "1111",
    "price": 10.0
}
```

A resposta dessa operação deve ser o produto criado, já com o sua identificação única definida pelo banco de dados:

```
{
    "id": 1,
    "name": "Name1",
    "model": "Model1",
    "code": "1111",
    "price": 10.0
}
```

Para alterar um produto já existente, através de sua identificação única, basta criar o método conforme o trecho a seguir:

```
@PutMapping(path = "/{id}")
public ResponseEntity<Product> updateProduct(
    @RequestBody @Valid Product product, @PathVariable("id") long id) {
    if (productRepository.existsById(id)) {
        product.setId(id);
        return new ResponseEntity<Product>(productRepository.save(product),
            HttpStatus.OK);
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

Perceba que o verbo **HTTP PUT** é definido aqui através da anotação `PutMapping`, assim como sua URL de acesso, que nesse caso é `/api/products/{id do produto}`.

Como parâmetros, o método recebe, além do produto a ser alterado, sua identificação única para ser localizado no banco de dados, como é feito na linha 4, utilizando o método `existsbyId` do repositório de dados.

Caso o produto não seja encontrado, o código de retorno será o **HTTP 404 Not Found**.

E para apagar um produto, através do verbo **HTTP DELETE** e da URL `/api/products/{id do produto}`, basta criar o método como o exemplo a seguir:

```

@DeleteMapping(path = "/{id}")
public ResponseEntity<Product> deleteProduct(@PathVariable("id") long id) {
    Optional<Product> optProduct = productRepository.findById(id);
    if (optProduct.isPresent()) {
        Product product = optProduct.get();
        productRepository.delete(product);
        return new ResponseEntity<Product>(product, HttpStatus.OK);
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}

```

A resposta, caso o produto seja encontrado, será ele mesmo em formato JSON, com o código de retorno **HTTP 200 OK**. Caso o produto não seja encontrado, o código de retorno será o **HTTP 404 Not Found**.



Existem casos que devem ser tratados, como por exemplo exceções que são lançadas quando tenta-se criar um produto com um código já existente. Por simplicidade de código, esse e outros tratamentos foram omitidos.

Resumindo, as operações desse *controller* são:

Operação	URL	Verbo HTTP
Listar todos os produto	/api/products	HTTP GET
Buscar um produto pelo id	/api/products/{id do produto}	HTTP GET
Criar um produto	/api/products	HTTP POST
Alterar um produto pelo id	/api/products/{id do produto}	HTTP PUT
Apagar um produto pelo id	/api/products/{id do produto}	HTTP DELETE

E essas foram as operações básicas do *controller* de produtos.

11.4 Consultas avançadas

As operações básicas criadas na seção anterior podem satisfazer muitas necessidades de uma API, porém é interessante acrescentar operações com consultas avançadas, que serão discutidas nessa seção.

Felizmente, a aplicação `aws_project01` está utilizando o Spring Data, que além de várias funcionalidades já vistas nesse capítulo, permite a criação de métodos de busca em entidades do banco de dados, apenas escrevendo novos métodos no repositório de dados, seguindo um padrão que pode ser encontrado [aqui¹²](#).

¹²<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/>

Para demonstrar como fazer consultas avançadas utilizando o Spring Data, será criado um método para **consultar um produto pelo seu código**. Para isso, abra a interface `ProductRepository` e acrescente o seguinte método:

```
Optional<Product> findByCode(String code);
```

Repare que o nome do método deixa explícito que deve-se buscar um produto pelo valor do campo `code`. Como esse campo foi configurado para conter apenas valores únicos, o método então pode retornar um opcional de `Product`, ou seja, caso o produto seja encontrado, ele será retornado dentro `Optional`, caso contrário esse mesmo `Optional` será retornado vazio.

Agora na classe `ProductController` crie um novo método para expor uma nova operação ao serviço de produtos, que é a busca pelo código, conforme o trecho a seguir:

```
@GetMapping(path = "/bycode")
public ResponseEntity<Product> findByCode(@RequestParam String code) {
    Optional<Product> optProduct = productRepository.findByCode(code);
    if (optProduct.isPresent()) {
        return new ResponseEntity<Product>(optProduct.get(), HttpStatus.OK);
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

O endereço dessa nova operação será `api/products/code?code=<código do produto>` e poderá ser acessada pelo verbo **HTTP GET**.

Repare que o atributo `code`, que é passado como parâmetro de pesquisa na URL, é injetado no método através da anotação `@RequestParam`. Esse valor é então utilizado na chamada do novo método `findByCode` criado no repositório de produtos.



Perceba que nenhum código foi escrito para que essa consulta passasse a existir na aplicação. Ela foi criada apenas utilizando a notação que o Spring Data disponibiliza para consultas através de campos das entidades.

11.5 - Criando uma nova versão da aplicação

Nesse ponto, a aplicação `aws_project01` já está preparada para ser publicada novamente no DockerHub, mas primeiro, altere sua versão no arquivo `build.gradle` para **0.3.0**, conforme o trecho à seguir:

```
bootJar {  
    baseName = 'aws_project01'  
    version = '0.3.0'  
}
```

Em seguida, execute a tarefa dockerPush0.3.0 na aba Gradle, para que essa nova versão possa ser publicada no DockerHub, como foi feito no capítulo 9.

11.5 - Configurando o container da aplicação no ECS

Agora que a nova versão da aplicação aws_project01 foi publicada no DockerHub, é necessário atualizar a definição da tarefa que está sendo executada pelo service-01 que está em execução no cluster-01, criados no capítulo 9. Porém será necessário passar alguns parâmetros adicionais para a aplicação, para que ela possa acessar o banco de dados.

Para começar, vá até o console do AWS ECS e clique na opção Task Definitions no menu lateral esquerdo. A definição da tarefa task-01 deverá parecer. Clique nela e então inicie o processo para a criação de uma nova revisão a partir dela, através do botão Create new revision.

Como a intenção aqui é apenas alterar definições do *container* da aplicação, vá até essa seção e clique no nome do *container*, como mostra a figura a seguir:



Abrindo a definição do container

Na janela que abrir, altere o valor do campo Image para a nova versão da imagem da aplicação que foi gerada na seção anterior, ou seja 0.3.0.

A próxima alteração a ser feita é passar os parâmetros de acesso ao banco de dados. Eles já foram colocados no arquivo application.properties do projeto:

```
spring.datasource.url=jdbc:mariadb://localhost:3306/aws_project01
spring.datasource.username=root
spring.datasource.password=root
```

Mas agora eles terão que receber os valores corretos da instância do banco que foi criado no RDS. Para isso, vá até a seção **Environment variables** e acrescente três novos valores, como mostrado na figura a seguir:

Environment variables

You may also designate AWS Systems Manager Parameter Store keys or ARNs using the 'valueFrom' field. ECS

Key	Value	Add value
SPRING_DATASOURCE_URL	Value	Add value
SPRING_DATASOURCE_USERNAME	Value	Add value
SPRING_DATASOURCE_PASSWORD	Value	Add value

Configurando variáveis de ambiente

O campo **Key** é a chave que será passada para a aplicação e a terceira coluna é o seu valor. Eles são:

- **SPRING_DATASOURCE_URL**: a URL de acesso à instância do banco de dados. Seu valor deve ser composto pelo prefixo `jdbc:mariadb://`, mais a mesma URL que foi utilizada para a configuração do MySQL Workbench anteriormente nesse capítulo, a porta e o nome do schema. Esse valor deverá ficar como no exemplo a seguir:

```
jdbc:mariadb://aws-project01-db.c1wmq3e.us-east-1.rds.amazonaws.com:3306/aws_project\01
```

- **SPRING_DATASOURCE_USERNAME**: o usuário de acesso criado para dar acesso à instância do banco de dados criado no RDS;
- **SPRING_DATASOURCE_PASSWORD**: a senha de acesso criada para dar acesso à instância do banco de dados criado no RDS;

Esses valores serão passados para a aplicação e substituirão os valores de mesmo nome no arquivo `application.properties`, fazendo com ela acesse o banco de dados criado no RDS.

Tendo finalizado as configurações, clique no botão **Update** para concluir as alterações na imagem do *container*.

Por fim, clique no botão **Create** para criar a nova revisão da definição da tarefa.

De volta ao console do AWS ECS, clique no menu **Cluster** e selecione o `cluster-01`, para alterar o serviço em execução utilizar a nova definição da tarefa que foi criada. Para isso, selecione o

service-01 e clique em Update. Na tela que abrir, apenas selecione a última revisão da tarefa no campo Revision.

Como essa é a única alteração necessária no serviço, clique no botão Next até chegar no último passo e então clique no botão Update Service para atualizar o serviço.

O serviço service-01 deverá iniciar a nova definição da tarefa, com a nova imagem da aplicação, que agora acessa o banco de dados que foi criado no RDS. A conclusão do processo pode ser observada na aba de eventos do serviço, no console do AWS ECS.

Como a aplicação aws_project01 foi configurada para criar ou atualizar o *schema* do banco dados, através da propriedade `spring.jpa.hibernate.ddl-auto=update` do arquivo `application.properties`, uma nova tabela deverá ser criada no banco de dados do RDS. Para observá-la, abra o MySQL Workbench e acesse o *schema* `aws_project01`, como mostra a figura a seguir:

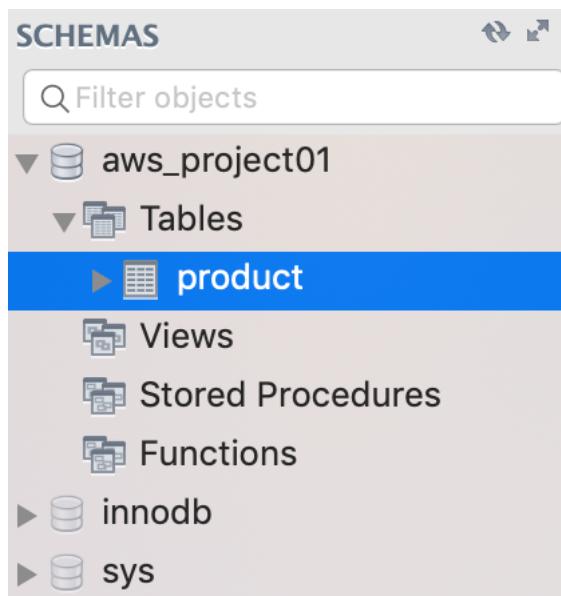


Tabela criada no schema

Abra a tabela e crie alguns produtos, em seguida acesse as operações do novo serviço de gerenciamento de produtos com o Postman, utilizando o endereço do *application load balancer* criado no capítulo 9, com o sufixo `:8080/api/products`, como no exemplo a seguir:

`cluster-01-1b-1184386073.us-east-1.elb.amazonaws.com:8080/api/products`

Utilize as instruções com os formatos de *payload* e verbos HTTP, conforme descrito na seção 11.3 desse capítulo, para acessar todas as operações desse novo serviço com o Postman.

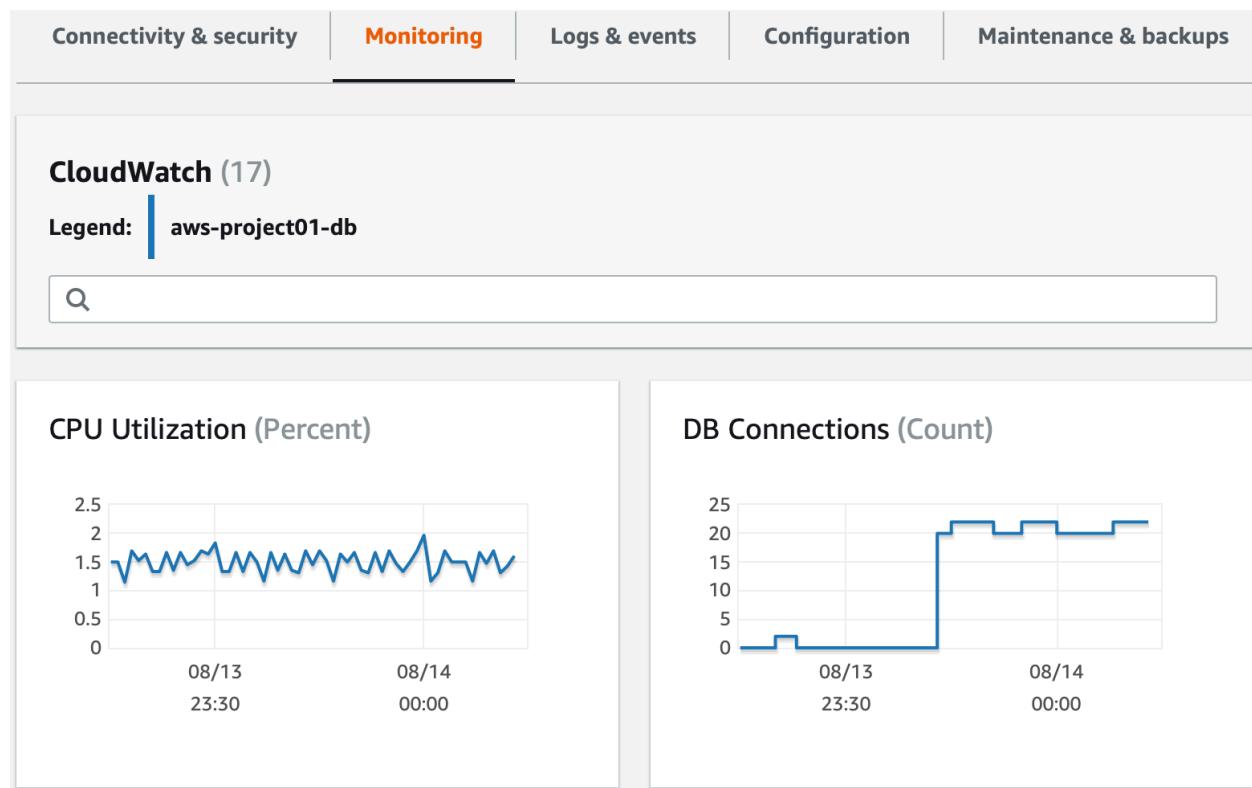
Pronto! A aplicação `aws_project01` agora acessa um banco de dados em uma instância MySQL criada no AWS RDS!

11.6 - Monitorando a instância do banco de dados

O console da AWS oferece opções interessantes para o monitoramento da instância do banco de dados, como:

- Utilização de CPU;
- Número de conexões com o banco de dados;
- Espaço disponível para armazenamento de dados;
- Utilização de memória;
- Operações de escrita e leitura.

Essas opções podem ser visualizadas através de gráficos na aba **Monitoring** do console do AWS RDS, como mostra a figura a seguir:



Monitorando a instância do banco de dados

Uma curiosidade com relação ao número de conexões no gráfico da figura anterior é que cada instância da aplicação abre, inicialmente, 10 conexões. Como haviam duas instâncias da aplicação em execução, mais o MySQL Workbench conectado ao banco, esse número chegou a 21 conexões.

Clicando-se em cada gráfico, é possível ter uma visão expandida, além de filtrar por intervalos de tempo customizados.



É importante lembrar que o código do projeto está no GitHub e pode ser acessado [aqui¹³](#).

11.7 - Conclusão

Embora o exemplo tenha sido bem simples nesse capítulo, ou seja, apenas criar um novo serviço de gerenciamento de produtos com uma tabela, foi possível demonstrar passos importantes e essenciais para a criação e configuração da instância no RDS, além de configurar a imagem da aplicação para acessá-la.

O próximo capítulo discutirá como o serviço de notificações da AWS pode ser utilizado permitir que a aplicação `aws_project01` possa enviar mensagens sobre criação e alteração de produtos, de forma a permitir que outras aplicações possam ficar sabendo de tais eventos, sem um acoplamento forte entre elas.

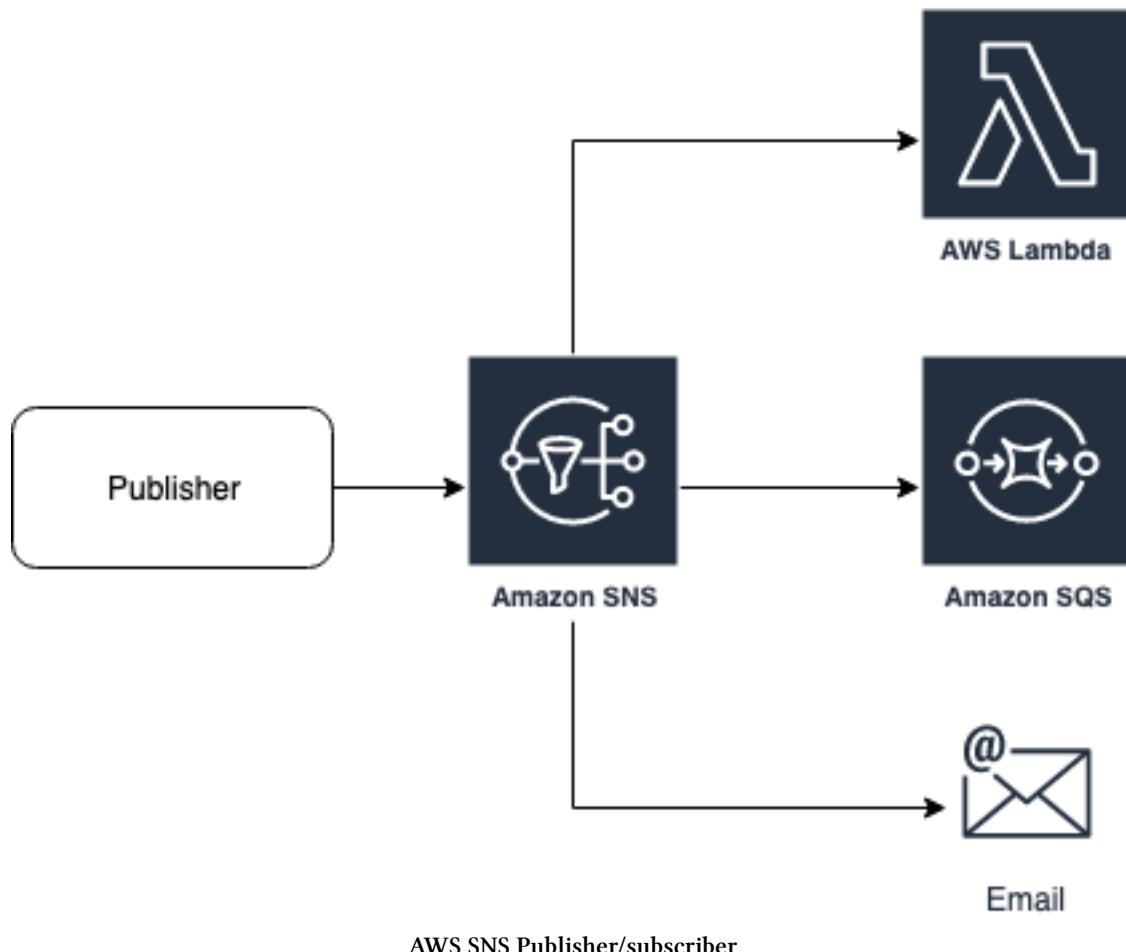
¹³https://github.com/siecola/aws_project01

12 - Amazon Simple Notification Service

O Simple Notification Service, ou SNS, é um serviço da AWS que permite que aplicações enviem notificações a outras aplicações de forma simples e descomplicada, sem a necessidade de muitas configurações e administração.

Com o AWS é possível reduzir o acoplamento entre microsserviços do ponto de vista do tratamento de eventos gerados por uma aplicação e consumidos por outras. Com ele também é possível **notificar vários sistemas de forma paralela**, sem a necessidade que o gerador dos eventos tenha que se preocupar com a distribuição das mensagens.

O SNS trabalha com um mecanismo de *publisher/subscriber*, onde quem gera as mensagens não precisa se preocupar com questões de entrega, bastando apenas publicá-las em um tópico do SNS.



Por sua vez, uma ou várias aplicações que se interessarem por tais eventos devem se **inscrever nos tópicos** de interesse para poderem receber uma cópia da mensagem publicada, através de *endpoints* HTTP, filas com o Simple Queue Service da AWS (que será visto no capítulo 13) ou AWS Lambda (que será visto no capítulo 16).

Também é possível gerar mensagens para serem entregues a dispositivos móveis, através de notificações *push* ou SMS além de entregar mensagens via e-mail.

O propósito deste capítulo é fazer com que a aplicação `aws_project01` gere eventos toda vez que um produto for criado, alterado ou apagado, com o intuito de, em capítulos à frente, criar uma aplicação capaz de gerar um histórico desses e de outros eventos.

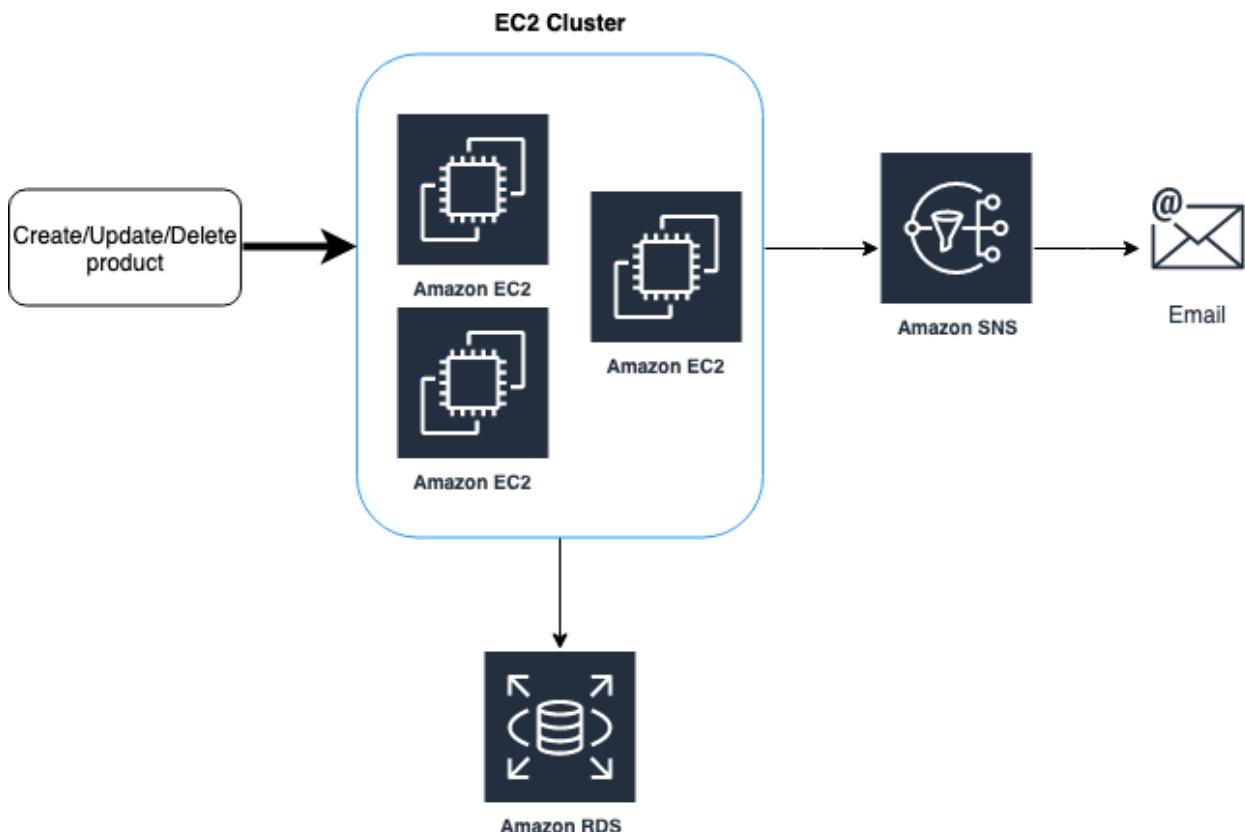


Diagrama da aplicação com SNS

No momento, como não existe ainda uma aplicação capaz de receber as mensagens que serão geradas aqui, será utilizada a funcionalidade do SNS para entregar mensagens a um endereço de e-mail configurável. Dessa forma, será possível receber uma cópia da mensagem.

12.1 - Criando um tópico SNS

A criação de um tópico SNS é bem simples. Para começar, acesse o console da AWS e localize o serviço SNS. No menu lateral esquerdo, escolha a opção **Topics**. Essa é a tela onde todos os tópicos são exibidos.

Clique no botão **Create topic**, localizado no canto superior direito dessa tela, para iniciar o processo de criação de um novo tópico.

Na tela que aparecer, digite apenas o nome do tópico, que nesse exemplo deverá ser **product-events**. Para finalizar, clique no botão **Create topic**, localizado no canto inferior direito.

Após a criação do tópico, suas informações serão exibidas no console da AWS, como mostra a figura a seguir:

The screenshot shows the AWS SNS console. In the top navigation bar, 'Amazon SNS' is selected, followed by 'Topics' and 'product-events'. The main title 'product-events' is displayed prominently. A 'Details' section is visible, containing the 'Name' field with the value 'product-events'. Below it is the 'ARN' field, which contains the value 'arn:aws:sns:us-east-1:666336910744:product-events', enclosed in a red rectangular border. At the bottom of the screen, a message 'Detalhes do tópico criado' is displayed.

Uma informação muito importante dessa tela é o **Amazon Resource Name**, ou simplesmente ARN, que é uma identificação de um recurso da AWS, no seguinte formato:

`arn:aws:<tipo do recurso>:<código da região>:<identificação da conta>:<nome do recurso>`

O ARN será necessário pela aplicação `aws_project01`, pois é através dele que o tópico será identificado para a publicação das mensagens.

12.2 - Inscrevendo um e-mail para receber notificações

Ainda na tela de detalhes do tópico SNS, é possível criar uma inscrição para receber por e-mail as notificações geradas nesse tópico. Para isso, clique no botão `Create subscription`, localizado na parte central à direita da página.

Na página que abrir, configure o parâmetro `Protocol` com o valor `Email` e no campo `Endpoint` configure um e-mail para receber as notificações, como mostra a figura a seguir:

Create subscription

Details

Topic ARN

X

Protocol
The type of endpoint to subscribe

▼

Endpoint
An email address that can receive notifications from Amazon SNS.

Criando uma inscrição por e-mail

Depois disso, clique no botão `Create subscription` localizado no canto inferior da tela. A AWS deverá enviar um e-mail para o endereço configurado, para que a inscrição seja confirmada. Por isso, quando receber tal e-mail, confirme a inscrição para poder continuar.

A ideia aqui é fazer com que esse endereço de e-mail receba as mensagens desse tópico, pois ainda não existe outra aplicação capaz de receber as notificações que serão enviadas nesse exemplo.

12.3 - Enviando notificações via SNS

Essa seção detalha os passos para que a aplicação `aws_project01` possa enviar mensagens para o tópico SNS que foi criado, bem como atualizar a definição da tarefa do ECS para tal.

O tópico SNS que foi criado na seção anterior possui um parâmetro chamado **ARN**, que é um identificador do recurso AWS. Seu valor deve ser passado para a aplicação `aws_project01` para que ela possa enviar as notificações através desse tópico.

O valor do ARN deverá ser passado para aplicação da mesma forma que o endereço do banco RDS e suas credenciais de acesso foram passadas no capítulo anterior, ou seja, como variáveis de ambiente do *container*, pois assim o Spring Boot pode reconhecê-las e inseri-las em seu contexto. Por essa razão que a definição da tarefa do ECS deverá ser alterada.

12.4 - Atribuindo a permissão de acesso ao SNS ao papel `ecsTaskExecutionRole`

A aplicação `aws_project01` está rodando com as permissões definidas no papel `ecsTaskExecutionRole`, conforme a criação e a configuração da definição da tarefa `task-01` que diz como ela deve ser executada. As permissões desse papel podem ser observadas no console da AWS, dentro do serviço IAM.

Para que a aplicação possa publicar mensagens em qualquer tópico SNS, é necessário adicionar tal permissão ao papel `ecsTaskExecutionRole`. Para isso, abra o console da AWS e localize pela seção onde as configurações do IAM são feitas.

Nessa página, clique no menu `Roles` e localize o papel `ecsTaskExecutionRole`. Clique nesse papel para abrir a tela que configura suas permissões.

Na tela que aparecer, clique no botão `Attach policies`, como mostra a figura a seguir:

Identity and Access Management (IAM)

▼ AWS Account (666336910744)

Dashboard

Groups

Users

Roles

Policies

Identity providers

Account settings

Credential report

 Search IAM

▼ AWS Organizations

Organization activity

Adicionando a permissão de acesso ao SNS

A nova tela exibirá todas as políticas que podem ser adicionadas a um papel. No filtro de políticas, localize a que possui o nome `AmazonSNSFullAccess`. Essa política dá plenos poderes de acesso ao serviço SNS.

Marque essa política e clique no botão `Attach policy`. Agora o papel `ecsTaskExecutionRole` possui essa política, logo, tudo o que rodar com esse papel, terá as permissões que estão definidas nele, como é o caso da aplicação `aws_project01` que agora possui permissão para acessar o SNS.



Essas políticas podem ser customizadas por tipo de operação e também podem ser limitadas a um recurso específico.

Roles > `ecsTaskExecutionRole`

Summary

Role ARN	arn:aws:iam::666336910744:role/ecsTaskExecutionRole
Role description	Execution role for Amazon ECS tasks
Instance Profile ARNs	
Path	/
Creation time	2023-08-22T14:45:00Z
Maximum CLI/API session duration	11 hours

Permissions

Trust relationships

▼ Permissions policies

Attach policies

12.3.1 - Configurando a aplicação para acessar o tópico SNS

Para começar a configurar a aplicação `aws_project01` para poder acessar o tópico SNS criado, é necessário adicionar uma nova propriedade no arquivo `application.properties` do projeto, para que receba o ARN desse tópico, como no exemplo a seguir:

```
aws.sns.topic.product.events.arn=product-events
```

O valor dessa propriedade será substituído por uma variável de ambiente de mesmo nome, que será inserida no *container* da aplicação, quando uma nova definição de tarefa do ECS for criada para essa nova versão da aplicação.

Também será necessário uma nova propriedade para representar a região AWS onde a aplicação está em execução, como no trecho a seguir:

```
aws.region=us-east-1
```

Esse valor será necessário para configurar o cliente SNS da aplicação `aws_project01`, e também será passada como variável de ambiente para o *container* da aplicação.

É interessante deixar o máximo de propriedades configuráveis, para que a aplicação seja versátil o suficiente para ser executada em outras regiões e consequentemente com outros recursos AWS.

Agora é preciso adicionar a dependência do cliente SNS que será utilizado pela aplicação. Para isso, abra o arquivo `build.gradle` e adicione a seguinte linha na seção `dependencies`:

```
compile group: 'com.amazonaws', name: 'aws-java-sdk-sns', version: '1.11.613'
```

A próxima etapa é criar uma classe para a configuração do cliente SNS da aplicação, de tal forma que ela possa ser executada durante a sua inicialização. Felizmente, o Spring Boot permite que isso possa ser feito utilizando anotações.

Para fazer essa classe de configuração, crie um novo pacote no projeto chamado `config`. Em seguida, crie uma nova classe dentro desse pacote chamado `SnsConfig`, como no trecho a seguir:

```
package br.com.siecola.aws_project01.config;

import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;
import com.amazonaws.services sns AmazonSNS;
import com.amazonaws.services sns AmazonSNSClientBuilder;
import com.amazonaws.services sns.model.Topic;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SnsConfig {
}
```

No trecho anterior, já estão colocadas todas as bibliotecas que serão utilizadas nessa classe. A seguir, adicione dois atributos privados:

```
@Value("${aws.region}")
private String awsRegion;

@Value("${aws.sns.topic.product.events.arn}")
private String productEventsTopic;
```

Eles guardarão os valores de qual região a aplicação está em execução e o ARN do tópico para envio de eventos de produtos.

A anotação @Value é utilizada para indicar que o valor do atributo deverá ser preenchido pela propriedade definida em seu parâmetro. Nesse caso, as propriedades já foram definidas no arquivo application.properties.

Agora é necessário criar um método para retornar o cliente SNS que será utilizado pela aplicação. Para esse caso, é interessante utilizar o Bean do Spring, um objeto que criado e gerenciado por ele em tempo de execução. Veja como deve ficar no trecho a seguir:

```
@Bean
public AmazonSNS snsClient() {
    return AmazonSNSClientBuilder.standard()
        .withRegion(awsRegion)
        .withCredentials(new DefaultAWSCredentialsProviderChain())
        .build();
}
```

Em qualquer ponto da aplicação que um objeto do tipo AmazonSNS for necessário, será utilizada uma instância devolvida por esse método. Esse é o cliente SNS que será utilizado para envio de

notificações através do tópico criado. Perceba que ele é configurado com o a região em que a aplicação está em execução.

Um outro parâmetro que é configurado no cliente SNS é credencial do contexto que a aplicação está em execução na AWS. Como já foi visto no capítulo 4 e também no capítulo 9, para que uma aplicação seja executada na AWS, é necessário que ela assuma um IAM com um papel contendo suas permissões para poder acessar outros serviços AWS. O método `DefaultAWSCredentialsProviderChain` busca por essas credenciais dentro do contexto de execução da aplicação e configura o cliente SNS.

O último método necessário nessa classe é o responsável por devolver um objeto do tipo `Topic`, que será necessário toda vez que uma mensagem tiver que ser publicada num tópico SNS. Veja como ele deve ficar:

```
@Bean(name = "productEventsTopic")
public Topic snsProductEventsTopic() {
    return new Topic().withTopicArn(productEventsTopic);
}
```

Aqui a anotação `Bean` recebe um nome, pois pode ser necessário criar outros do mesmo tipo `Topic`, sendo necessário diferenciá-los através desse nome.

A criação do objeto `Topic` utiliza o atributo privado `productEventsTopic`, que será preenchido por uma variável de ambiente com o ARN do tópico criado para a publicação de eventos de produtos.

Pronto! Isso é suficiente para configurar a aplicação `aws_project01` para utilizar o AWS SNS.

12.3.2 - Criando um serviço para envio de notificações

O formato do corpo da mensagem a ser enviada pelo tópico SNS é livre, mas é interessante definir um padrão para que todas as aplicações que receberem essa mensagem sejam capazes de identificar detalhes do evento. O trecho a seguir mostra como ficará a mensagem a ser publicada no SNS:

```
{
    "eventType": "PRODUCT_CREATED",
    "data": "{\"productId\":4, \"code\":\"COD4\", \"username\":\"matilde\"}"
}
```

Para isso é necessário criar dois modelos para representar as seguintes entidades:

- Um envelope da mensagem que está sendo publicada, contendo o tipo do evento e o dado que carrega;
- O evento em si, contendo informações sobre a alteração realizada em um produto.

Sobre o tipo do evento, é possível criar um enum no Java para melhor defini-lo. Por isso, crie um novo pacote chamado `enums` e acrescente a seguinte classe a ele:

```
public enum EventType {  
    PRODUCT_CREATED,  
    PRODUCT_UPDATE,  
    PRODUCT_DELETED  
}
```

Esses eventos identificarão melhor o tipo do evento publicado no SNS. Futuramente ele será modificado para comportar outros tipos de eventos.

Agora, para criar o modelo que define o envelope da mensagem, crie uma nova classe no pacote `model`, chamada `Envelope`, como no trecho a seguir:

```
public class Envelope {  
    private EventType eventType;  
    private String data;  
  
    //getters and setters  
}
```

Repare no atributo `eventType`. Ele é do tipo do enum recém criado e carregará o tipo do evento.

O atributo `data` conterá a mensagem do evento serializada em formato JSON.

Para definir o modelo do evento de produtos, crie uma nova classe no mesmo pacote `model` chamado `ProductEvent`, como no trecho a seguir:

```
public class ProductEvent {  
    private long productId;  
    private String code;  
    private String username;  
  
    //getters and setters  
}
```

A ideia é enviar pelo tópico SNS alguns dados do produto que sofreu alteração, ao invés de todo ele, além do usuário responsável por essa alteração, caso um dia a aplicação possua um mecanismo de autenticação de usuários.

Apesar o código necessário para envio de mensagens pelo SNS ser muito simples, é interessante criar um serviço do Spring para fazer esse trabalho. Dessa forma, ele poderá ser reutilizado e chamado em diversos pontos da aplicação. Para isso, crie uma nova classe no pacote `service` chamada `ProductPublisher`, como no trecho a seguir:

```

@Service
public class ProductPublisher {
    private static final Logger log = LoggerFactory.getLogger(
        ProductPublisher.class);

    private AmazonSNS snsClient;
    private Topic productEventsTopic;
    private ObjectMapper objectMapper;

    public ProductPublisher(AmazonSNS snsClient,
                           @Qualifier("productEventsTopic") Topic productEventsTopic,
                           ObjectMapper objectMapper) {
        this.snsClient = snsClient;
        this.productEventsTopic = productEventsTopic;
        this.objectMapper = objectMapper;
    }
}

```

A anotação `@Service` faz com que uma instância dessa classe seja criada e gerenciada pelo Spring, como já foi visto com o serviço de exemplo `TestService` no capítulo 5.

O construtor dessa classe recebe 3 parâmetros:

- `AmazonSNS`: esse é o cliente SNS configurado para poder acessar o AWS SNS;
- `Topic`: esse atributo contém o ARN do tópico SNS para envio de eventos de produtos;
- `ObjectMapper`: um serializador utilizado para converter objetos Java em JSON e vice-e-versa.

Esses atributos serão utilizados no método para envio da notificação no tópico SNS, como no trecho a seguir:

```

public void publishProductEvent(Product product, EventType eventType,
                                String username) {
    ProductEvent productEvent = new ProductEvent();
    productEvent.setProductId(product.getId());
    productEvent.setCode(product.getCode());
    productEvent.setUsername(username);

    Envelope envelope = new Envelope();
    envelope.setEventType(eventType);

    try {
        envelope.setData(objectMapper.writeValueAsString(productEvent));
    }
}

```

```
PublishResult publishResult = snsClient.publish(  
    productEventsTopic.getTopicArn(),  
    objectMapper.writeValueAsString(envelope));  
  
    log.info("Product event sent - ProductId: {} - MessageId: {}",  
        product.getId(), publishResult.getMessageId());  
} catch (JsonProcessingException e) {  
    log.error("Failed to create product event message");  
}  
}
```

Repare na assinatura do método que ele recebe 3 parâmetros:

- O produto em questão;
- O tipo do evento sofrido pelo produto;
- O usuário que realizou a alteração.

Em seguida uma instância de `ProductEvent` é criada com os valores recebidos pelo método.

Uma instância do envelope da mensagem também é criado, contendo o `ProductEvent` serializado em formato JSON utilizando o `ObjectMapper`.

Logo em seguida, o método `publish` do cliente SNS é chamado com 3 parâmetros:

- O tópico SNS onde a mensagem deve ser enviada;
- A mensagem em si, serializada em formato JSON.

O resultado desse método é um objeto do tipo `PublishResult`, que dentre outras coisas, possui uma identificação única da mensagem que foi publicada no SNS.



A identificação única gerada pela publicação do evento no SNS serve de rastreio da mensagem, tanto em quem envia quanto em quem recebe.

Essa classe agora pode ser chamada dentro do *controller* de produtos, toda vez que um produto for alterado. Veja como deve ficar para que a classe `ProductController` tenha uma instância dela:

```
private ProductPublisher productPublisher;

@Autowired
public ProductController(Repository productRepository,
    ProductPublisher productPublisher) {
    this.productRepository = productRepository;
    this.productPublisher = productPublisher;
}
```

O atributo `productPublisher` deve ser criado nessa classe, assim como o construtor deve ser alterado como no trecho abaixo, para receber uma instância dessa classe e atribuir a esse atributo.

Veja como deve ficar para o método de criação de produto da classe `ProductController`, para publicar um evento toda vez que um produto for criado:

```
@PostMapping
public ResponseEntity<Product> saveProduct(
    @RequestBody @Valid Product product) {
    Product productCreated = productRepository.save(product);

    productPublisher.publishProductEvent(productCreated,
        EventType.PRODUCT_CREATED, "matilde");

    return new ResponseEntity<Product>(productCreated,
        HttpStatus.CREATED);
}
```

Depois que o produto é criado, o método `publishProductEvent` do serviço `ProductPublisher` é chamado para publicar tal evento. Aqui, e nos demais casos a seguir, será utilizado um nome de usuário fictício, apenas para demonstrar que é possível informar o usuário responsável pela ação.

No método de alteração de produto desse *controller*, é necessário apenas acrescentar a linha para chamar o método `publishProductEvent`, alterando o tipo de evento para `PRODUCT_UPDATE`, como no trecho a seguir:

```

@PutMapping(path = "/{id}")
public ResponseEntity<Product> updateProduct(
    @RequestBody @Valid Product product, @PathVariable("id") long id) {
    if (productRepository.existsById(id)) {
        product.setId(id);

        productPublisher.publishProductEvent(product,
            EventType.PRODUCT_UPDATE, "doralice");

        return new ResponseEntity<Product>(productRepository.save(product),
            HttpStatus.OK);
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}

```

E finalmente no método de exclusão desse *controller*, apenas acrescente a linha para chamar o método publishProductEvent, alterando o tipo de evento para PRODUCT_DELETED, como no trecho a seguir:

```

@DeleteMapping(path = "/{id}")
public ResponseEntity<Product> deleteProduct(@PathVariable("id") long id) {
    Optional<Product> optProduct = productRepository.findById(id);
    if (optProduct.isPresent()) {
        Product product = optProduct.get();

        productPublisher.publishProductEvent(product,
            EventType.PRODUCT_DELETED, "hannah");

        productRepository.delete(product);
        return new ResponseEntity<Product>(product, HttpStatus.OK);
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}

```

Agora, toda vez que um produto for criado, alterado ou excluído, uma mensagem será publicada no tópico do SNS, informando tal evento.

Para finalizar, altere a versão da aplicação no arquivo `build.gradle` para **0.4.0** e publique no Docker Hub, como feito no capítulo anterior:

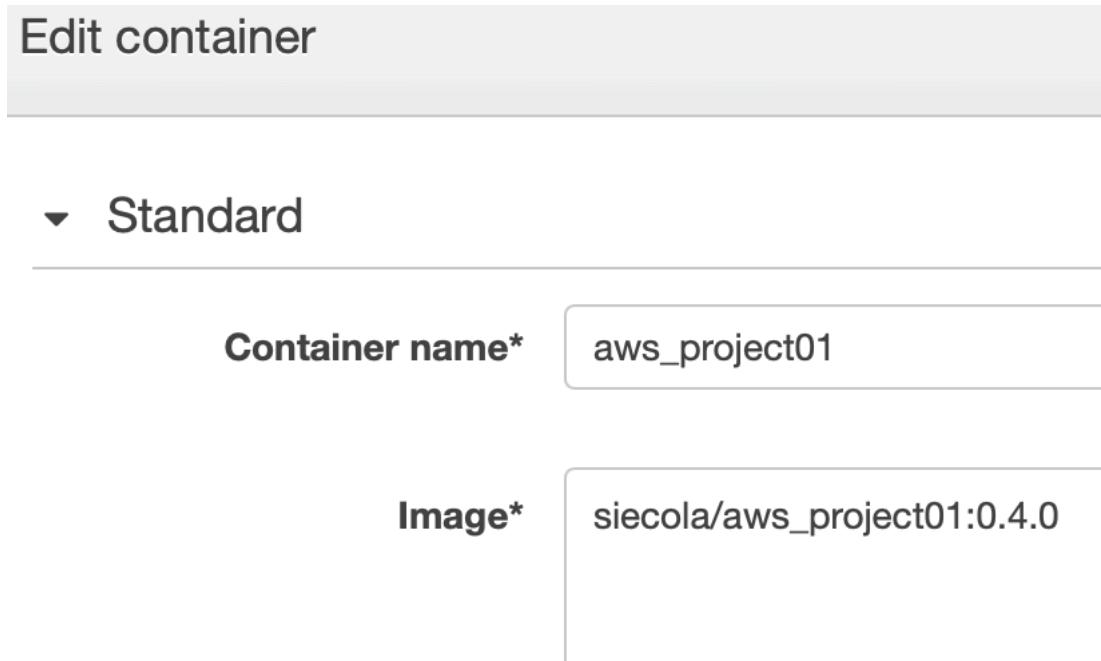
```
bootJar {  
    baseName = 'aws_project01'  
    version = '0.4.0'  
}
```

12.3.3 - Atualizando a tarefa do ECS

Como explicado anterior, a definição da tarefa responsável por executar a aplicação `aws_project01` deve ser alterada com as seguintes informações:

- A versão da imagem do *container* Docker deve ser alterada para **0.4.0**, que é a versão da aplicação gerada e publicada na seção anterior;
- Duas novas variáveis de ambiente devem ser adicionadas à configuração do *container* da aplicação:
 - A região onde a aplicação está sendo executada;
 - O ARN do tópico SNS onde os eventos de produtos serão publicados.

Começando pela versão do *container*, veja na figura a seguir como deve ficar:



Em seguida, como observado anteriormente, crie mais duas variáveis de ambiente a serem passadas para a aplicação:

Environment variables

You may also designate AWS Systems Manager Parameter Store keys or ARNs using the 'valueFrom' field. ECS will pass these values to your container at run-time.

Key	Value	
SPRING_DATASOURCE_PASSWORD	Value	admin123456
SPRING_DATASOURCE_URL	Value	jdbc:mariadb://aws-project01-db.c1mb
SPRING_DATASOURCE_USERNAME	Value	admin
AWS_REGION	Value	us-east-1
AWS_SNS_TOPIC_PRODUCT_EVENTS	Value	arn:aws:sns:us-east-1:666336910744:product-events

Variáveis de ambiente para o container

Os valores dessas duas variáveis serão atribuídas às seguintes propriedades dentro da aplicação `aws-project01`:

```
aws.region=us-east-1  
aws sns topic product events arn=product-events
```

Ou seja, ao invés dos valores que foram configurados no arquivo `application.properties` do projeto, elas assumirão os valores que foram configurados na seção do `container` da nova revisão da definição da tarefa do ECS.

O valor da variável `AWS_SNS_TOPIC_PRODUCT_EVENTS_ARN` é o ARN do tópico SNS que foi criado nesse capítulo. Ele pode ser obtido no console do SNS, dentro das configurações do tópico, como pode ser visto na figura a seguir:

The screenshot shows the AWS SNS Topics page. The URL in the address bar is "Amazon SNS > Topics > product-events". The main title is "product-events". Below it, there's a "Details" section. Under "Name", it says "product-events". Under "ARN", it shows "arn:aws:sns:us-east-1:666336910744:product-events". The ARN field is highlighted with a red border.

Buscando o valor do ARN do tópico

Feitas essas configurações, conclua a revisão da definição da tarefa.

Depois de criada a nova revisão da tarefa, atualize o service-01 para utilizá-la, como feito no capítulo anterior, na seção 11.5.

12.3.4 - Testando o envio de notificações pelo tópico SNS

Para testar o envio de notificações pela aplicação aws_project01, através do tópico SNS criado, basta acessar as seguintes operações do serviço de gerenciamento de produtos, criado no capítulo 11, através do Postman:

- Criação de produtos;
- Alteração de produtos;
- Exclusão de produtos.

Qualquer uma dessas operações irá chamar o serviço de publicação de eventos de produtos, o que fará com que uma mensagem seja publicada no tópico SNS. Esse processo irá criar uma mensagem de log que pode ser vista no CloudWatch Logs, como no exemplo a seguir:

```
b.c.s.a.service.ProductPublisher : Product event sent - ProductId: 4 - MessageId: ea\438575-1452-58c3-8202-d0a65bf228f6
```

Repare que o id do produto foi impresso no log, juntamente com a identificação única da mensagem gerada pelo tópico SNS.

A pública da mensagem também irá disparar uma mensagem para o endereço de e-mail que foi inscrito no tópico SNS, como no exemplo a seguir:

AWS Notification Message ➔ Inbox ×



AWS Notifications <no-reply@sns.amazonaws.com>
to me ▾

```
{"eventType": "PRODUCT_CREATED", "data": "{\"productId\": 4, \"code\": \"COD4\", \"username\": \"matilde\"}"}
```

E-mail de notificação do SNS

Repare no conteúdo da mensagem:

```
{
  "eventType": "PRODUCT_CREATED",
  "data": "{\"productId\": 4, \"code\": \"COD4\", \"username\": \"matilde\"}"
}
```

A mensagem em si está no formato JSON do modelo da classe Envelope e o conteúdo está em uma string, também no formato JSON, do modelo da classe ProductEvent.

Obviamente a ideia de enviar a mensagem por e-mail foi com propósitos didáticos, para que o leitor possa observar seu conteúdo.



É importante lembrar que o código do projeto está no GitHub e pode ser acessado [aqui](#)¹⁴.

12.4 - Conclusão

Nesse capítulo foi discutido como publicar eventos e mensagens utilizando o AWS SNS, um serviço muito simples para tal propósito.

O capítulo 13 irá discutir como consumir essa mensagem através do AWS SQS, um serviço de filas da Amazon, igualmente simples ao SNS, principalmente se utilizado com bibliotecas como Java Messaging Service API.

¹⁴https://github.com/siecola/aws_project01

13 - Amazon Simple Queue Service

O Simple Queue Service da Amazon, ou apenas SQS, é um serviço de filas capaz de trafegar mensagens entre aplicações e serviços, sem a necessidade da criação de *middlewares* para o gerenciamento dessa tarefa.

Com o SQS é possível enviar e receber mensagens de forma descomplicada e eficiente, sem a preocupação de armazenamento de eventos gerados e não lidos.

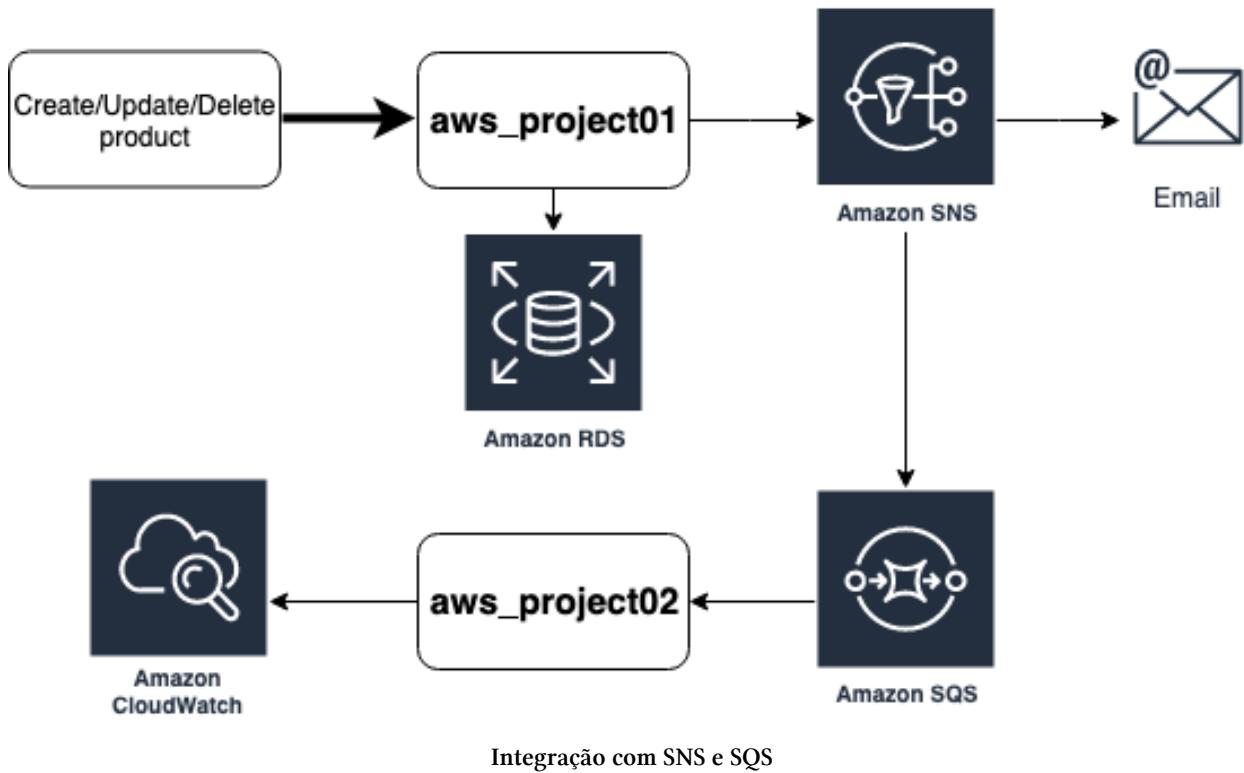
A fonte de informação de um SQS pode ser uma aplicação em execução em um EC2, um Lambda ou um SNS. Nesse último caso, é possível criar uma subscrição entre o SQS e o SNS, de tal modo que a cada mensagem publicada no SNS seja automaticamente copiada para a fila.

Existem dois tipos de fila:

- **Standard:** esse tipo possui uma capacidade de entrega de mensagens superior, com a garantia de entrega da mensagem de pelo menos uma vez e não garante a ordem entre as mensagens que foram publicadas.
- **FIFO:** esse tipo de fila tem uma capacidade de entrega inferior, porém entrega cada mensagem somente um vez, garantindo a ordem em que foi publicada.

A escolha entre os dois tipos de fila deve levar em conta suas vantagens e desvantagens. Este capítulo irá focar na fila do tipo *standard*.

O intuito desse capítulo é criar um consumidor das mensagens geradas pela aplicação `aws_project01` e que estão sendo publicadas no AWS SNS, conforme implementado no capítulo 12. Para isso será criado uma outra aplicação, conforme ilustra a figura a seguir:



Perceba que não existe acoplamento direto entre as duas aplicações, pois quem publica as mensagens no SNS não possui conhecimento ou dependência de quem lê através do SQS. Esse talvez seja um dos mais importantes conceitos desse tipo de arquitetura.

Essa nova aplicação seguirá o mesmo formato que a aplicação **aws_project01**, ou seja, será criado uma imagem Docker que será executada em um serviço no `cluster-01` do ECS.

13.1 - Porque usar um SQS

Duas das grandes questões em uma arquitetura construída com microserviços é o acoplamento entre as diferentes partes e a forma de comunicação entre elas: assíncrona ou síncrona. Sobre a sincronicidade, é importante ressaltar que a melhor escolha depende da operação sendo executada, ou seja, se a resposta deve ser entregue imediatamente, provavelmente a melhor escolha seja uma comunicação síncrona.

Com filas, é possível reduzir o acoplamento entre partes de um sistema, de tal forma que quem publica um evento não se preocupa quando os receptores irão tratá-lo. Isso faz com que a comunicação entre essas partes seja feita de assíncrona.

Uma outra vantagem dessa forma de comunicação assíncrona é que o receptor pode tratar as mensagens na velocidade em que ele é capaz, ou seja, se milhares de eventos forem publicados em um único minuto, o consumidor poderá levar dezenas de minutos para tratá-los, se eles não forem de suma importância para ele.

Com o AWS SQS essas questões podem ser facilmente resolvidas, se a arquitetura permitir uma comunicação assíncrona entre suas partes.

13.2 - Criando uma nova aplicação

Para esse e o capítulo 14, será necessário criar uma nova aplicação com o Spring Boot. Ela será responsável por algumas funcionalidades que ainda serão criadas, começando pelo propósito desse capítulo: ler os eventos de produtos gerados pela aplicação `aws_project01`.

Essa seção cita os passos necessários para construir a nova aplicação, baseada em capítulos anteriores desse livro.

- Crie uma nova aplicação Spring Boot chamada `aws_project02` utilizando o [Spring Initializr¹⁵](#), seguindo os passos descritos no capítulo 5;
- Selecione a opção `Gradle` na página do Spring Initializr;
- No Spring Initializr, selecione apenas a dependência Web;
- Abra o projeto no IntelliJ IDEA;
- Copie todo o conteúdo do arquivo `build.gradle` de `aws_project01` para seu novo projeto. Ele será adaptado mais adiante;
- Copie todo o conteúdo do arquivo `Dockerfile` de `aws_project01` para seu novo projeto. Ele será adaptado mais adiante.

As alterações que devem ser feitas no arquivo `build.gradle` do novo projeto `aws_project02` são as seguintes:

- Na seção `bootJar`, altere a tag `baseName` para `aws_project02`, pois esse é o nome da nova aplicação;
- Ainda nessa mesma seção, altere a versão da aplicação para `0.1.0`;
- Na seção `dependencies`, remova as três linhas a seguir, pois elas não serão utilizadas nesse projeto:

```
compile group: 'org.mariadb.jdbc', name: 'mariadb-java-client-jre7', version: '1.6.1'  
compile group: 'com.amazonaws', name: 'aws-java-sdk-sns', version: '1.11.613'  
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
```

A alteração que deve ser feita no arquivo `Dockerfile` é apenas alterar o nome da aplicação, na última linha, como no exemplo a seguir:

¹⁵<https://start.spring.io>

```
ENTRYPOINT ["java", "-cp", "app:app/lib/*", "br.com.siecola.aws_project02.AwsProject02A\\
pplication"]
```

As seções a seguir irão descrever como fazer com que a aplicação `aws_project02` consuma as mensagens geradas pela aplicação `aws_project01` no tópico do SNS, utilizando uma fila que será criada no SQS. Da mesma forma, irão citar os passos necessários para fazer com que ela execute em um novo serviço no `cluster-01` no ECS.

13.3 - Configurando o projeto com JMS

O processo de consumir mensagens de uma fila SQS em uma aplicação Spring Boot em Java é bem simples, principalmente quando se utiliza a biblioteca Java Messaging Service, ou JMS. Basicamente, aqui estão os passos que devem ser feitos:

- Adicionar as dependências necessárias no arquivo `build.gradle`;
- Criar uma classe para configurar o cliente SQS;
- Criar uma classe para consumir as mensagens.

O processo de criação da fila será discutido mais adiante.

13.3.1 - Configurando o projeto para trabalhar com SQS e JMS

Para começar a configurar o projeto para trabalhar com SQS e JMS, abra o arquivo `build.gradle` e vá na seção `dependencies` e acrescente as seguintes dependências:

```
compile group: 'com.amazonaws', name: 'aws-java-sdk-sqs', version: '1.11.613'
compile group: 'com.amazonaws', name: 'amazon-sqs-java-messaging-lib', version: '1.0\\
.8'
compile group: 'org.springframework', name: 'spring-jms', version: '5.1.9.RELEASE'
```

Essas são as dependências para poder acessar o AWS SQS, além do JMS.

Da mesma forma que foi feito no capítulo 12, é necessário criar propriedades para que a aplicação possa acessar o SQS, por isso abra o arquivo `application.properties` e acrescente as duas propriedades a seguir:

```
aws.region=us-east-1
aws.sqs.queue.product.events.name=product-events
```

Essas propriedades terão seus valores substituídos por variáveis de ambientes que serão injetadas na imagem do `container` Docker.

Uma terceira propriedade que deve ser adicionada a esse arquivo é a porta em que a aplicação será executada:

```
server.port=9090
```

Esse valor foi escolhido apenas para ficar diferente da aplicação aws_project01.

13.3.2 - Criando a classe para configurar o cliente SQS

Para se trabalhar com o SQS utilizando o JMS é necessário configurar seu cliente dentro da aplicação, da mesma forma como foi feito com o cliente SNS da aplicação aws_project01. Para isso, crie o pacote config no projeto aws_project02 e dentro dele crie uma nova classe chamada JmsConfig, como no trecho a seguir:

```
import com.amazonaws.sqs.javamessaging.ProviderConfiguration;
import com.amazonaws.sqs.javamessaging.SQSConnectionFactory;
import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;
import com.amazonaws.services.sqs.AmazonSQSClientBuilder;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.annotation.EnableJms;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.support.destination.DynamicDestinationResolver;

import javax.jms.Session;

@Configuration
@EnableJms
public class JmsConfig {

    @Value("${aws.region}")
    private String awsRegion;

    private SQSConnectionFactory sqsConnectionFactory;
}
```

A anotação @Configuration indica que essa é uma classe de configuração e que deve ser executada pelo Spring Boot assim que a aplicação for iniciada.

A anotação @EnableJms habilita métodos que forem anotados com uma anotação especial, que será vista na seção seguinte, a se tornarem ouvintes de uma fila.

O atributo awsRegion serve para configurar o cliente SQS sobre qual região da AWS ele deverá trabalhar.

O atributo `sqsConnectionFactory` será utilizado para configura o JMS.

Para continuar, crie um método para devolver um *Bean* de configuração do JMS, como no trecho a seguir:

```
@Bean
public DefaultJmsListenerContainerFactory jmsListenerContainerFactory() {
    sqsConnectionFactory = new SQSConnectionFactory(
        new ProviderConfiguration(),
        AmazonSQSClientBuilder.standard()
            .withRegion(awsRegion)
            .withCredentials(new DefaultAWSCredentialsProviderChain())
            .build());

    DefaultJmsListenerContainerFactory factory =
        new DefaultJmsListenerContainerFactory();
    factory.setConnectionFactory(sqsConnectionFactory);
    factory.setDestinationResolver(new DynamicDestinationResolver());
    factory.setConcurrency("2");
    factory.setSessionAcknowledgeMode(Session.CLIENT_ACKNOWLEDGE);

    return factory;
}
```

Perceba que o atributo `sqsConnectionFactory` é criado com a região em que a aplicação for executada.

Logo em seguida, o cliente JMS é configurado para ser utilizado pelos consumidores das filas. Um dos parâmetros interessantes de sua configuração é o seguinte:

```
factory.setConcurrency("2");
```

Essa configuração define o número de *threads* que a aplicação irá alocar para cada consumidor de mensagens de uma fila. Quanto maior o número de *threads* configurada, mais processamento é dedicado a esse consumidor, o que também faz com que as mensagens sejam lidas mais rapidamente.

O último método nessa classe é o que cria um *Bean* do cliente JMS:

```
@Bean
public JmsTemplate jmsTemplate() {
    return new JmsTemplate(sqsConnectionFactory);
}
```

Toda vez que um consumidor for criado com o JMS, esse *Bean* será utilizado. Veja que ele está utilizando o cliente SQS que foi configurado nessa classe.

13.4 - Consumindo mensagens de um SQS utilizando JMS

Agora que a aplicação já foi configurada para trabalhar com SQS e JMS, pode-se criar a estrutura para consumir as mensagens de eventos de produtos que virão da aplicação `aws_project02`. Para isso será necessário:

- Criar o modelo do envelope da mensagem;
- Criar o modelo do evento de produto;
- Criar o consumidor das mensagens.

Existe um outro modelo que também deverá ser criado, que o que representa a mensagem proveniente do SNS. Toda mensagem que é enviada no SNS e que é publicada em um SQS, possui um envelope próprio. Esse envelope possui algumas informações, como:

- O payload que foi publicado no SNS;
- O tipo da mensagem;
- O ARN do tópico SNS de origem;
- O *timestamp* que a mensagem foi publicada;
- A identificação única da mensagem, ou seja, o `messageId`, que já foi mencionado no capítulo 12 e impresso nos logs do *publisher* de mensagens no SNS.

13.4.1 - Criando os modelos para receber as mensagens

Para criar os modelos necessários para interpretar as mensagens vindas do SQS, comece criando um pacote `model` no projeto `aws_project02`. Em seguida, crie a classe `SnsMessage`, para representar a mensagem proveniente do SNS, como no trecho a seguir:

```
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;

@JsonIgnoreProperties(ignoreUnknown = true)
public class SnsMessage {

    @JsonProperty("Message")
    private String message;

    @JsonProperty("Type")
    private String type;

    @JsonProperty("TopicArn")
```

```
private String topicArn;

@JsonProperty("Timestamp")
private String timestamp;

@JsonProperty("MessageId")
private String messageId;

//getters and setters
}
```

A anotação `@JsonIgnoreProperties` serve para indicar para o conversor de JSON para o objeto Java que propriedades desconhecidas devem ser ignoradas.

Dentro dessa classe estão alguns dos campos enviados pelo SNS, como explicado na seção anterior. A anotação `@JsonProperty` em cada um dos atributos é importante para orientar o conversor de JSON para objeto Java sobre os nomes corretos dos parâmetros na mensagem.

Os dois outros modelos necessários já foram criados no projeto `aws_project01`. São eles:

- Envelope;
- ProductEvent.

Copie-os do pacote `model` do projeto `aws_project01` para o pacote de mesmo nome no projeto `aws_project02`.

Também é necessário copiar o pacote `enum` do projeto `aws_project01` para o projeto `aws_project02`.

Com esses modelos criados, agora é possível criar o consumidor das mensagens, como será detalhado na seção seguinte.

13.4.2 - Criando o consumidor de mensagens

O consumidor de mensagens de um SQS utilizando o JMS nada mais é do um serviço do Spring, com um método anotado com `@JmsListener` com o nome da fila a consumir. Por isso crie um novo pacote no projeto `aws_project02` com o nome de `service`. Dentro desse novo pacote, crie a classe `ProductEventConsumer` como no trecho a seguir:

```
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Service;

import javax.jms.TextMessage;

@Service
public class ProductEventConsumer {
    private static final Logger log = LoggerFactory.getLogger(
        ProductEventConsumer.class);

    private ObjectMapper objectMapper;

    @Autowired
    public ProductEventConsumer(ObjectMapper objectMapper) {
        this.objectMapper = objectMapper;
    }

    @JmsListener(destination = "${aws.sqs.queue.product.events.name}")
    public void receiveProductEvent(TextMessage textMessage)
        throws JMSException, IOException {

    }
}
```

Veja que a classe está anotada com `@Service`, indicando que ela é um serviço do Spring. Ela possui um atributo do tipo `ObjectMapper`, cujo valor é injetado pelo Spring. Ele será útil para converter mensagem em formato JSON para modelos Java.

Perceba também que o método `receiveProductEvent` está anotado com `@JmsListener`. Essa anotação indica que o método é o *listener*, ou seja, o método que ficará ouvindo e consumindo as mensagens da fila cujo o nome está definido no parâmetro `destination` dessa anotação. Seu valor será passado como variável de ambiente no *container* da definição da tarefa no novo serviço que será criado para ser executado no `cluster-01`.

Esse método recebe um parâmetro do tipo `TextMessage`. Ele contém a mensagem proveniente do tópico SNS onde a fila SQS estará inscrita. Dentro dessa mensagem está o evento de produtos publicado pela aplicação `aws_project01`.

Para interpretar essa mensagem é necessário realizar os seguintes passos:

- Ler o conteúdo do atributo `text` do parâmetro `textMessage`;
- Converter esse conteúdo em um objeto do tipo `SnsMessage`, que já foi criado no projeto;
- Ler o conteúdo do atributo `message` do objeto do tipo `SnsMessage`;
- Converter esse conteúdo em um objeto do tipo `Envelope`, já criado nesse projeto. Esse objeto contém a mensagem que foi publicada pela aplicação `aws_project01`;

- Converter o conteúdo do campo data de Envelope em um objeto do tipo ProductEvent, que também já foi criado no projeto.

Esse é o processo para “desempacotar” o evento de produtos gerado pela aplicação aws_project01, enviado ao SNS e publicado no SQS. Veja como deve ficar o método receiveProductEvent no trecho a seguir:

```
@JmsListener(destination = "${aws.sqs.queue.product.events.name}")
public void receiveProductEvent(TextMessage textMessage)
    throws JMSException, IOException {

    SnsMessage snsMessage = objectMapper.readValue(textMessage.getText(),
        SnsMessage.class);

    Envelope envelope = objectMapper.readValue(snsMessage.getMessage(),
        Envelope.class);

    ProductEvent productEvent = objectMapper.readValue(
        envelope.getData(), ProductEvent.class);

    log.info("Product event received - Event: {} - ProductId: {} - " +
        "MessageId: {}", envelope.getEventType(),
        productEvent.getProductId(), snsMessage.getMessageId());
}
```

Veja com ao final do método, o id do produto é impresso no log, juntamente com a identificação única da mensagem gerado no momento em que a aplicação aws_project01 enviou a mensagem ao SNS. Dessa forma é possível criar uma relação entre o evento de publicação da mensagem e o momento em que ela é consumida.

No capítulo 14 esse evento será armazenado em uma tabela Dynamo, para registo dos eventos.

13.5 - Criando uma nova fila no AWS SQS

Agora que tudo já foi preparado no projeto aws_project02 em termos de código, é necessário cria a fila no AWS SQS e inscrevê-la para receber os eventos de alteração de produtos publicados no SNS.

13.5.1 - Criando a fila de eventos de produtos

Para criar a fila que irá receber os eventos de produtos gerados pela aplicação aws_project01, vá no console da AWS e selecione o serviço SQS.



Lembre-se sempre de selecionar a mesma região da AWS. Nesse livro foi adotada a região US East (N. Virginia).

Caso nenhuma fila ainda tenha sido criada, clique no botão Get Started Now.

Na página de criação de uma nova fila, apenas digite seu nome e selecione o tipo Standard Queue, como na figura a seguir:

Create New Queue

Queue Name

product-events

Region US East (N. Virginia)

Standard Queue

Criando uma nova fila

Em seguida, clique no botão Quick-Create Queue.

Depois que a fila for criada, ela será exibida no console do SQS, como mostra a figura a seguir:

The screenshot shows the AWS Simple Queue Service (SQS) console. At the top, there's a navigation bar with tabs for 'Name', 'Queue Type', 'Content-Based Deduplication', and 'Messages Available'. Below this, a table lists one queue: 'product-events' (Standard type, N/A deduplication), with 0 messages available. A message below the table states '1 SQS Queue selected'. Below this message, there are several tabs: 'Details' (which is active and highlighted in orange), 'Permissions', 'Redrive Policy', 'Monitoring', 'Tags', and 'Encryption'. Under the 'Details' tab, the following information is displayed:

- Name:** product-events
- URL:** <https://sqs.us-east-1.amazonaws.com/666336910744/product-events>
- ARN:** arn:aws:sqs:us-east-1:666336910744:product-events
- Created:** 2019-08-20 19:49:40 GMT-03:00
- Last Updated:** 2019-08-20 19:49:40 GMT-03:00
- Delivery Delay:** 0 seconds
- Queue Type:** Standard
- Content-Based Deduplication:** N/A

A small note below the details says 'Fila criada'.

Repare que nos detalhes da fila, há um parâmetro chamado **ARN**. Ele é o que define essa fila como recurso da AWS, da mesma forma como foi explicado no capítulo 12 sobre o ARN do tópico SNS que foi criado.

13.5.2 - Inscrevendo uma SQS em um SNS para receber mensagens de outras aplicações

Depois que a fila foi criada, é necessário inscrevê-la no tópico de eventos de produtos, criado no capítulo 12. Dessa forma ela receberá as notificações da aplicação `aws_project01` sobre esses eventos. Para isso, vá até o console do SNS e localize o tópico `product-events`.

Dentro da tela com os detalhes do tópico `product-events`, clique no botão `Create subscription`, da mesma forma como foi feito para inscrever um endereço de e-mail no capítulo 12.

Na tela para a criação de uma nova inscrição, selecione a opção `Amazon SQS` para a opção `Protocol` e coloque o ARN da fila que foi criada nesse capítulo, como mostra a última figura.

The screenshot shows the 'Create subscription' wizard in the AWS SNS console. The top navigation bar includes 'Amazon SNS > Subscriptions > Create subscription'. The main title 'Create subscription' is displayed prominently. A 'Details' section is open, showing the 'Topic ARN' field containing 'arn:aws:sns:us-east-1:666336910744:produ'. Below it, the 'Protocol' section is set to 'Amazon SQS'. The 'Endpoint' section shows an ARN for an SQS queue: 'arn:aws:sqs:us-east-1:666336910744:produ'. A small note states: 'An Amazon SQS queue that can receive notifications from Amazon SNS.'

170_sqs_03

Perceba que o ARN configurado na figura anterior é o da fila product-events criada nesse capítulo. Tendo configurado esses dois campos, clique no botão `Create subscription` para concluir a operação. Essa ação faz com que todas as mensagens que estão sendo publicadas no tópico `product-events`, pela aplicação `aws_project01`, também serão entregues nessa fila, para serem consumidas pela nova aplicação `aws_project02`.

A última configuração para que essa inscrição da fila no tópico se concretize é a de dar permissão ao tópico SNS de publicar na fila SQS. Para isso, volte no no console do SQS, clique na fila que foi criada nesse capítulo e vá até a aba `Permissions`. Em seguida, clique em `Add a Permission`.

Na janela que se abrir, realize as seguintes configurações:

- Escolha a opção `Allow` para o campo `Effect`. Isso faz com que a regra permita as ações que

serão configuradas a seguir:

- Marque a caixa Everybody para o campo Principal.
- Marque a caixa All SQS Actions (SQS:*) no campo Actions. Isso permite que todas as ações possam ser realizadas na fila, inclusive a de publicar mensagens.

Veja como deve ficar essa janela, até esse momento:

The screenshot shows the AWS IAM 'Add a Permission to product-events' configuration window. It includes fields for Effect (Allow selected), Principal (aws account number(s) and Everybody selected), Actions (All SQS Actions (SQS:*) selected), and a link to Add Conditions (optional).

Add a Permission to product-events

Permissions enable you to control which operations a user can perform on a queue. [Click here](#) to about access control concepts.

Effect Allow Deny

Principal aws account number(s) Everybody (*)

Use commas between multiple values.

Actions --- No Specific Actions --- All SQS Actions (SQS:*)

[Add Conditions \(optional\)](#)

Configurando as permissões da fila

Continuando ainda nessa janela, clique no link ‘Add Conditions (optional)’. Nessa tela será possível restringir qual recurso poderá exercer as ações na fila, como publicar mensagens.

Na parte inferior da mesma janela realize as seguintes configurações:

- Escolha a opção None para o campo Qualifier;
- Escolha a opção ArnEquals para o campo Condition;
- Escolha a opção aws:SourceArn para o campo Key;
- No campo value, digite o ARN do tópico product-events.

Veja como deve ficar essa segunda seção de configurações:

Conditions (optional)

Conditions specify additional restrictions on when a permission can take effect. For more information about using conditions, see the [description of the Condition element](#).

Qualifier	None
Condition	ArnEquals
Key	aws:SourceArn
Value	arn:aws:sns:us-east-1:666336910744:product-events

Use commas between multiple values.

Add Condition

Configurando condições da permissão

Essas configurações fazem com que somente o tópico product-events possa publicar mensagens nessa fila. Para finalizar, clique no Add Condition e em seguida no botão Add Permission.

E isso finaliza as configurações necessárias para que a fila criada nesse capítulo possa receber cópias das mensagens publicada pela aplicação aws_project01 através do tópico product-events criado no capítulo anterior.

13.5.3 - Atribuindo a permissão de acesso ao SQS ao papel ecsTaskExecutionRole

Para fazer com que a aplicação aws_project02 possa acessar o serviço AWS SQS, é necessário adicionar tal permissão ao papel ecsTaskExecutionRole, no qual ela está sujeita. Para isso, execute o mesmo procedimento descrito na seção 12.4 do capítulo 12, para adicionar a permissão AmazonSQSFullAccess a esse papel.

Isso fará com que a aplicação aws_project02 possa ler mensagens da fila product-events.



Tal permissão poderia ser customizada por operação e também por um recurso específico.

13.5.4 - Criando regras para descarte de mensagens

O que aconteceria se, ao tentar tratar uma mensagem da fila product-events, a aplicação não conseguisse interpretar a mensagem e lançasse uma exceção? A resposta é simples, com o JMS e da forma como o método para tal foi implementado, a mensagem voltaria para a fila e seria tratada novamente. Muito provável que a mesma exceção fosse lançada, deixando a aplicação em um *loop* tentando tratar a mesma mensagem, que por alguma razão, não pode ser tratada.

Para evitar esse tipo de comportamento indesejado, é possível criar uma regra na fila que está sendo consumida que, caso uma mesma mensagem lance, por exemplo 3 exceções ao tentar tratar ser tratada, faça com que ela seja encaminhada para uma fila de mensagens mortas. Essa fila recebe um nome especial: **Dead letter queue** ou simplesmente DLQ. Esse mecanismo é conhecido como *redrive policy*.

Felizmente, a criação dessa regra é bem simples e não exige que nenhum código tenha que ser desenvolvido por parte do consumidor das mensagens, ou seja, as exceções que não devem ser tratadas podem ser lançadas pelo método que as trata, como parte do *redrive policy*.

Para começar a configurar esse mecanismo, crie uma nova fila com o nome product-events-dlq. Agora, clique com o botão direito sobre a fila product-events para acessar o menu de configurações. Nesse menu, selecione a opção **Configure queue**.

Na janela que aparecer, vá até a seção Dead Letter Queue Settings, marque a opção **Use Redrive Policy**. Em seguida, preencha o campo **Dead Letter Queue** com o nome da nova fila criada para a DLQ, ou seja product-events-dlq.

Para finalizar, preencha o campo **Maximum Receives** com o número máximo que uma mensagem deve ser tratada, antes de ser encaminhada para a DLQ. Nesse campo, coloque o valor 3.

Veja como deve ficar essa tela de configuração do *redrive policy*:



Para finalizar, clique no botão **Save Changes**. Essa configuração deve aparecer na aba **Redrive Policy** da fila product-events.

Uma das coisas interessantes de se fazer o mecanismo, além de proteger para que a aplicação não

fique em *loop* tentando tratar uma mensagem, é permitir seu processamento posterior por um outro mecanismo de análise de mensagens descartadas.

13.6 - Executando a aplicação no cluster-01 do ECS

Agora que a infraestrutura da fila já foi criada, é necessário criar tudo o que é necessário para a execução da nova aplicação `aws_project02`, que consiste em:

- Publicar a imagem da aplicação `aws_project02` no Docker Hub;
- Criar uma definição de tarefa no ECS. Chame-a de `task-02`;
- Criar um *application load balancer* para ser usado pelo serviço no ECS;
- Criar um serviço para execução dessa tarefa. Chame-o de `service-02`;
- Não é obrigatório criar políticas de *auto scaling* para o novo serviço;
- Colocar o serviço para ser executado no `cluster-01` já existente.

Felizmente esse passos já foram detalhados no capítulo 9, por isso siga-os novamente, mas agora com algumas observações importantes:

- No Docker Hub, crie um novo repositório para a aplicação `aws_project02`. Isso foi explicado no capítulo 8;
- A porta que a aplicação `aws_project02` deve ser configurada é a 9090, por isso tome esse cuidado ao configurar todas as portas ao longo do processo;
- É necessário adicionar a porta 9090 ao grupo de segurança criado para o `cluster-01`, conforme mostra a figura a seguir:

The screenshot shows the AWS EC2 Dashboard with the 'Security Groups' section selected. The left sidebar includes links for EC2 Dashboard, Events, Tags, Reports, Limits, Instances, Images, Elastic Block Store, Network & Security, Security Groups, Load Balancing, and Target Groups. The 'Security Groups' section is expanded, showing sub-links for Elastic IPs, Placement Groups, Key Pairs, and Network Interfaces. The main pane displays a list of security groups, with 'sg-006ee2d49de6b4db6' selected. Below the list, the 'Inbound' tab is active, showing a table of rules. One rule for 'Custom TCP' on port 9090 is highlighted with a red box.

Type	Protocol	Port Range	Source	Description
Custom TCP	TCP	8080	0.0.0.0/0	
Custom TCP	TCP	9090	0.0.0.0/0	
MYSQL/Auror	TCP	3306	0.0.0.0/0	

Adicionando a porta 9090 ao grupo de segurança

- Ao configurar a definição da tarefa, lembre-se que a versão da aplicação aws_project02 é a 0.1.0;
- Durante a configuração da imagem Docker na definição da tarefa, lembre-se de configurar as variáveis de ambiente para passar a região da AWS e o nome da fila a ser consumida, como mostra a figura a seguir:

Environment variables

You may also designate AWS Systems Manager Parameter Store keys or ARNs using the 'valueFrom' field. ECS will

<i>Key</i>	<i>Value</i>	
AWS_REGION	Value	us-east-1
AWS_SQS_QUEUE_PRODUCT_EVENTS_NAME	Value	product-events

Variáveis de ambiente do container

- O novo serviço será executado no mesmo *cluster* que já foi criado, ou seja, não é necessário criar um novo *cluster* somente para essa nova aplicação.

Caso tenha dúvidas, volte ao capítulo 9 para verificar os passos necessários que devem ser feitos.

Guarde o endereço público do DNS criado no *application load balancer*, pois ele será necessário para acessar a aplicação `aws_project02` no capítulo 14.

13.7 - Testando o consumidor de mensagens da fila

Agora que a aplicação `aws_project02` já está em execução no novo serviço `service-02` no `cluster-01`, será possível testar integralmente a ligação entre essa aplicação e a `aws_project01`, através da inscrição do SQS `product-events` e o SNS de mesmo nome, utilizado para a publicação dos eventos de produtos.

Para testar o funcionamento do consumidor de mensagens da fila `product-events`, acesse a operação de criação, alteração ou exclusão de produtos da aplicação `aws_project01` através do Postman. Isso fará com que a aplicação `aws_project01` publique uma mensagem no SNS `product-events`.

Como a fila `product-events` está inscrita no SNS de mesmo nome, uma cópia da mensagem do evento de produtos será publicada nela e será consumida pela aplicação `aws_project02`. O método consumidor dessa fila irá tratar o evento, “desempacotando” a mensagem adequadamente:

```
@JmsListener(destination = "${aws.sqs.queue.product.events.name}")
public void receiveProductEvent(TextMessage textMessage)
    throws JMSException, IOException {

    SnsMessage snsMessage = objectMapper.readValue(textMessage.getText(),
        SnsMessage.class);

    Envelope envelope = objectMapper.readValue(snsMessage.getMessage(),
        Envelope.class);

    ProductEvent productEvent = objectMapper.readValue(
        envelope.getData(), ProductEvent.class);

    log.info("Product event received - Event: {} - ProductId: {} - " +
        "MessageId: {}", envelope.getEventType(),
        productEvent.getProductId(), snsMessage.getMessageId());
}
```

A última linha desse código irá gerar um log que pode ser visto no CloudWatch, dentro do `logGroup /ecs-task-02`, como mostra o trecho a seguir:

```
Product event received - Event: PRODUCT_CREATED - ProductId: 11 - MessageId: f2f64ee\8-be9a-544b-913c-1b33d4864898
```

Nesse log é possível observar:

- O evento gerado pela aplicação `aws_project01` e publicado no SNS `product-events`;
- O id do produto que foi criado;
- O id da mensagem gerado pelo SNS, no momento da publicação do evento.

Esse último parâmetro, o `MessageId` é importante para fazer um rastreamento de logs gerados por aplicações distintas, ou seja, é possível associar logs da aplicação que gerou o evento, com logs da aplicação que consumiu tal evento.

Para comprovar tal hipótese, abra os logs da aplicação `aws_project01` no CloudWatch, e localize a mensagem que gerou o evento de publicação da mensagem consumida pela aplicação `aws_project02`. Nesse exemplo mostrado aqui, a mensagem corresponde é a mostrada a seguir:

```
Product event sent - ProductId: 11 - MessageId: f2f64ee8-be9a-544b-913c-1b33d4864898
```

Repare que, além do id do produto, também existe o `MessageId`. Esse número foi gerado pelo SNS no momento da publicação do evento e devolvido para a aplicação `aws_project01`, informando qual era a identificação desse evento publicado. Esse número é o mesmo mostrado pela aplicação `aws_project02`, quando consumiu tal evento.

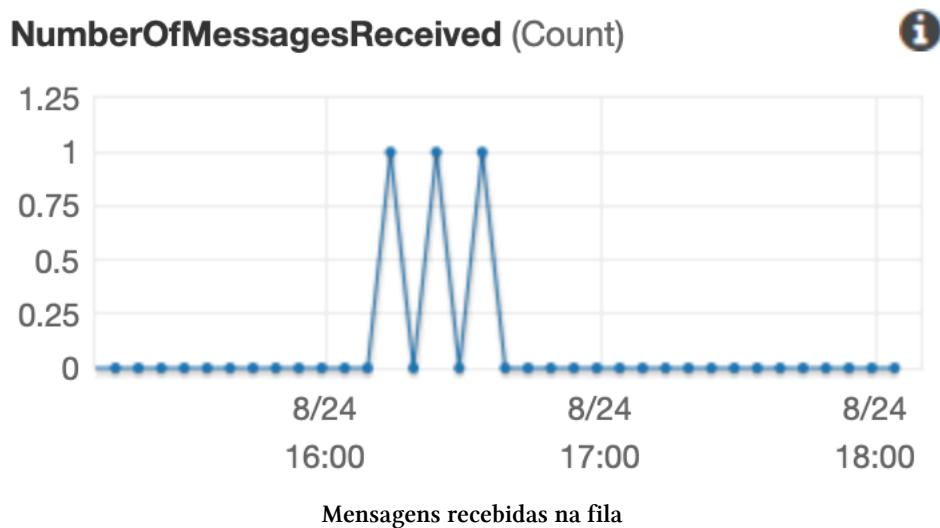
13.8 - Monitorando SQS

O console da AWS para o SQS possui diversos gráficos de monitoramento, disponíveis na aba `Monitoring`, dentre elas:

a) Número de mensagens recebidas:

Esse gráfico é útil para saber se mensagens estão sendo publicadas na fila. Ele deve sempre tender a zero, indicando que existe um consumidor que está tratando as mensagens. Caso ele tende a crescer, pode ser um indicativo de que o consumidor não está lendo as mensagens ou não está sendo capaz de processá-las mais rapidamente do que elas estão sendo publicadas.

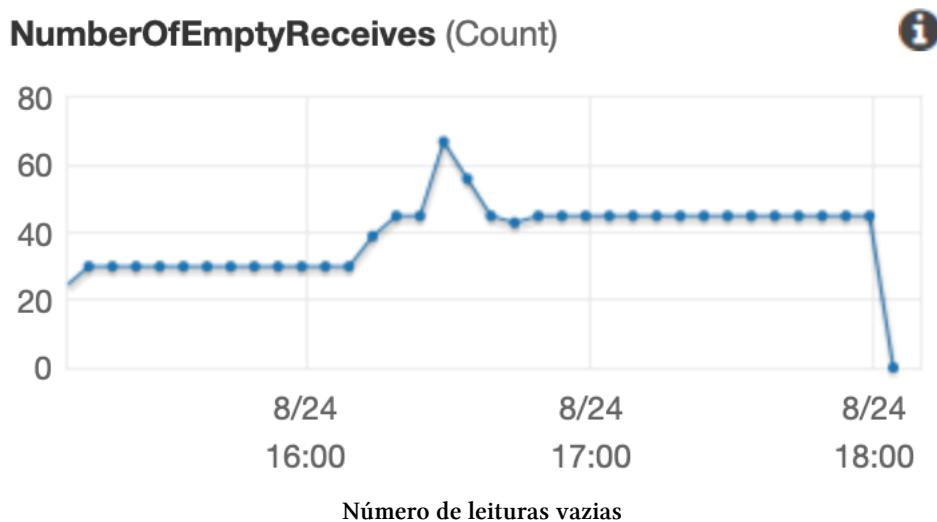
Veja um exemplo, na figura a seguir, do gráfico de monitoramento de mensagens recebidas:



Nesse caso, 3 mensagens foram recebidas nessa fila.

b) Número de leituras realizadas pelo consumidor:

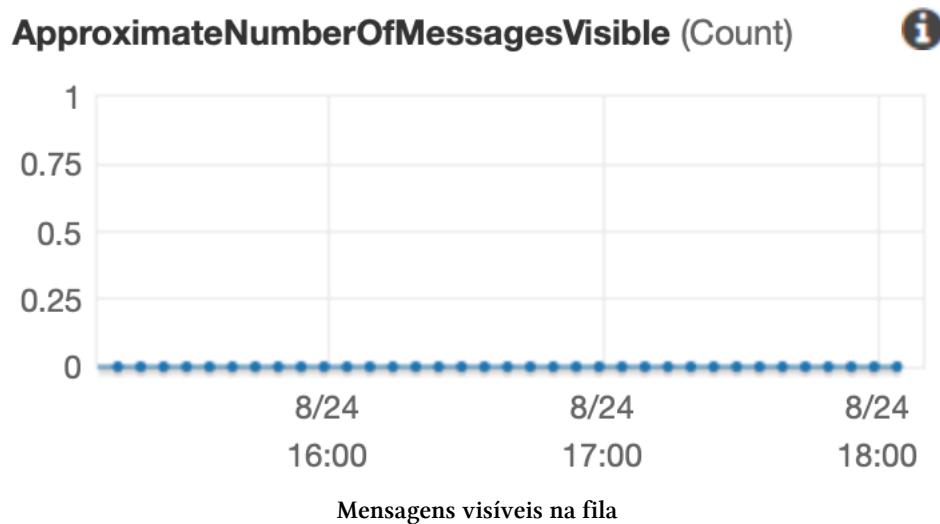
Esse é um outro indicativo de que o consumidor das mensagens está trabalhando corretamente. Nesse gráfico, é possível ver todas as tentativas de leituras que o consumidor fez na fila, que resultaram em leituras vazias:



Embora o conceito de “leituras vazias” possa parecer estranho, ela demonstra que o consumidor está ativamente acessando a fila na busca de novas mensagens.

c) Número de mensagens visíveis:

Esse gráfico mostra, ao longo do tempo, quantas mensagens ficaram visíveis na fila. Obviamente, em uma situação onde o consumidor está constantemente lendo mensagens, esse gráfico deve tender a zero durante todo o tempo, como no exemplo a seguir:



Um valor diferente de zero nesse gráfico pode indicar que uma mensagem ficou muito tempo sem ser tratada, o que pode representar que o consumidor foi interrompido.



É importante lembrar que o código do projeto está no GitHub e pode ser acessado [aqui](#)¹⁶.

13.9 - Conclusão

Esse capítulo mostrou como a integração entre duas microsserviços pode ser simples com a utilização de filas utilizando o AWS SQS.

A configuração do projeto em Java e o código necessário para consumir mensagens de um SQS, utilizando a biblioteca JMS, são relativamente simples e sucintos.

Também foi visto como proteger o consumidor das mensagens de tal forma que ele não entre em *loop* infinito, caso uma mensagem não consiga ser tratada, com a utilização de uma fila DLQ.

No próximo capítulo os eventos de produtos gerados pela aplicação `aws_project01` serão salvos pela aplicação `aws_project02` em uma tabela DynamoDB, um serviço da AWS de banco de dados não-relacionais.

¹⁶https://github.com/siecola/aws_project02

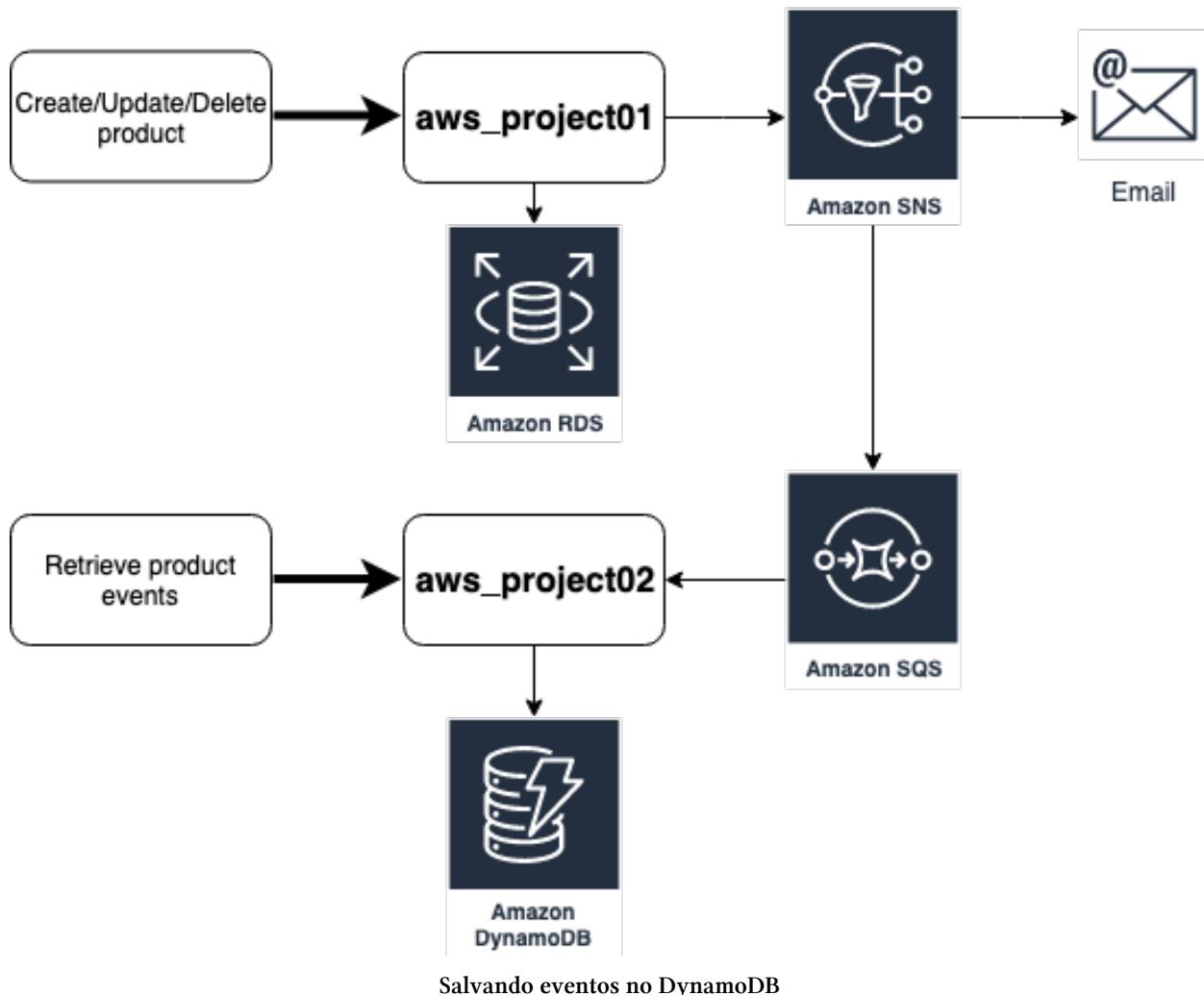
14 - Amazon DynamoDB

DynamoDB é um banco de dados não relacional da Amazon, onde as informações podem ser armazenadas na forma de chave-valor e em documentos. Esse serviço possui uma alta capacidade para suportar operações de leitura e escrita e não requer que sejam criados servidores.

Ele também possui as seguintes características:

- A capacidade de escrita e leitura pode ser escalada de acordo com a demanda;
- Registros podem ser configurados para serem apagados após um tempo;
- É possível configurar um *stream* para aplicações possam ser avisadas sobre alterações em registros;
- Permite a criação de índices para pesquisas;
- É possível alterar a estrutura da tabela, sem a necessidade de execução de scripts para mudanças de *schema* ou algo do tipo.

O intuito deste capítulo é fazer com que a aplicação `aws_project02` salve os eventos provenientes da fila `product-events` em uma tabela do DynamoDB, para que possam ser consultados através de `endpoints` que ainda serão criados nessa aplicação.



As seções a seguir detalham o processo para configurar o projeto `aws_project02` para que possa ser capaz de acessar uma tabela DynamoDB, bem como os passos necessários para sua criação na AWS. Também irá mostrar como visualizar os dados de uma tabela e monitorá-la, através do console da AWS.

14.1 - Criando uma tabela no DynamoDB

Para criar uma nova tabela no DynamoDB, basta acessar seu console na AWS. Dentro dele, acesse a opção o menu lateral esquerdo na opção **Tables**.

Nessa página serão mostradas todas as tabelas existentes da região AWS selecionada.



Da mesma forma como os outros recursos, é importante selecionar a região desejada para a criação da tabela do DynamoDB.

Para começar o processo de criação de uma nova tabela, clique no botão `Create table`. Na tela que aparecer digite o nome da tabela e também o nome do campo que será considerado como sua chave primária, como na figura a seguir:

Create DynamoDB table

DynamoDB is a schema-less database that only requires a table name and primary key. data, and sort data within each partition.

The screenshot shows the 'Create DynamoDB table' interface. The 'Table name*' field contains 'product-events'. The 'Primary key*' dropdown is set to 'Partition key' with 'id' selected. Below this, a 'String' type dropdown is shown. A checkbox labeled 'Add sort key' is present. At the bottom, there is a button labeled 'Configurando nome da tabela'.

Mantenha o tipo da chave primária como String.

Nessa mesma tela, mais abaixo, mantenha a opção `Use default settings` habilitada. Isso fará com que a tabela seja criada com as seguintes características adicionais:

- Não será criado nenhum índice para pesquisas;
- A tabela será criada com uma capacidade de leitura e escrita limitada a um determinado valor;
- Tal capacidade de leitura e escrita não será escalada, caso ultrapasse o limite pre-determinado.



Tais configurações são suficientes para a demonstração das funcionalidades do DynamoDB desse capítulo. Porém, importante ter em mente que a capacidade de leitura e escrita da tabela é algo crítico que deve ser avaliado com cuidado, para uma aplicação real, pois impacta diretamente na disponibilidade do serviço, bem como em seu custo de operação.

Para finalizar o processo de criação da tabela, clique no botão `Create` no canto inferior direito da página. Em alguns segundos a tabela será criada e será exibida na lista de tabelas no console do DynamoDB.

14.1.1 - Configurando o tempo de expiração de registros

Depois que a tabela foi criada, é possível configurá-la para excluir registros automaticamente após um período em que ele foi criado. Essa característica do DynamoDB é interessante quando se deseja

criar registros temporários na tabela, sem a necessidade de se criar um *scheduler* na aplicação para poder removê-los. Essa funcionalidade do DynamoDB é chamada de *time to live*.

Para configurar esse tempo de vida dos registros, acesse a tabela recém criada e vá na aba Overview. Na seção Table details, clique na configuração Manage TTL, como mostra a figura a seguir:

Table details

Table name	product-events
Primary partition key	id (String)
Primary sort key	-
Point-in-time recovery	DISABLED Enable
Encryption Type	DEFAULT Manage Encryption
KMS Master Key ARN	Not Applicable
Time to live attribute	DISABLED Manage TTL
Table status	Active
Creation date	August 24, 2019 at 6:35:18 PM UTC-3

Configurando o TTL dos registros

Na janela que aparecer, simplesmente configure o nome do atributo do registro que representará o valor de seu tempo de vida. Para isso, digite ttl no atributo TTL attribute:

Enable TTL

TTL is a mechanism to set a specific timestamp for expiring item an attribute on the items in the table. The attribute should be a N the timestamp expires, the corresponding item is deleted from the table.

TTL attribute

ttl

Configurando o nome do parâmetro de TTL

Isso fará com que o DynamoDB olhe para o valor desse atributo, em cada registro, de forma especial. Ele deverá conter o *timestamp* em milisegundos, de quando o registro deverá ser apagado automaticamente pelo DynamoDB.

Quando um registro novo for inserido na tabela product-events, a aplicação aws_project02 deverá gravar um valor nesse campo, informando um *timestamp* no futuro. O DynamoDB por sua vez irá apagar esse registro a partir desse tempo gravado.



O tempo em que o DynamoDB apaga os registros marcados com TTL não é preciso. A documentação diz que eles são apagados em um tempo após esse período. Por isso, não utilize essa funcionalidade para fazer controles precisos de tempos em registros marcados com TTL.

Pronto! Agora a tabela product-events está pronta para ser utilizada.

14.2 - Configurando a aplicação para acessar a tabela

Para configurar o projeto aws_project02 para trabalhar com o DynamoDB, são necessários os seguintes passos:

- Adicionar as dependências das bibliotecas necessárias no arquivo build.gradle;
- Criar a classe de modelo que irá representar os registros na tabela do DynamoDB;
- Criar o repositório de dados para acessar a tabela;
- Criar a classe de configuração para acesso ao DynamoDB;

Além desses passos, também é necessário dar a permissão adequada à aplicação em execução no ECS, para acessar o serviço do DynamoDB.

Essa seção discute detalhadamente esses passos.

14.2.1 - Adicionando as dependências ao projeto

Para adicionar as bibliotecas necessárias para se trabalhar com o DynamoDB, abra o arquivo build.gradle do projeto aws_project02 e vá na seção dependencies.

Nessa seção, adicione as duas dependências a seguir:

```
compile group: 'com.amazonaws', name: 'aws-java-sdk-dynamodb', version: '1.11.618'  
compile group: 'com.github.derjust', name: 'spring-data-dynamodb', version: '5.1.0'
```

A primeira dependência é a biblioteca da AWS que permite a criação do cliente de acesso ao DynamoDB. A segunda é uma biblioteca **não oficial** que permite trabalhar com entidades no DynamoDB utilizando o *estilo* do Spring Data.

14.2.2 - Criando a classe modelo

Agora, para criar um modelo que representará as entidades que serão salvas no DynamoDB, vá no pacote model do projeto aws_project02 e crie uma nova classe chamada ProductEventLog, como no trecho a seguir:

```
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAttribute;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBAutoGeneratedKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBHashKey;
import com.amazonaws.services.dynamodbv2.datamodeling.DynamoDBTable;

@DynamoDBTable(tableName = "product-events")
public class ProductEventLog {

    @DynamoDBHashKey
    @DynamoDBAttribute(attributeName = "id")
    @DynamoDBAutoGeneratedKey
    private String id;

}
```

Repare na anotação `@DynamoDBTable`. Ela serve para definir o nome da tabela no DynamoDB que essa classe modelo representa. Esse valor deve ser exatamente o nome da tabela criada no DynamoDB.

O atributo `id` do tipo `String` representa a chave primária da tabela. Durante o processo de criação da tabela `product-events` seu nome foi definido. As anotações nesse atributo configuram as seguintes propriedades:

- **DynamoDBHashKey**: o tipo da chave primária é um *hash*;
- **DynamoDBAttribute**: o nome do atributo na tabela do DynamoDB é o definido no valor `attributeName`;
- **DynamoDBAutoGeneratedKey**: a própria aplicação irá gerar um valor aleatório para a chave primária.

Agora os demais parâmetros dessa classe modelo podem ser criados, como no trecho a seguir:

```
@DynamoDBTypeConvertedEnum
@DynamoDBAttribute(attributeName = "eventType")
private EventType eventType;

@DynamoDBAttribute(attributeName = "productId")
private long productId;

@DynamoDBAttribute(attributeName = "code")
private String code;

@DynamoDBAttribute(attributeName = "username")
private String username;
```

```

@DynamoDBAttribute(attributeName = "timestamp")
private long timestamp;

@DynamoDBAttribute(attributeName = "ttl")
private long ttl;

//getters and setters

```

Para cada atributo, é necessário utilizar a anotação `@DynamoDBAttribute` para definir o nome da coluna na coluna do DynamoDB.

Repare que no atributo do tipo `EventType`, que é um enum que já foi criado no projeto, é necessário utilizar uma outra anotação, a `@DynamoDBTypeConvertedEnum`, para converter seu valor em uma String no momento de salvá-lo e de lê-lo.

O atributo que define o *time to live* do registro não possui características especiais, além ter que se chamar `ttl`, como definido na configuração da tabela no DynamoDB, e ser do tipo `long`. Quando um registro novo for inserido na tabela, esse atributo deverá possuir o *timestamp* de quando deseja-se apagá-lo.

14.2.3 - Criando o repositório de dados

O repositório de dados para tabelas DynamoDB, dentro de uma aplicação Spring Boot, funciona de forma muito parecida com o que foi criado na aplicação `aws_project01`. Mas antes de começar, crie um novo pacote chamado `repository` no projeto `aws_project02`.

Dentro do novo pacote `repository`, crie uma interface chamada `ProductEventLogRepository`, como no trecho a seguir:

```

import br.com.siecola.aws_project02.model.ProductEventLog;
import org.socialsignin.spring.data.dynamodb.repository.EnableScan;
import org.springframework.data.repository.CrudRepository;

@EnableScan
public interface ProductEventLogRepository
    extends CrudRepository<ProductEventLog, String> {
}

```

Com esse repositório de dados será possível salvar e realizar consultas na tabela `product-events` do DynamoDB.

Mais adiante, ainda nesse capítulo, ele será incrementado com novas operações.

14.2.4 - Criando a classe de configuração

Para que a aplicação aws_project02 possa acessar a tabela product-events no DynamoDB, é necessário configurar seu cliente dentro do projeto. Para isso, vá até o pacote config e crie uma nova classe chamada DynamoDBConfig, como no trecho a seguir:

```
import br.com.siecola.aws_project02.repository.ProductEventLogRepository;
import org.socialsignin.spring.data.dynamodb.repository.config.EnableDynamoDBRepositories;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableDynamoDBRepositories(basePackageClasses = ProductEventLogRepository.class)
public class DynamoDBConfig {
```

```
}
```

O funcionamento dessa classe é semelhante à que foi criada para a configuração do cliente SQS no capítulo 13. Ela será executada assim que a aplicação Spring Boot iniciar.

Repare que uma grande diferença aqui é a presença da anotação @EnableDynamoDBRepositories. Ela é usada para definir quais as classes, que representam os repositórios de dados, devem ser mapeados por essa configuração.

Nessa classe é necessário criar 3 beans que serão utilizados pelo Spring Boot para acessar o DynamoDB, como pode ser observado no trecho a seguir:

```
@Configuration
@EnableDynamoDBRepositories(basePackageClasses = ProductEventLogRepository.class)
public class DynamoDBConfig {
    @Bean
    @Primary
    public DynamoDBMapperConfig dynamoDBMapperConfig() {
        return DynamoDBMapperConfig.DEFAULT;
    }

    @Bean
    @Primary
    public DynamoDBMapper dynamoDBMapper(AmazonDynamoDB amazonDynamoDB,
                                           DynamoDBMapperConfig config) {
        return new DynamoDBMapper(amazonDynamoDB, config);
    }

    @Bean
```

```

@Primary
public AmazonDynamoDB amazonDynamoDB() {
    return AmazonDynamoDBClientBuilder.standard()
        .withCredentials(new DefaultAWSCredentialsProviderChain())
        .withRegion(Regions.US_EAST_1).build();
}
}

```

Com esses três últimos métodos na classe `DynamoDBConfig` o projeto já está preparado para acessar a tabela `product-events` do DynamoDB.

14.2.5 - Atribuindo a permissão de acesso ao DynamoDB ao papel `ecsTaskExecutionRole`

Para que a aplicação `aws_project02` tenha permissão para acessar tabelas no DynamoDB, é necessário adicionar tal permissão ao papel `ecsTaskExecutionRole`, no qual ela está sujeita. Para isso, adicione a política `AmazonDynamoDBFullAccess` nesse papel, da mesma forma como foi feito na seção 12.4 do capítulo 12.

14.3 - Salvando entidades na tabela

Agora que tudo já foi configurado no projeto, o repositório de dados pode ser utilizado para escrever os eventos de produtos gerados pela aplicação `aws_project01` no SNS e consumidos pela aplicação `aws_project02` pelo SQS. Para isso, abra a classe `ProductEventConsumer` e adicione o repositório de dados ao seu construtor e também atribua-o a um novo atributo da classe, como no trecho a seguir:

```

private ProductEventLogRepository productEventLogRepository;

@Autowired
public ProductEventConsumer(ObjectMapper objectMapper,
    ProductEventLogRepository productEventLogRepository) {
    this.objectMapper = objectMapper;
    this.productEventLogRepository = productEventLogRepository;
}

```

Com esse repositório agora será possível salvar os eventos de produtos recebidos na tabela `product-events`. Para isso, comece criando um método privado nessa classe para converter o evento recebido pela fila em uma entidade a ser salva nessa tabela, como no trecho a seguir:

```
ProductEventLog buildProductEventLog(Envelope envelope,
                                      ProductEvent productEvent) {
    ProductEventLog productEventLog = new ProductEventLog();
    productEventLog.setEventType(envelope.getEventType());
    productEventLog.setProductId(productEvent.getProductId());
    productEventLog.setCode(productEvent.getCode());
    productEventLog.setUsername(productEvent.getUsername());
    productEventLog.setTimestamp(Instant.now().toEpochMilli());

    productEventLog.setTtl(Instant.now().plus(
        Duration.ofMinutes(10)).toEpochMilli());

    return productEventLog;
}
```

Repare que o objeto do tipo `ProductEventLog`, que será salvo na tabela `product-events` recebe os valores provenientes do envelope e da própria mensagem em si.

O último atributo a ser configurado nesse objeto é o `ttl`. Aqui, apenas como exemplo, ele está sendo configurado com um valor de 10 minutos a frente de seu tempo de criação. Como ele está configurado como o parâmetro de *time to live* da tabela `product-events`, significa que após 10 minutos depois de ser salvo ele será apagado automaticamente pelo DynamoDB.

O parâmetro `id` não foi configurado, pois ele será gerado automaticamente, para ser utilizado como chave primária da tabela.

Tendo criado esse método, utilize-o para criar uma nova entidade a ser salva no DynamoDB, a partir do evento e da mensagem, no final do método `receiveProductEvent`, como no trecho a seguir:

```
ProductEventLog productEventLog = buildProductEventLog(envelope,
                                                       productEvent);
productEventLogRepository.save(productEventLog);
```

Dessa forma, quando a aplicação `aws_project02` receber um evento de produto, gerado pela aplicação `aws_project01`, ele será salvo na tabela `product-events` do DynamoDB, com um tempo de vida de 10 minutos.



A escolha do tempo de vida de um registro no DynamoDB depende da lógica de negócio, que pode ser de alguns minutos, horas ou até dias. O valor escolhido aqui é apenas para poder demonstrar o efeito de forma mais rápida.

14.4 - Visualizando os dados tabela do DynamoDB

Apenas para efeitos didáticos, crie um novo pacote no projeto `aws_project02`, chamado `controller` e uma nova classe chamada `ProductEventLogController`, que será responsável por oferecer operações de consultas aos eventos que gravados por essa aplicação. Dessa forma, será possível consultar, através do Postman, tais eventos.

```
@RestController
@RequestMapping("/api")
public class ProductEventLogController {
    private static final Logger log = LoggerFactory.getLogger(
        ProductEventLogController.class);

    private ProductEventLogRepository productEventLogRepository;

    @Autowired
    public ProductEventLogController(
        ProductEventLogRepository productEventLogRepository) {
        this.productEventLogRepository = productEventLogRepository;
    }
}
```

Agora crie uma operação para retornar todos os eventos presentes na tabela `product-events`, utilizando o repositório de dados `ProductEventLogRepository`, como no trecho a seguir:

```
@GetMapping("/events")
public Iterable<ProductEventLog> getAllEvents() {
    return productEventLogRepository.findAll();
}
```

Essa operação fará uma requisição na tabela `product-events` do DynamoDB e trará todos os registros que estiverem lá. Ela pode ser acessada, através do endereço `/api/events` com o método HTTP GET. Agora, para demonstrar como consultas avançadas também podem ser feitas no DynamoDB, crie um novo método na interface `ProductEventLogRepository` para consultar todos os registros criados por um determinado `username`, como no trecho a seguir:

```
List<ProductEventLog> findAllByUsername(String username);
```

Essa nomenclatura de métodos segue o mesmo padrão do Spring Data. Nesse caso, ele será utilizado para buscar todos os registros que contêm o parâmetro `username` com o mesmo valor do parâmetro do método.

Volte na classe `ProductEventLogController` e crie uma nova operação para buscar todos os eventos por um determinado `username`, como no trecho a seguir:

```
@GetMapping("/events/{username}")
public List<ProductEventLog> findById(@PathVariable String username) {
    return productEventLogRepository.findAllByUsername(username);
}
```

Repare que o novo método do repositório de dados foi utilizado para essa consulta. Essa operação pode ser acessada pelo endereço /api/events/{username}, com o método HTTP GET.

Agora a aplicação aws_project02 já está pronta para ser testada na AWS. Altere sua versão, no arquivo build.gradle para 0.2.0, publique no Docker Hub e atualize a definição da tarefa task-02 para usar a nova imagem e consequentemente o serviço service-02 para utilizá-la, como já foi feito algumas vezes nos capítulos anteriores.

14.4.1 - Scan versus query

No DynamoDB, as operações de consulta podem ser feitas, resumidamente, através das seguintes:

- Através da chave primária de cada registro;
- Através de operações de *scan* por qualquer parâmetro da tabela;
- Através de operações de *query* através de índices configurados na tabela.

As operações de *scan*, que não são feitas através de índices, são mais demoradas, pois o DynamoDB deve varrer por todos os registros da tabela para satisfazer a consulta solicitada.

Para operações de busca que exigem um desempenho melhor, é necessário a criação de índices na tabela e, dessa forma, realizar consultas através desses índices. Porém, esse tópico **não faz parte do escopo desse capítulo**.



Índices devem ser planejados com muito cuidado, pois algumas estratégias requerem que a tabela seja recriada, além de envolver um custo superior na operação da tabela.

14.5 - Testando e monitorando a tabela do DynamoDB

Para testar a implementação realizada nesse capítulo na aplicação aws_project02, depois de promovido sua versão 0.2.0 no service-02 do cluster-01, basta acessar a operação de criação, alteração e exclusão de produtos da aplicação aws_project01. Isso fará com que um evento seja publicado no tópico product-events do SNS, copiado para a fila de mesmo nome no SQS e consumido pela aplicação aws_project02. Esse por sua vez irá persistir tal evento na tabela product-events do DynamoDB, criado nesse capítulo.

Os eventos de produtos criados pela aplicação aws_product02 podem ser visualizados no console do DynamoDB, através da aba Items, como pode ser observado na figura a seguir:

product-events [Close](#)

[Overview](#) **Items** [Metrics](#) [Alarms](#) [Capacity](#) [Indexes](#) [Global Tables](#) [Backups](#) [Triggers](#) [Access control](#)

[Create item](#) [Actions](#) ▾

Scan: [Table] product-events: id ▾

Scan [Add filter](#) Start search

	id	code	eventType	productId	timestamp	ttl (TTL)	username
<input type="checkbox"/>	cebf82f0-f843	COD12	PRODUCT_CREATED	12	1566761817740	1566762417740	matilde
<input type="checkbox"/>	5fad06ee-693	COD1	PRODUCT_UPDATE	1	1566762204475	1566762804475	doralice
<input type="checkbox"/>	16430d68-85c	COD3	PRODUCT_DELETED	3	1566762213484	1566762813484	hannah

Itens na tabela do DynamoDB

Nessa tela, é possível inclusive realizar consultas na tabela, bem como alterar e apagar registros.

Clicando-se em um dos itens, é possível visualizar e editar seus parâmetros, como na figura a seguir:

Edit item

The screenshot shows the AWS Lambda function editor with the title "Edit item". At the top, there are buttons for "Tree" and "List". Below the tree view, there is a list of items under "Item {7}":

- code String : COD12
- eventType String : PRODUCT_CREATED
- id String : cebf82f0-f843-4f0d-a00a-4eb7cd20293d
- productId Number : 12
- timestamp Number : 1566761817740
- ttl Number : 1566762417740
- username String : matilde

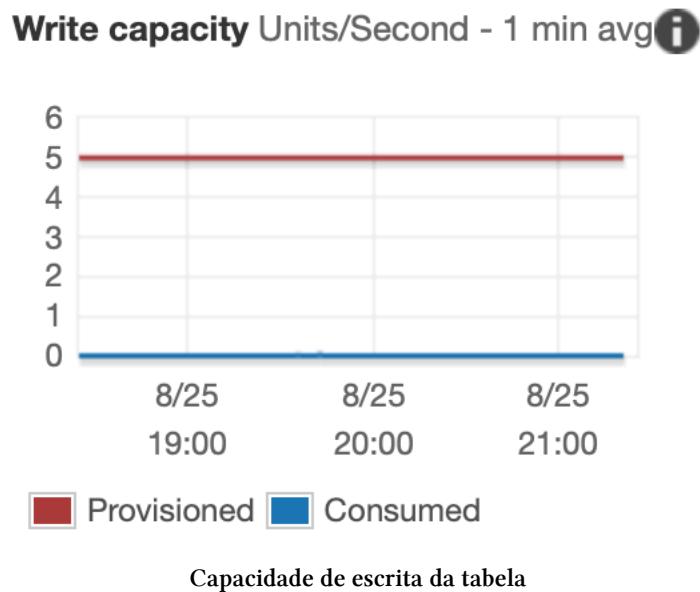
At the bottom, there is a button labeled "Editando um registro".

Os eventos gravados na tabela do DynamoDB também podem ser acessados através das duas operações criadas no *controller* ProductEventLogController, que retorna todos os eventos ou filtrando pelo `username`.



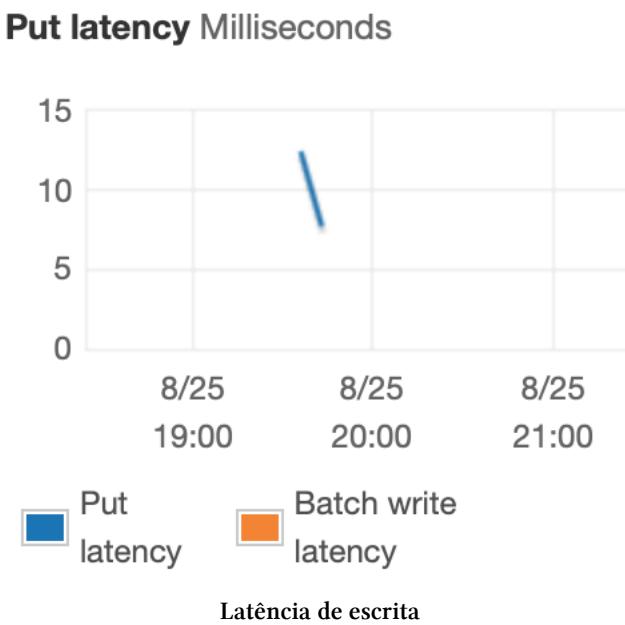
Lembre-se de utilizar o DNS público do *application load balancer*, criado no capítulo anterior, para poder acessar a aplicação `aws_project02` através do Postman.

O console do DynamoDB também possui gráficos para o monitoramento do desempenho de suas operações. Tais gráficos podem ser vistos na aba `Metrics` do console do DynamoDB. A seguir, alguns exemplos:



Nesse gráfico, é possível acompanhar a capacidade de escrita consumida pela aplicação. A linha vermelha indica a capacidade que está provisionada, que no caso da tabela que foi criada, significa o limite que aplicação pode chegar, uma vez que a tabela não está configurada para escalar automaticamente.

O gráfico a seguir mostra a latência na escrita dos itens na tabela:



Acessando-se a operação para listar todos os eventos do *controller ProductEventLogController*, pelo endereço do DNS público do *application load balancer* cluster-01-2-1b, acrescentado do sufixo :9090/api/events pode-se obter todos os eventos gerados, como no trecho a seguir:

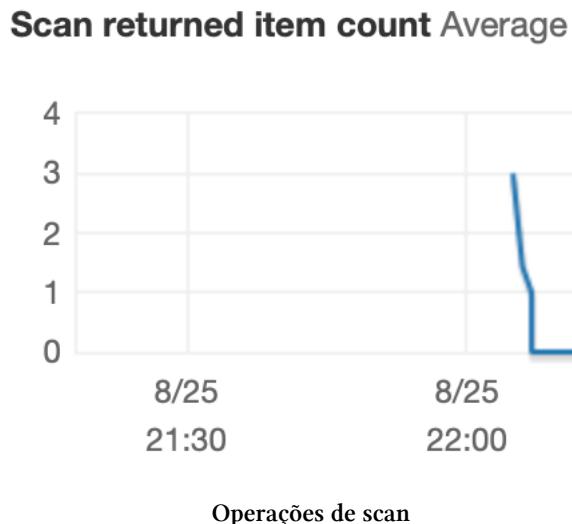
```
[  
  {  
    "id": "5fad06ee-6939-43f8-9f6b-f02d8dd0a846",  
    "eventType": "PRODUCT_UPDATE",  
    "productId": 1,  
    "code": "COD1",  
    "username": "doralice",  
    "timestamp": 1566762204475,  
    "ttl": 1566762804475  
  },  
  {  
    "id": "ceb f82f0-f843-4f0d-a00a-4eb7cd20293d",  
    "eventType": "PRODUCT_CREATED",  
    "productId": 12,  
    "code": "COD12",  
    "username": "matilde",  
    "timestamp": 1566761817740,  
    "ttl": 1566762417740  
  },  
  {  
    "id": "16430d68-8530-40b4-b173-7c3aaedc27e9",  
    "eventType": "PRODUCT_DELETED",  
    "productId": 3,  
    "code": "COD3",  
    "username": "hannah",  
    "timestamp": 1566762213484,  
    "ttl": 1566762813484  
  }  
]
```

Perceba que cada registro possui seu próprio `id`, que foi gerado de forma aleatória.

Acessando a operação de buscar os eventos realizados por um determinado `username`, com o sufixo `:9090/api/events/{username}`, é possível fazer uma busca nos registros da tabela do DynamoDB e obter o seguinte resultado:

```
[  
  {  
    "id": "cebf82f0-f843-4f0d-a00a-4eb7cd20293d",  
    "eventType": "PRODUCT_CREATED",  
    "productId": 12,  
    "code": "COD12",  
    "username": "matilde",  
    "timestamp": 1566761817740,  
    "ttl": 1566762417740  
  }  
]
```

Ao realizar operações como essa, algumas vezes, é possível obter um gráfico das operações de *scan* realizadas na tabela, como no exemplo a seguir:



Esses gráficos auxiliam na investigação de possíveis problemas, bem como no ajuste adequado do desempenho de cada tabela no DynamoDB.



É importante lembrar que o código do projeto está no GitHub e pode ser acessado [aqui](#)¹⁷.

14.6 - Conclusão

Esse capítulo tratou do DynamoDB, um serviço de banco de dados não-relacionais da AWS. O exemplo adotado mostrou a integração do consumidor de eventos de produtos provenientes da fila SQS criada no capítulo anterior, que persistia tais eventos em uma tabela do DynamoDB.

¹⁷https://github.com/siecola/aws_project02

Além disso, foi feito um novo *controller* na aplicação para expor consultas a essa tabela, mostrando inclusive operações que faziam buscas em determinados campos da tabela.

O próximo capítulo volta à aplicação `aws_project01`, para mostrar como o serviço S3 pode ser utilizado para armazenamento e tratamento de arquivos.

15 - Amazon Simple Storage Service

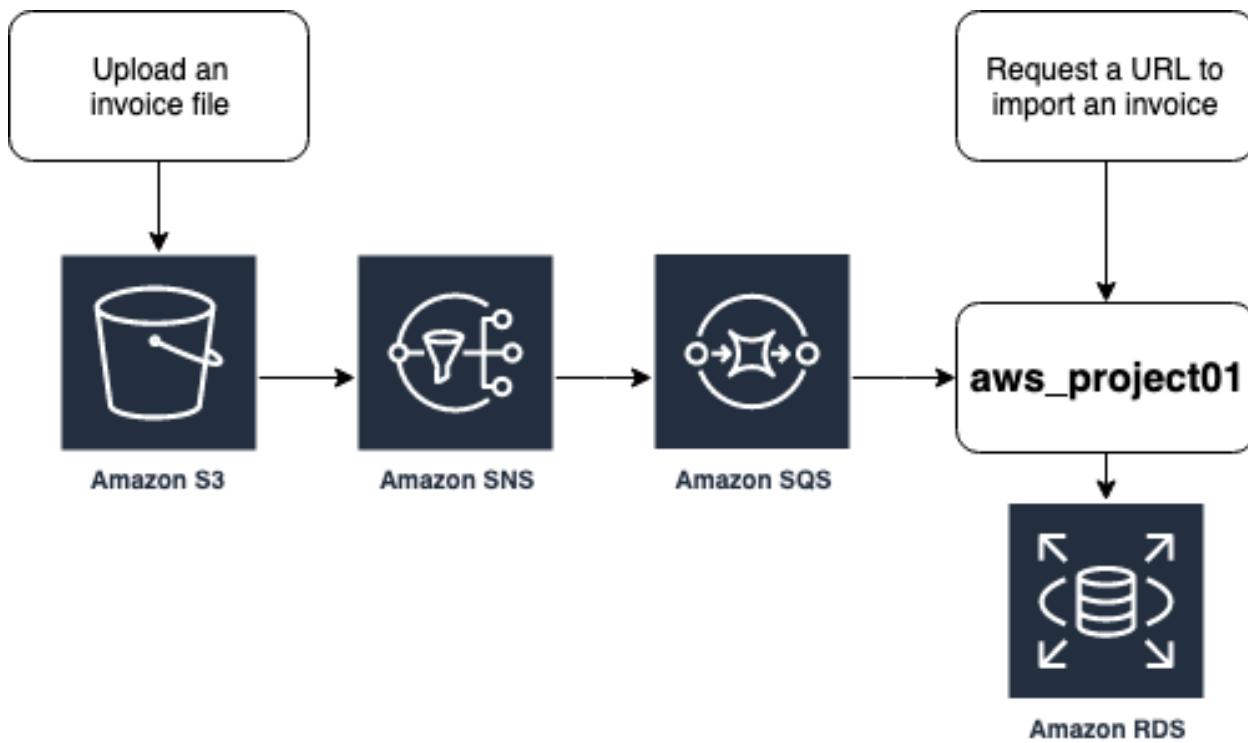
O Simple Storage Service, ou S3, da AWS, é um serviço de armazenamento, backup e análise de arquivos, capaz de oferecer escalabilidade e durabilidade para os dados, sem a necessidade do gerenciamento de servidores ou discos para tal armazenamento.

Com o S3 é possível gerar URLs pré-assinadas para que serviços de fora da infraestrutura da AWS possam acessar os arquivos. Tais URLs podem inclusive ter validades temporárias.

Dentro do S3, é possível criar *buckets* para armazenamento de qualquer tipo de arquivo, que nesse caso é chamado de objeto. Esses objetos ficam armazenados em mais de uma zona de disponibilidade em uma região, além de poderem ser replicados para outras regiões.

O S3 também pode notificar outros serviços da AWS, como SNS ou Lambda, sobre eventos em um *bucket*, como a inserção ou alteração de um objeto.

O intuito deste capítulo é voltar ao projeto `aws_projeto01` e criar um novo serviço de processamento de notas fiscais (ou **invoice**, do Inglês). O *upload* dos arquivos das notas fiscais seria feito por um sistema externo, que apenas possui uma URL para subir o arquivo ao S3. Dessa forma, a aplicação `aws_project01` seria notificada, através de uma mensagem em um novo tópico no SNS, que por sua vez publicaria uma mensagem em uma nova fila do SQS, para ser consumida pela aplicação, como ilustra a figura a seguir:



Importando arquivos com o S3

A URL para o *upload* do arquivo no S3 será gerada pelo aplicação `aws_project01`, mediante uma requisição para iniciar o processo de importação do arquivo de nota fiscal pela aplicação externa.

Quando o arquivo for inserido no S3 a aplicação `aws_project01` receberá uma notificação pela fila de tal arquivo. Nesse momento ela irá baixar o arquivo do S3, interpretá-lo, salvar seus dados em uma nova tabela do banco de dados e em seguida excluir o arquivo do S3.

Todo esse processo que será detalhado nesse capítulo é apenas uma das funcionalidades que o S3 possui, mas que já demonstra seu poder e versatilidade na manipulação de objetos.

15.1 - Criando a infraestrutura

Como pode ser observado na figura da seção anterior, é necessário criar uma pequena infraestrutura na AWS para que o exemplo proposto possa exercitado. Essa seção detalha os passos para essa preparação.

15.1.1 - Criando o SNS

Para que o S3 possa notificar a aplicação `aws_project01` é necessário criar um tópico novo no SNS. Para isso, acesse o console do SNS e clique no menu **Topics**.

Na tela que aparecer, clique no botão **Create topic**, localizado no canto superior direito da tela.

Na tela de criação do novo tópico, configure apenas seu nome: `s3-invoice-events`. Finalize clicando no botão `Create topic`, localizado na parte inferior da tela.

Esse tópico será configurado como fonte de eventos no *bucket* que será criado no S3 mais adiante.

15.1.2 - Criando o SQS

Para que a aplicação possa receber as mensagens do S3 através do SNS, é necessário criar uma fila no SQS. O intuito da fila é fazer com que a aplicação trate todos os eventos ao seu tempo, permitindo que eles possam ficar armazenados na fila para serem processados quando a aplicação `aws_project01` puder. Isso faz com que nenhum evento seja perdido.

Para começar, vá no console do SQS. Nessa tela, clique no botão `Create New Queue` para iniciar o processo.

Na tela de criação de uma nova fila, configure seu nome como `s3-invoice-events`, escolha o tipo `Standard Queue` e clique no botão `Quick-Create Queue`.

Depois da fila criada, inscreva no tópico `s3-invoice-events`, conforme descrito na seção 13.5.2 do capítulo 13. Lembre-se também de configurar a permissão da fila para que o tópico possa publicar mensagens nela, como também foi feito nessa mesma seção 13.5.2.

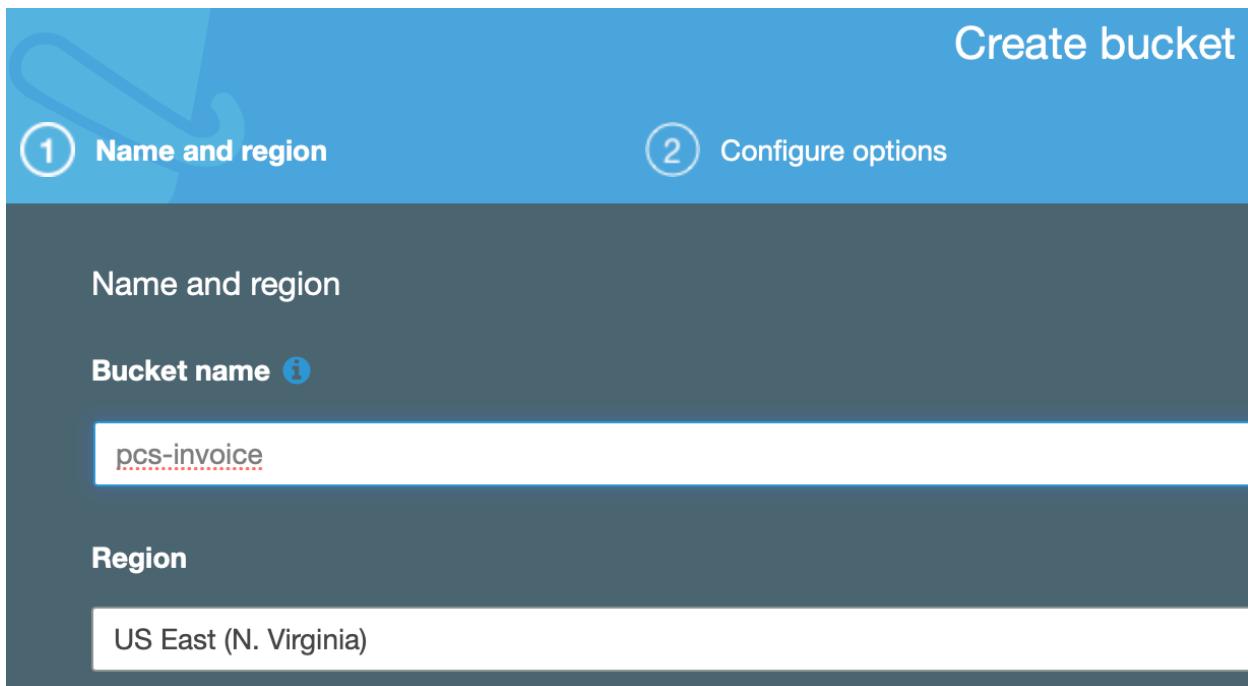
Para garantir a segurança do consumidor de eventos do *bucket* do S3, dentro da aplicação `aws_project01`, crie também uma *dead letter queue* para a fila `s3-invoice-events`, assim como foi descrito na seção 13.5.4 do capítulo 13. Dessa forma, caso algum evento ou arquivo não consiga ser tratado, a mensagem será redirecionado para essa DLQ.

15.1.3 - Criando o bucket S3

Para finalizar a parte da infraestrutura proposta nesse capítulo, vá até o console do S3 para criar um novo *bucket*.

No console do S3, clique no botão `Create bucket` para começar o processo de criação de um novo *bucket*.

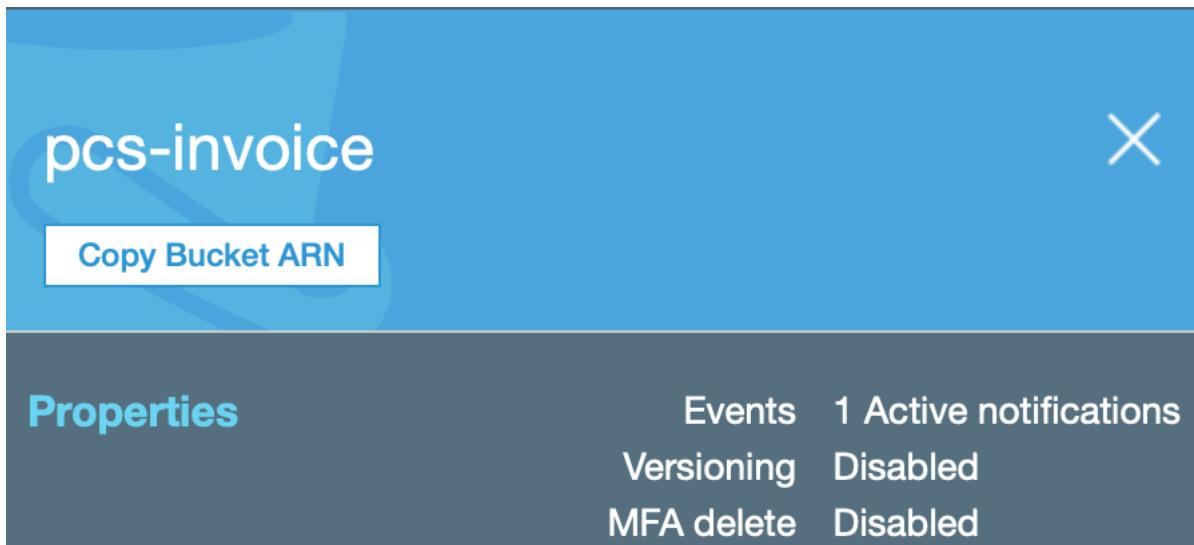
Na primeira tela de criação do *bucket*, configure seu nome para um valor único dentre todos os *buckets* do S3. Por exemplo, coloque as iniciais do seu nome seguido de `invoice` e escolha a região, que nesse livro está sendo adotado `US East (N. Virginia)`, como mostra a figura a seguir:



Configurando o nome do bucket

Para finalizar, clique no botão **Create**, localizado no canto inferior esquerdo dessa tela.

O novo *bucket* deverá aparecer na lista do S3. Marque o que acabou de ser criado e clique no botão **Copy Bucket ARN** no popup que aparecer, como mostra a figura a seguir:



Copiando o ARN do bucket

O ARN do *bucket* será necessário para configurar a política de acesso do SNS no qual ele irá publicar os eventos. Para isso, volte na tela de configuração do tópico `s3-invoice-events` e clique no botão **Edit**, localizado no canto superior direito da tela.

Na tela de edição do tópico, expanda a seção Access policy e edite o JSON com as regras da política de acesso, alterando a seção Condition para ficar como o trecho a seguir:

```
"Condition": {  
    "ArnEquals": {  
        "aws:SourceArn": "arn:aws:s3:::pcs-invoice"  
    }  
}
```

No valor do atributo aws:SourceArn, copie o ARN do *bucket* que acabou de ser criado. Para finalizar essa parte, clique no botão Save changes, localizado no canto inferior direito da página.

Isso fará com que o *bucket* criado tenha permissão de publicar eventos nesse tópico.

Depois que o *bucket* for criado, clique em seu nome para abrir suas configurações. Na tela que abrir, vá até a aba Properties e em seguida na seção Events. Nessa tela é possível configurar quais os eventos serão gerados quando, por exemplo, um novo objeto for criado no *bucket*.

Nessa tela, configure o nome do evento para invoice-events, selecione a o evento PUT e escolha o tópico SNS s3-invoice-events criado anteriormente, como mostra a figura a seguir:

Events

[+ Add notification](#) [Delete](#) [Edit](#)

Name	Events	Filter	Type
New event			X

Name *i*

Events *i*

<input checked="" type="checkbox"/> PUT	<input type="checkbox"/> Permanently deleted
<input type="checkbox"/> POST	<input type="checkbox"/> Delete marker created
<input type="checkbox"/> COPY	<input type="checkbox"/> All object delete events
<input type="checkbox"/> Multipart upload completed	<input type="checkbox"/> Restore initiated
<input type="checkbox"/> All object create events	<input type="checkbox"/> Restore completed
<input type="checkbox"/> Object in RRS lost	

Prefix *i*

Suffix *i*

Send to *i*

SNS

Essas configurações farão com que um evento seja publicado no tópico `s3-invoice-events` toda vez que um novo objeto for inserido no *bucket*.

Para finalizar, clique no botão Save.

Veja como deve ficar a configuração de eventos do *bucket*:

The screenshot shows the 'Events' configuration page for a bucket. At the top, there's a blue header bar with the word 'Events'. Below it, there are three buttons: '+ Add notification', 'Delete', and 'Edit'. The main area is a table with four columns: 'Name', 'Events', 'Filter', and 'Type'. A single row is present in the table, representing a notification named 'invoice-events' that triggers on 'PUT' events, filtered by 'SNS'. At the bottom left, there's a message '1 Active notifications'. On the right side, there are two buttons: 'Cancel' and 'Save'.

Name	Events	Filter	Type
invoice-events	PUT		SNS

1 Active notifications

Cancel Save

Eventos do bucket

Agora é necessário configura o projeto para poder acessar o *bucket* criado.

15.2 - Configurando o projeto para acessar o S3 e o SQS

Essa seção detalha os passos necessários para preparação do projeto `aws_project01` para se trabalhar com o S3. Além disso, também irá prepará-lo para ler os eventos gerados pelo S3 através da nova fila que foi criada.

15.2.1 - Adicionando as dependências ao projeto

Para começar, adicione as seguintes dependências ao arquivo `build.gradle` do projeto `aws_project01`:

```
compile group: 'com.amazonaws', name: 'aws-java-sdk-s3', version: '1.11.613'
compile group: 'com.amazonaws', name: 'aws-java-sdk-sqs', version: '1.11.613'
compile group: 'com.amazonaws', name: 'amazon-sqs-java-messaging-lib', version: '1.0\.
.8'
compile group: 'org.springframework', name: 'spring-jms', version: '5.1.9.RELEASE'
```

A primeira dependência é a única necessária para o S3. As demais são para o SQS, como já foi visto no capítulo 13.

15.2.2 - Criando a classe de configuração do S3

Assim como foi feito com outros serviços, como o SQS e SNS, é necessário criar uma classe de configuração para que a aplicação `aws_project01` prepare um cliente para acessar o S3. Para isso, vá no pacote `config` desse projeto e crie uma nova classe chamada `S3Config`, como no trecho a seguir:

```
import com.amazonaws.auth.DefaultAWSCredentialsProviderChain;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3ClientBuilder;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class S3Config {

    @Value("${aws.region}")
    private String awsRegion;

    @Bean
    public AmazonS3 amazonS3Client() {
        return AmazonS3ClientBuilder.standard()
            .withRegion(awsRegion)
            .withCredentials(new DefaultAWSCredentialsProviderChain())
            .build();
    }
}
```

O único método presente nessa classe cria um *bean*, que é o cliente que permitirá com que a aplicação possa acessar o S3 para geração da URL pré-assinada, baixar o arquivo e apagá-lo depois de processá-lo.

A única propriedade que deve ser criada é o nome do *bucket* do S3 que será passado como parâmetro durante a inicialização da aplicação. Por isso, adicione a linha a seguir no arquivo `application.properties`:

```
aws.s3.bucket.invoice.name=pcs-invoice
```

É importante lembrar que o valor dessa propriedade será substituída pela variável de ambiente na configuração do *container*, portanto seu valor aqui não terá efeito.

15.2.3 - Criando a classe de configuração do SQS

Para configurar o projeto para poder ler filas do SQS, copie a classe `JmsConfig` de `aws_project02` para a pasta `config` do projeto `aws_project01`.

Essa classe configura o cliente do JMS para ler as filas do SQS.

Também é necessário adicionar uma nova propriedade, para representar a fila que será utilizada para receber os eventos do S3. Para isso adicione a seguinte linha ao arquivo `application.properties`:

```
aws.sqs.queue.invoice.events.name=s3-invoice-events
```

O valor dessa propriedade será substituída pela variável de ambiente na configuração do *container*.

15.2.4 - Atribuindo a permissão de acesso ao S3 ao papel `ecsTaskExecutionRole`

Para que a aplicação `aws_project01` tenha permissão para acessar o S3, é necessário adicionar tal permissão ao papel `ecsTaskExecutionRole`, no qual ela está sujeita. Para isso, adicione a política `AmazonS3FullAccess` nesse papel, da mesma forma como foi feito na seção 12.4 do capítulo 12.

15.3 - Criando uma URL para upload de um arquivo

Para fazer o *upload* do arquivo de notas fiscais ao S3 é necessário que uma URL seja gerada. Porém, para que esse processo seja feito de forma segura, evitando que qualquer um possa fazer um *upload* de um arquivo qualquer, é necessário gerar uma URL pré-assinada, que tem uma validade curta.

A URL gerada é devolvida para a aplicação externa, que então faz o *upload* do arquivo no S3.

15.3.1 - Criando o modelo para carregar a URL

É interessante criar um modelo de resposta para quando a aplicação externa solicitar uma nova URL para fazer o *upload* do arquivo de notas fiscais. Esse modelo será utilizado na nova operação no *controller* que será criado na seção seguinte.

Para isso, crie uma nova classe chamada `UrlResponse` no pacote `model`, como no trecho a seguir:

```

public class UrlResponse {
    private String url;
    private long expirationTime;

    //getters and setters
}

```

Nesse modelo, além da URL que deverá ser utilizada para fazer o *upload* do arquivo de notas fiscais, também é devolvido o tempo em que ela é válida, em segundos. Dessa forma a aplicação externa sabe quanto tempo tem para fazer o *upload* do arquivo.

15.3.2 - Criando um novo controller de notas fiscais

Para permitir que a aplicação externa possa solicitar uma nova URL para fazer o *upload* do arquivo com as notas fiscais para o S3, é necessário criar um novo *controller* no projeto aws_project01.

Para isso, vá no pacote *controller* e crie uma nova classe chamada *InvoiceController*, como no trecho a seguir:

```

import com.amazonaws.services.s3.AmazonS3;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/invoices")
public class InvoiceController {

    @Value("${aws.s3.bucket.invoice.name}")
    private String bucketName;

    private AmazonS3 amazonS3;
    private InvoiceRepository invoiceRepository;

    @Autowired
    public InvoiceController(AmazonS3 amazonS3,
                            InvoiceRepository invoiceRepository) {
        this.amazonS3 = amazonS3;
        this.invoiceRepository = invoiceRepository;
    }
}

```

Repare que o cliente do S3 já é injetado no construtor do *controller* para ser utilizado por ele.

A seguir, crie uma nova operação nesse *controller*, seguindo o exemplo do trecho a seguir:

```
@PostMapping  
public ResponseEntity<UrlResponse> createInvoiceUrl() {  
    UrlResponse urlResponse = new UrlResponse();  
    Instant expirationTime = Instant.now().plus(Duration.ofMinutes(5));  
    String processId = UUID.randomUUID().toString();  
  
    GeneratePresignedUrlRequest generatePresignedUrlRequest =  
        new GeneratePresignedUrlRequest(bucketName, processId)  
            .withMethod(HttpMethod.PUT)  
            .withExpiration(Date.from(expirationTime));  
    generatePresignedUrlRequest.addRequestParameter(  
        Headers.S3_USER_METADATA_PREFIX + "process-id", processId);  
  
    urlResponse.setExpirationTime(expirationTime.getEpochSecond());  
    urlResponse.setUrl(amazonS3.generatePresignedUrl(  
        generatePresignedUrlRequest).toString());  
  
    return new ResponseEntity<UrlResponse>(urlResponse, HttpStatus.OK);  
}
```

Nesse método, um objeto do tipo `GeneratePresignedUrlRequest` é criado para poder ser utilizado para a geração da URL pré-assinada para o *upload* do arquivo de notas fiscais. Ele possui as seguintes definições:

- O método a ser utilizado para fazer o *upload* foi definido como o HTTP PUT;
- O tempo de expiração da URL foi definido em 5 minutos;
- O parâmetro de nome `process-id` foi criado para armazenar uma identificação única do processo de *upload* do arquivo.

Com esse objeto é possível gerar a URL pré-assinada através do método:

```
amazonS3.generatePresignedUrl(generatePresignedUrlRequest)
```

Tendo a URL e o tempo de expiração, um objeto do tipo `UrlResponse` é montado e devolvido na operação criada. Dessa forma a aplicação externa terá a URL completa para acessar o S3 e fazer o *upload* do arquivo.

15.4 - Recebendo a notificação do S3 pela fila

Depois que a URL for criada na nova operação no novo *controller* da seção anterior, a aplicação externa irá fazer o *upload* do arquivo de notas fiscais no S3. Esse por sua vez irá publicar uma

mensagem no tópico `s3-invoice-events`, que irá copiar uma mensagem na fila do SQS de mesmo nome.

Com isso a aplicação `aws_project01` poderá consumir o evento do S3 através da fila e baixar o arquivo do S3. Porém, quando o arquivo de notas fiscais for baixado para a aplicação, ele terá que ser interpretado e seus dados armazenados em uma nova tabela do banco de dados RDS que a aplicação `aws_project01` já possui.

As notas fiscais poderão ser visualizadas em uma nova operação que será criada na seção seguinte.

O arquivo de notas fiscais, a ser utilizado nesse exemplo, pode ser um simples arquivo texto com seu conteúdo em formato JSON, como no trecho a seguir:

```
{  
    "invoiceNumber": "ABC-123",  
    "customerName": "Matilde",  
    "totalValue": 1250.00,  
    "productId": 1,  
    "quantity": 2  
}
```

Para facilitar o código a ser desenvolvido, o arquivo poderá conter apenas uma nota fiscal, cada uma com um único produto.



Obviamente, em um exemplo real, um arquivo de notas fiscais teria uma lista delas e cada uma poderia ter vários produtos. A ideia aqui é apenas demonstrar o mecanismo de importação de dados através do S3.

15.4.1 - Criando o modelo de nota fiscal

Agora que o formato do arquivo de nota fiscal já foi definido, é necessário definir a classe modelo que irá representar as entidades no banco de dados da aplicação `aws_project01`. Para isso, vá até o pacote `model` e crie uma nova classe como no trecho a seguir:

```
@Table(  
    uniqueConstraints = {  
        @UniqueConstraint(columnNames = {"invoiceNumber"})  
    }  
)  
  
@Entity  
public class Invoice {  
    @Id  
    @GeneratedValue(strategy= GenerationType.IDENTITY)  
    private long id;  
  
    @Column(length = 32, nullable = false)  
    private String invoiceNumber;  
  
    @Column(length = 32, nullable = false)  
    private String customerName;  
  
    private float totalValue;  
  
    private long productId;  
  
    private int quantity;  
  
    //getters and setters  
}
```

Aqui foi colocado uma restrição para não haver mais de uma nota fiscal com o mesmo número, através da anotação `@Table`.

Esse modelo será utilizado para interpretar o arquivo de notas fiscais, bem como para persisti-la no banco de dados da aplicação `aws_project01`.

15.4.2 - Criando o repositório de nota fiscal

Da mesma forma como foi feito com a entidade de produto, é necessário criar um repositório de dados para persistir a entidade de nota fiscal. Para isso, vá até o pacote `repository` e crie uma nova interface chamada `InvoiceRepository`, como no trecho a seguir:

```
import br.com.siecola.aws_project01.model.Invoice;
import org.springframework.data.repository.CrudRepository;

import java.util.List;
import java.util.Optional;

public interface InvoiceRepository extends CrudRepository<Invoice, Long> {

    Optional<Invoice> findByInvoiceNumber(String invoiceNumber);

    List<Invoice> findAllByCustomerName(String customerName);
}
```

Repare que já foram criados dois métodos:

- Pesquisa por uma nota fiscal através de seu número;
- Pesquisa de todas as notas fiscais de um determinado cliente.

Esse repositório será utilizado no consumidor da fila, quando a nota fiscal tiver que ser persistida, e também no *controller* criado nesse capítulo, para retornar as notas fiscais pelo seu número ou pelos nomes dos clientes.

15.4.3 - Criando o consumidor da fila

Quando o S3 publicar o evento de um novo arquivo, a aplicação deverá consumir tal evento pela fila s3-invoice-events. Para isso, crie um novo pacote no projeto aws_project01 chamado consumer. Dentro dele, crie uma nova classe chamada `InvoiceConsumer`, como no trecho a seguir:

```
@Service
public class InvoiceConsumer {
    private static final Logger log = LoggerFactory.getLogger(
        InvoiceConsumer.class);
    private ObjectMapper objectMapper;
    private InvoiceRepository invoiceRepository;
    private AmazonS3 amazonS3;

    @Autowired
    public InvoiceConsumer(ObjectMapper objectMapper,
                          InvoiceRepository invoiceRepository,
                          AmazonS3 amazonS3) {
        this.objectMapper = objectMapper;
        this.invoiceRepository = invoiceRepository;
    }
}
```

```
    this.amazonS3 = amazonS3;
}
}
```

Também será necessário criar um modelo para interpretar a mensagem que virá da fila. Felizmente seu formato é o mesmo utilizado no projeto aws_project02. Por isso copie a classe SnsMessage do pacote model desse projeto para o pacote de mesmo nome no projeto aws_project01.

Em seguida, crie o método que irá consumir a mensagem do evento do S3, pela fila SQS, como no trecho a seguir:

```
@JmsListener(destination = "${aws.sqs.queue.invoice.events.name}")
public void receiveS3Event(TextMessage textMessage)
    throws JMSException, IOException {

    SnsMessage snsMessage = objectMapper.readValue(textMessage.getText(),
        SnsMessage.class);

    S3EventNotification s3EventNotification = objectMapper
        .readValue(snsMessage.getMessage(), S3EventNotification.class);

    processInvoiceNotification(s3EventNotification);
}
```

Perceba que o evento gerado pelo S3 é encapsulado em uma mensagem do SNS. Por isso é necessário antes abrir essa mensagem para retirar o evento em si do S3, que pode ser transformado em um objeto do tipo S3EventNotification. Dentro desse objeto é que se encontram as informações para o *download* do arquivo que foi importado no S3.

15.4.4 - Baixando o arquivo de nota fiscal

Para baixar o arquivo de nota fiscal, é necessário obter algumas informações do evento do S3 e convertê-lo para String, como pode ser visto no trecho a seguir:

```

private void processInvoiceNotification(S3EventNotification
                                      s3EventNotification) throws IOException {
    for (S3EventNotification.S3EventNotificationRecord
         s3EventNotificationRecord : s3EventNotification.getRecords()) {
        S3EventNotification.S3Entity s3Entity =
            s3EventNotificationRecord.getS3();
        String bucketName = s3Entity.getBucket().getName();
        String objectKey = s3Entity.getObject().getKey();

        String invoiceFile = downloadObject(bucketName, objectKey);

        Invoice invoice = objectMapper.readValue(invoiceFile, Invoice.class);
        log.info("Invoice received: {}", invoice.getInvoiceNumber());

        invoiceRepository.save(invoice);

        amazonS3.deleteObject(bucketName, objectKey);
    }
}

```

Na realidade, a notificação do S3 pode conter vários registros, por isso o laço para iterar nessa lista. Para baixar o objeto do S3 é necessário obter o valor de sua chave e também o nome do *bucket*. Depois de baixá-lo, o conteúdo do objeto pode ser convertido em um objeto do tipo *Invoice* e esse, por sua vez, pode ser salvo na nova tabela da aplicação *aws_project01*.

Depois da nota fiscal ter sido salva, o objeto pode ser apagado do *bucket*, para que não fique ocupando espaço, embora esse passo seja opcional.

Para concluir, falta apenas implementar o método para realmente baixar o arquivo do S3, como pode ser visto no trecho a seguir:

```

private String downloadObject(String bucketName, String objectKey)
    throws IOException {
    S3Object s3Object = amazonS3.getObject(bucketName, objectKey);

    StringBuilder stringBuilder = new StringBuilder();
    BufferedReader bufferedReader = new BufferedReader(
        new InputStreamReader(s3Object.getObjectContent()));
    String content = null;
    while ((content = bufferedReader.readLine()) != null) {
        stringBuilder.append(content);
    }
    return stringBuilder.toString();
}

```

Veja que o conteúdo do objeto do S3 deve ser convertido para String para que possa ser processado.

Essa técnica faz com que não seja necessário salvar um arquivo no disco em que a aplicação está executando.

15.5 - Visualizando as notas fiscais

Para expor as notas fiscais que forma importadas, e salvas no banco de dados, volte na classe InvoiceController e crie dois novos métodos, como no trecho a seguir:

```
@GetMapping  
public Iterable<Invoice> findAll() {  
    return invoiceRepository.findAll();  
}  
  
@GetMapping(path = "/bycustomername")  
public Iterable<Invoice> findByCustomerName(@RequestParam  
                                              String customerName) {  
    return invoiceRepository.findAllByCustomerName(customerName);  
}
```

Eles acessaram a base de dados, através do repositório de nota fiscal, retornando todos as notas salvas e também aquelas de um cliente específico.

15.6 - Publicando a nova versão da aplicação aws_project01

Agora a aplicação aws_project01 já está pronta para ser testada na AWS. Altere sua versão, no arquivo build.gradle para 0.5.0 , publique no Docker Hub e atualize a definição da tarefa task-01 para usar a nova imagem e consequentemente o serviço service-01 para utilizá-la, como já foi feito algumas vezes nos capítulos anteriores.

Também será necessário adicionar duas novas variáveis de ambiente, que são:

- O nome do *bucket* S3;
- O nome da fila que irá receber o evento do S3.

Veja como deve ficar essa sessão na configuração de variáveis de ambiente do *container* da tarefa task-01:

Environment variables

You may also designate AWS Systems Manager Parameter Store keys or ARNs using the 'valueFrom' field. ECS will inject the value into containers at run-time.

Key

AWS_REGION	Value	us-east-1
AWS_S3_BUCKET_INVOICE_NAME	Value	pcs-invoice
AWS_SNS_TOPIC_PRODUCT_EVENTS_ARN	Value	arn:aws:sns:us-east-1:666336910744:product-events
AWS_SQS_QUEUE_INVOICE_EVENTS_NAME	Value	s3-invoice-events
SPRING_DATASOURCE_PASSWORD	Value	
SPRING_DATASOURCE_URL	Value	jdbc:mariadb://aws-project01-db.c1mbcbfwmq3e.us-e
SPRING_DATASOURCE_USERNAME	Value	admin

Adicionando variáveis de ambiente

Lembre-se de colocar o nome do *bucket* S3 que foi criado nesse capítulo. O que aparece na figura anterior é apenas um nome de exemplo.

15.7 - Testando a importação de arquivos de nota fiscal:

Depois que nova versão da aplicação `aws_project01` for publicada e atualizada no serviço, é possível testar o processo de importação de arquivos de notas fiscais, construído nesse capítulo. Os passos devem ser os seguintes:

- Acessar a operação `/api/invoices` com o método HTTP POST, utilizando o Postman da aplicação `aws_project01`. Lembre-se de utilizar o endereço do DNS público, o mesmo quando a aplicação foi criada no capítulo 9;

A resposta a essa solicitação deve ser no seguinte formato:

```
{
  "url": "<URL de acesso ao S3>",
  "expirationTime": 1567361236
}
```

O parâmetro `url` da resposta contém a URL pré-assinada para importação do arquivo de nota fiscal no S3.

- Dentro do Postman, clique na URL da resposta. Uma nova janela deverá abrir para que a requisição seja feita ao S3. Altere o método da requisição para HTTP PUT;
- Crie um arquivo texto para ser importado, contendo o seguinte exemplo de nota fiscal:

```
{  
    "invoiceNumber": "ABC-123",  
    "customerName": "Matilde",  
    "totalValue": 1250.00,  
    "productId": 1,  
    "quantity": 2  
}
```

Salve esse arquivo com o nome de invoice.txt.

- De volta ao Postman, selecione a aba Body e em seguida a opção binary.
- Clique no botão Select File e selecione o arquivo de nota fiscal criado. Veja como deve ficar o Postman:

The screenshot shows the Postman interface with the following details:

- Method:** PUT
- URL:** <https://pcs-invoice.s3.amazonaws.com/613c5559-77b6-44a5>
- Body Tab:** Selected (indicated by an orange underline).
- Body Content:** A file named "invoice.txt" is selected, indicated by a yellow warning icon and the text "invoice.txt".
- Body Options:** Radio buttons for "none", "form-data", "x-www-form-urlencoded", "raw", and "binary" are shown, with "binary" being selected.

Configurando Postman para importar um arquivo no S3

Depois de configurar o Postman com os passos anteriores, clique no botão Send para enviar o arquivo ao S3.

Isso fará com que o S3 gere um evento no SNS s3-invoice-events, que por sua vez irá copiar para o SQS de mesmo nome. O consumidor da aplicação aws_project01 deverá receber essa mensagem, baixar o arquivo, interpretá-lo e persistir a nota fiscal no banco de dados.

Faça testes importando mais arquivos, com diferentes notas fiscais e acompanhe no CloudWatch a aplicação aws_project01 consumindo os eventos do S3 chegando na fila.



Para cada arquivo a ser importado, é necessário gerar uma nova URL pré-assinada.

Depois de importar várias notas fiscais, acesse os novos *endpoints* da aplicação `aws_project01` para visualizar as notas fiscais:

- `/api/invoices` com o método HTTP GET, para visualizar todas as notas fiscais:

```
[  
  {  
    "id": 1,  
    "invoiceNumber": "ABC-123",  
    "customerName": "Matilde",  
    "totalValue": 1250.0,  
    "productId": 1,  
    "quantity": 2  
  },  
  {  
    "id": 2,  
    "invoiceNumber": "XYZ-456",  
    "customerName": "Doralice",  
    "totalValue": 500.0,  
    "productId": 3,  
    "quantity": 1  
  }  
]
```

- `api/invoices/bycustomername?customerName=<customer_name>` com o método HTTP GET, para visualizar as notas fiscais de um determinado cliente.

Volte ao console do S3 e veja que os arquivos importados não estão lá, pois foram apagados pela aplicação `aws_project01`, durante o processo de tratamento desse arquivos.



É importante lembrar que o código do projeto está no GitHub e pode ser acessado [aqui¹⁸](#).

15.8 - Conclusão

Nesse capítulo foi demonstrado um processo de importação de arquivos utilizando o S3, em conjunto com o SNS e o SQS, persistindo as informações interpretadas do arquivo em uma tabela do banco de dados no RDS.

¹⁸https://github.com/siecola/aws_project02

Embora o arquivo importado tenha sido muito simples, foi possível demonstrar como construir a base para um sistema onde arquivos grandes e complexos pudessem ser utilizados, sem a necessidade de fazer com que a aplicação Java tome conta de tais arquivos, delegando todo o trabalho para o S3.

No próximo capítulo será mostrado como construir aplicações *serverless*, um assunto que está em evidência, principalmente em situações onde custos de infraestrutura devem ser reduzidos.

16 - Amazon Lambda

Executar aplicações utilizando instâncias EC2, através de *clusters* por exemplo, possui inúmeras justificativas. Porém existe também a possibilidade de executar pequenas tarefas sem a necessidade de uma grande infraestrutura para suportar uma aplicação.

Aplicações *serverless* tem chamado a atenção em arquiteturas de sistemas por possibilitar a execução de pequenas **funções**, sem a necessidade de grandes infraestruturas, e principalmente sem outras preocupações, como:

- Custos de infraestrutura ociosa;
- Escalabilidade para atender muitas requisições ao mesmo tempo;
- Facilidade de instalação de novas funções.

Com AWS Lambda, um serviço para aplicações *serverless* da AWS, é possível escrever pequenas funções em linguagens como Java e Python, para serem executadas a partir de eventos, como:

- Mensagens provenientes de um SNS;
- Alterações em uma tabela do DynamoDB;
- Requisições HTTP.

Os seguintes fatores influenciam no custo de execução de funções executadas em um Lambda:

- Tempo de execução;
- Utilização de CPU;
- Utilização de memória.

Esse capítulo demonstrará alguns exemplos construção e execução de funções Lambda, utilizando a linguagem Java. A seguir, os exemplos desse capítulo:

- Executar uma função Lambda a partir de um agendamento;
- Invocar uma função Lambda a partir de um evento no S3;
- Invocar uma função Lambda a partir de um evento publicado no SNS.

Também será mostrado como é possível acessar outros serviços da AWS de dentro de uma execução de uma função Lambda, como por exemplo escrever dados em uma tabela do DynamoDB.

Ao longo desse capítulo serão desenvolvidas algumas funções para demonstrar os conceitos de utilização de funções Lambda, bem como o processo para instalá-las através do console da AWS.

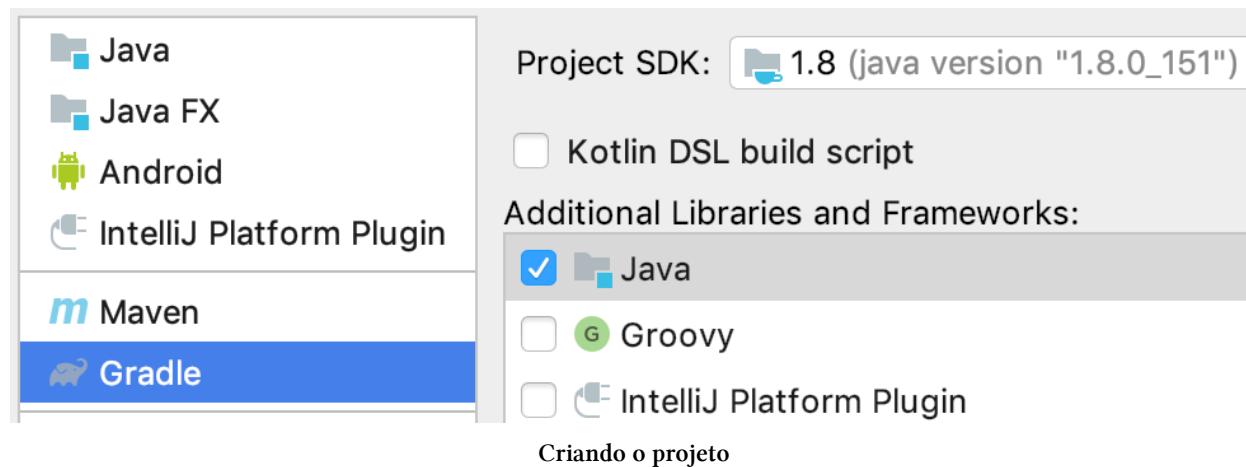
16.1 - Agendando a execução de uma função Lambda

Esse primeiro exemplo irá demonstrar como criar o projeto da função Lambda, utilizando o IntelliJ IDEA. Também irá demonstrar como a função em si, utilizando o console da AWS.

A ideia é criar uma função que seja executada a cada 2 minutos e que simplesmente imprima uma mensagem de log no CloudWatch. Obviamente, o exemplo é bem simples, mas o propósito é passar pelos passos iniciais e conceitos fundamentais de funções Lambda, desde o código do projeto, até à utilização do console da AWS.

16.1.1 - Criando o projeto para a função Lambda

Para começar a criar o projeto da função Lambda, no IntelliJ IDEA, acesse o menu **File** → **New Project**. Nessa tela, selecione a opção **Gradle** → **Java**, como mostra a figura a seguir:



Para prosseguir, clique em **Next**. Na próxima tela, configure os parâmetros como mostra a figura a seguir:

The screenshot shows the 'Details of the project' configuration screen. It has three fields: 'GroupId' with value 'br.com.siecola', 'ArtifactId' with value 'aws_lambda01', and 'Version' with value '1.0-SNAPSHOT'. At the bottom of the screen, the text 'Detalhes do projeto' is visible.

GroupId	br.com.siecola
ArtifactId	aws_lambda01
Version	1.0-SNAPSHOT

Detalhes do projeto

Esses valores são importantes, pois deverão ser configurados na tela de criação da função Lambda no console da AWS.

Para continuar, clique no botão Next. Na próxima tela, apenas marque a opção Use auto-import e finalize a criação do projeto. O IntelliJ IDEA criará um novo projeto, que a princípio é uma aplicação Java simples, mas com alterações especiais para se comportar como uma função Lambda.

Depois que o IntelliJ IDEA criar o projeto, abra o arquivo build.gradle e configure-o com o seguinte conteúdo:

```
plugins {
    id 'java'
}

group 'br.com.siecola'
version '1.0-SNAPSHOT'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    compile 'com.amazonaws:aws-lambda-java-core:1.2.0'
    compile 'com.amazonaws:aws-lambda-java-events:2.2.7'
}

task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
        from configurations.runtime
    }
}
build.dependsOn buildZip
```

A seção dependencies desse trecho configura o projeto para trabalhar com o AWS Lambda. Já a task buildZip configura uma tarefa para a criação do projeto em formato zip, para poder ser adicionado na função Lambda que será criada no console da AWS.

Para começar a criar o código dessa função Lambda, vá até a pasta src -> main -> java e crie um novo pacote com o nome br.com.siecola.aws_lambda01. Dentro desse pacote, crie uma nova classe chamada ScheduledJob, como no trecho a seguir:

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.ScheduledEvent;

public class ScheduledJob implements RequestHandler<ScheduledEvent, String> {

    @Override
    public String handleRequest(ScheduledEvent input, Context context) {
        LambdaLogger logger = context.getLogger();

        logger.log("Event source: " + input.getSource());

        return null;
    }
}
```

Repare que a classe implementa a interface RequestHandler, que nesse caso está apta a receber um evento do tipo ScheduledEvent e retornar uma String.

A interface obriga a implementação do método handleRequest, que recebe dois parâmetros:

- **ScheduledEvent**: esse parâmetro traz detalhes do evento que iniciou a função Lambda;
- **Context**: esse contém informações de contexto da execução do Lambda, como por exemplo como gerar logs.

A única função executada aqui será a geração de um log, que poderá ser visto no CloudWatch Logs.

Essa é a estrutura de uma função Lambda em Java: uma classe implementando uma interface do tipo RequestHandler. Dentro do método, a função é executada.

Embora a função criada seja bem simples, é possível demonstrar a estrutura de um projeto para a execução de uma função Lambda.

Para finalizar, abra o painel do Gradle, através do menu **View -> Tool Windows -> Gradle** e execute a tarefa localizada dentro de **Tasks -> build -> build**. Isso fará com que um arquivo no formato **zip** seja criado dentro da pasta **build -> distributions** do projeto. Esse arquivo será utilizado para a criação da função Lambda no console da AWS, ou seja, ele é o artefato que contém a função em si.

16.1.2 - Criando a função Lambda na AWS

Agora que o projeto já foi criado, é necessário criar a função Lambda no console da AWS. Para isso, abra-o e localize o serviço **Lambda**.

No console do Lambda, clique no botão `Create function` para iniciar o processo. Na primeira tela, selecione a opção `Author from scratch`, para poder configurar todos os passos necessários.

Ainda nessa tela, configure o nome da função para `scheduled-01` e escolha Java para a opção `Runtime`, como mostra a figura a seguir:

The screenshot shows the 'Basic information' section of the Lambda function creation wizard. It includes fields for 'Function name' (set to 'scheduled-01'), 'Runtime' (set to 'Java 8'), and 'Permissions' (with a link to 'Info'). A note at the bottom indicates that AWS will create an execution role with basic Lambda permissions.

Basic information

Function name
Enter a name that describes the purpose of your function.
scheduled-01
Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime Info
Choose the language to use to write your function.
Java 8

Permissions Info
Lambda will create an execution role with permission to upload your code.
▼ Choose or create an execution role

Execution role
Choose a role that defines the permissions of your function.
Create a new role with basic Lambda permissions

Criando a função Lambda

Ainda nessa tela, é possível observar que a função Lambda também deve possuir permissões de execução. Nesse caso, mantendo-se a configuração padrão, o AWS criará uma papel com permissões básicas, que dão acesso à escrita de logs no CloudWatch Logs. Mais adiante essas permissões serão alteradas para as novas funções que serão criadas.

Para finalizar, clique no botão `Create function`.

Quando a função for criada, será exibida no console da AWS. Dentro dessa tela, é possível fazer o *upload* do código fonte da função, criado na seção anterior. Isso pode ser feito na seção Function code dessa página, através do botão Upload, como pode ser observado na figura a seguir:

Function code Info

Code entry type

Upload a .zip or .jar file

Function package

 **Upload**

For files larger than 10 MB, consider

Upload do código fonte da função

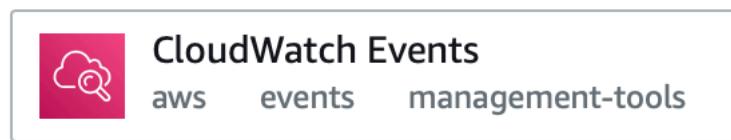
Para subir o arquivo zip gerado na seção anterior, clique no botão Upload e vá até a pasta build->distributions dentro do projeto aws_lambda01 e selecione o arquivo zip.

Ainda dentro da seção Funciton code, selecione Java para a opção Runtime. Também é necessário configurar o caminho do Handler, localizado no canto esquerdo dessa seção. Esse valor deve ser o caminho completo até a função implementada na seção anterior, ou seja: br.com.siecola.aws_lambda01.ScheduledJob::handleRequest. Ele é composto pelo pacote do projeto, juntamente com o nome da classe e o método que implementa a função.

Depois de selecionar o arquivo, clique no botão Save, localizado no canto superior direito da página do console do Lambda, para salvar as configurações até o momento.

Agora é necessário configurar o evento que irá disparar a execução dessa função. Para isso, clique no botão + Add trigger, localizado na parte central da tela, no lado esquerdo.

Nessa tela, escolha a opção CloudWatch Events como fonte do evento para disparar a função Lambda. Além disso, selecione a opção Create a new rule e configure o nome como lambda_event_01, como mostra a figura a seguir:



Rule

Pick an existing rule, or create a new one.

Create a new rule

Select or create a new rule

Rule name*

Enter a name to uniquely identify your rule.

lambda_event_01

Configurando o evento de trigger

Ainda nessa tela, selecione a opção `Schedule expression` e configure o campo com o valor `rate(2 minutes)`, como mostra a figura a seguir:

Rule type

Trigger your target based on an event pattern, or by schedule

Event pattern

Schedule expression

Schedule expression*

Self-trigger your target on an automated schedule

rate(2 minutes)

e.g. `rate(1 day)`, `cron(0 17 ? * MON-FRI *)`

Configurando o agendamento do evento

Isso fará com que um evento seja gerado pelo CloudWatch a cada 2 minutos, para invocar a execução dessa função Lambda. Repare que também é possível utilizar uma expressão cron nesse campo, para agendamentos mais sofisticados.

Para finalizar, clique no botão Add. Agora o CloudWatch possui um evento, que será disparado a cada

2 minutos, fazendo com que essa função Lambda seja executada, depois de concluir a configuração, mais adiante.

De volta à tela da função criada, é possível observar um diagrama como da figura a seguir:

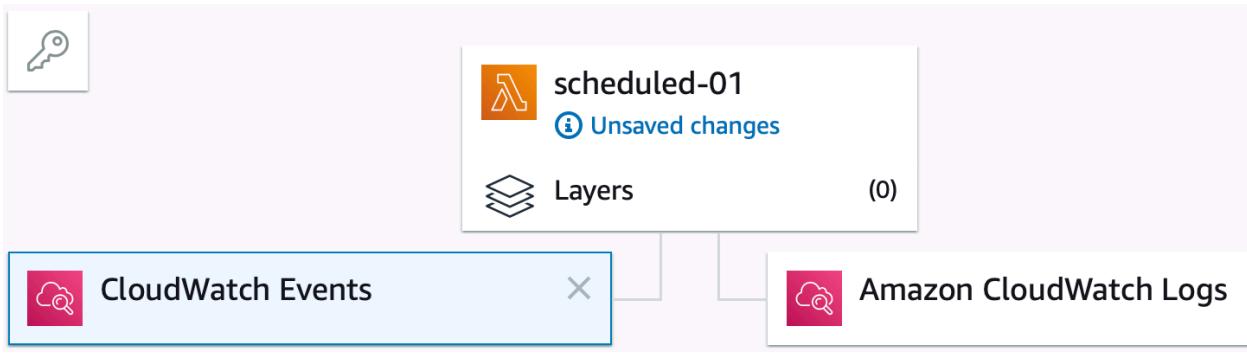


Diagrama da função

Nesse diagrama é possível observar, do lado esquerdo, a fonte de evento para iniciar a execução da função Lambda. Do lado direito, pode-se notar a saída da execução, que será um log a ser gerado no CloudWatch Logs.

Para concluir a configuração da função Lambda e iniciar sua execução, clique no botão Save novamente, localizado no canto superior direito da tela.

Para visualizar a execução da função e seus logs, vá até o console do AWS CloudWatch Logs e procure pelo Log Group com o nome de `/aws/lambda/scheduled-01`. Dentro do Log stream, será gerado um log, a cada 2 minutos, como mostrado na figura a seguir:

[CloudWatch](#) > [Log Groups](#) > [/aws/lambda/scheduled-01](#)

Filter events		
	Time (UTC +00:00)	Message
	2019-09-01	
▶	22:09:35	START RequestId: f35f55f1-6
▼	22:09:35	Event source: aws.events
Event source: aws.events		

Log de execução da função Lambda

A função será executada a cada 2 minutos, a partir do evento gerado pelo CloudWatch. Isso demonstra como é possível criar uma função para ser executada em intervalos de tempo pré-definidos, para a execução de alguma tarefa específica.

Perceba a cobrança somente será feita enquanto a função Lambda estiver em execução, ou seja, por alguns milisegundos, nesse caso. Nenhuma cobrança é feita além desse espaço de tempo, que é considerado como tempo ocioso sem a alocação de nenhum recurso da AWS. Essa é uma grande vantagem em comparação a aplicações que não são *serverless*, pois a cobrança é feita durante todo o tempo em que o recurso é alocado, mesmo não realizando nenhuma tarefa.

Para desabilitar a geração de eventos, e consequentemente a execução da função, no diagrama da função Lambda, clique no *trigger* CloudWatch Events e desmarque a opção Enabled, localizado no canto direito. Em seguida, clique novamente no botão Save, no canto superior direito.



O código fonte desse projeto pode ser encontrado [aqui](#)¹⁹.

¹⁹https://github.com/siecola/aws_lambda01

16.2 - Invocando uma função Lambda através de uma notificação SNS

Uma função Lambda também pode ser invocada a partir de uma notificação de um SNS, por exemplo, para gerar algum log em uma tabela de transações sobre algum evento publicado em um tópico.

16.2.1 - Criando o projeto para a função Lambda

Para demonstrar esse conceito, crie um novo projeto, seguindo os mesmos passos da seção 16.1.1 desse capítulo, com exceção da criação da classe da função em si. Chame esse outro projeto de aws_lambda02.

Dentro do pacote br.com.siecola.aws_lambda02 desse novo projeto crie uma classe chamada SnsEventJob, como no trecho a seguir:

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;

public class SnsEventJob implements RequestHandler<SNSEvent, String> {

    @Override
    public String handleRequest(SNSEvent input, Context context) {
        LambdaLogger logger = context.getLogger();

        for(SNSEvent.SNSRecord snsRecord : input.getRecords()) {
            logger.log("SNS message: " + snsRecord.getSNS().getMessage());
        }
        return null;
    }
}
```

Repare que agora a classe está apta a receber um evento do tipo SNSEvent, que contém uma lista de registros. Dentro de cada registro, é possível obter a mensagem do evento em si.

Nesse exemplo, a mensagem recebida através do SNS está sendo impressa em um log e será possível visualizá-la no CloudWatch Logs.

Gere o arquivo zip desse novo projeto, da mesma forma como foi feito para o aws_lambda01.

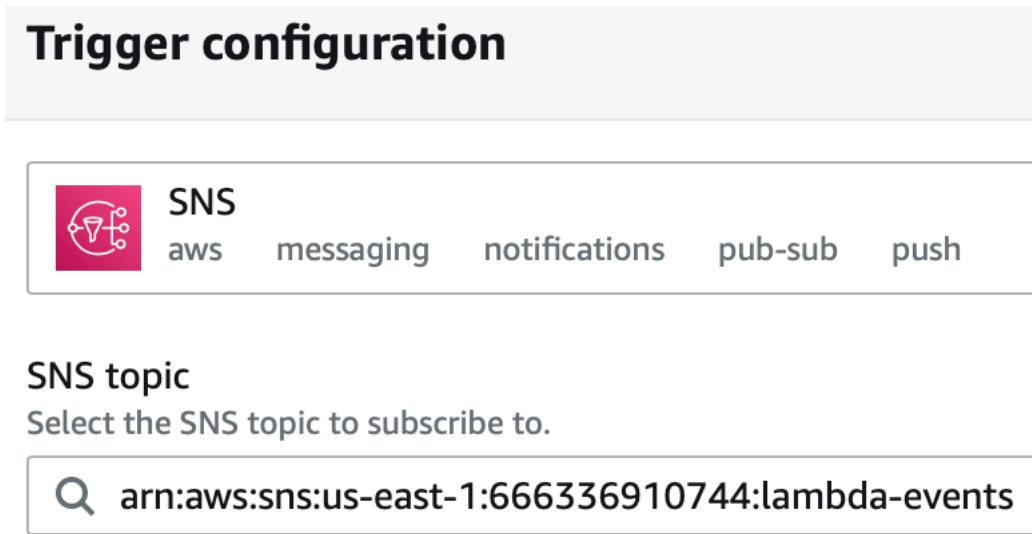
16.2.2 - Criando a função Lambda

No console do AWS Lambda, crie uma nova função chamada `aws_lambda02`, seguindo os passos descritos na seção 16.1.2 desse capítulo. Não configure o *trigger* ainda, pois para é necessário criar um tópico SNS para isso.

Faça o *upload* do arquivo `zip` do projeto, gerado na seção anterior, lembrando de configurar o parâmetro `Handler` na seção `Function code` para `br.com.siecola.aws_lambda02.SnsEventJob::handleRequest`, que é o caminho completo da função a ser executada.

Como a ideia é fazer com que essa função Lambda seja invocada a partir de um evento gerado em um tópico SNS, é necessário então criar um. Por isso, vá até o console do AWS SNS e crie um novo tópico chamado `lambda-events`.

Voltando à criação da nova função Lambda, `aws_lambda02`, clique no botão `+ Add trigger` para adicionar o tópico `lambda-events` como seu *trigger*, como mostrado na figura a seguir:



Criando o trigger a partir de um tópico SNS

Para finalizar a criação do *trigger*, clique no botão `Add`, no final da página.

A configuração da função Lambda deverá ficar como a figura a seguir:

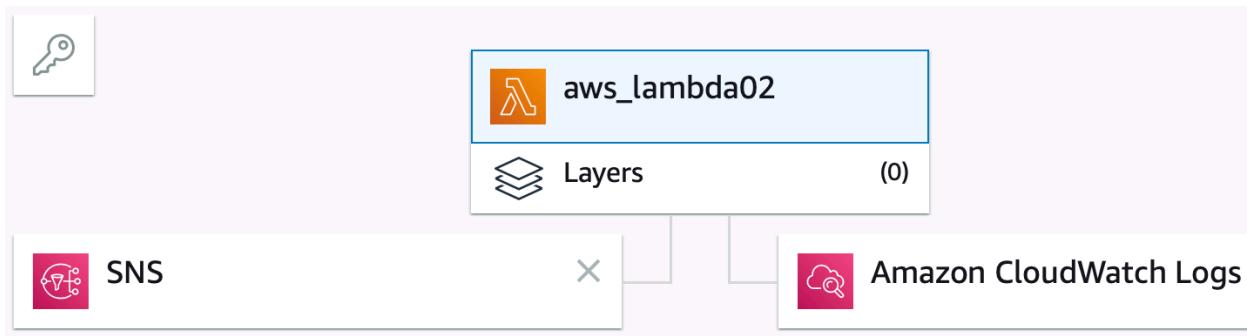


Diagrama da função Lambda

Agora que tudo já está configurado, vá até o console do SNS, no tópico criado nessa seção e publique uma nova mensagem, com um texto qualquer. Essa mensagem irá gerar um evento que irá invocar a função `aws_lambda02`, que por sua vez, irá receber tal evento e gerar uma mensagem de log no CloudWatch Logs, semelhante à figura a seguir:

[CloudWatch](#) > [Log Groups](#) > [/aws/lambda/aws_lambda02](#)

Filter events		
	Time (UTC +00:00)	Message
	2019-09-01	
▶	23:50:27	START RequestId: 170dc311-
▼	23:50:27	SNS message: Message publ
SNS message: Message published in a SNS topic		

Log da execução da função

Novamente, o exemplo foi bem simples, mas também demonstra a simplicidade de se criar uma função Lambda que é invocada a partir de um evento publicado em um tópico SNS.



O código fonte desse projeto pode ser encontrado [aqui²⁰](#).

16.3 - Invocando uma função Lambda através de uma notificação do S3

O último exemplo a ser demonstrado nesse capítulo será uma função Lambda que será invocada por um evento gerado por um *bucket* S3, quando um novo arquivo for inserido nele. Durante sua execução, ela irá baixar o arquivo, interpretá-lo e salvar seus dados em uma tabela no DynamoDB.

16.3.1 - Criando o projeto para a função Lambda

Para começar, crie um novo projeto chamado `aws_lambda03`, seguindo os mesmos passos definidos na seção 16.1.1 desse capítulo. Porém, será necessário alterar as versões das dependências no arquivo `build.gradle`, conforme o trecho a seguir:

```
dependencies {
    compile 'com.amazonaws:aws-lambda-java-core:1.1.0'
    compile 'com.amazonaws:aws-lambda-java-events:1.3.0'
}
```

Dentro do projeto, crie um novo pacote chamado `br.com.siecola.aws_lambda03` e dentro dele uma classe chamada `S3EventJob`, como no trecho a seguir:

```
public class S3EventJob implements RequestHandler<S3Event, String> {

    @Override
    public String handleRequest(S3Event input, Context context) {
        LambdaLogger logger = context.getLogger();

        logger.log("Starting execution...");

        return null;
    }
}
```

Crie também uma classe chamada `Invoice`, dentro do mesmo pacote, para representar a nota fiscal que será enviada através do mesmo arquivo utilizado no exemplo do capítulo anterior:

²⁰https://github.com/siecola/aws_lambda02

```

public class Invoice {
    private String invoiceNumber;
    private String customerName;
    private float totalValue;
    private long productId;
    private int quantity;

    //getters and setters
}

```

Voltando para a classe S3EventJob, crie o método privado para salvar a nota fiscal, que será extraída do arquivo que será importado no S3, como no trecho a seguir:

```

private void saveInvoice(Invoice invoice) {
    final AmazonDynamoDBClient ddbClient = new AmazonDynamoDBClient(
        new EnvironmentVariableCredentialsProvider());
    ddbClient.withRegion(Regions.US_EAST_1);
    DynamoDB dynamoDB = new DynamoDB(ddbClient);
    Table table = dynamoDB.getTable("invoice-lambda");

    table.putItem(new PutItemSpec().withItem(new Item()
        .withString("id", UUID.randomUUID().toString())
        .withString("invoiceNumber", invoice.getInvoiceNumber())
        .withString("customerName", invoice.getCustomerName())
        .withFloat("totalValue", invoice.getTotalValue())
        .withLong("productId", invoice.getProductId())
        .withInt("quantity", invoice.getQuantity())));
}

```

Como o Spring Data não pode ser utilizado aqui, a solução é criar a entidade que será persistida na tabela do DynamoDB de forma manual, configurando cada campo com seu valor. Da mesma forma, o cliente para acessar o DynamoDB é instanciado e configurado em toda execução da função Lambda.



Embora essa abordagem de criar o cliente do DynamoDB, toda vez que a função for executada, pareça estranha, é importante lembrar que o único contexto de execução, do ponto de vista da função, é presente somente durante sua execução em si. Isso significa que não é possível criar um *bean*, para ser utilizado por todas as execuções.

O nome da tabela a ser acessada é configurado manualmente, na instrução `dynamoDB.getTable("invoice-lambda")`, assim como as demais configurações do cliente de acesso ao DynamoDB.

Crie também um método para fazer o *download* do arquivo do S3, como no trecho a seguir:

```

private String downloadObject(String bucketName, String objectKey)
    throws IOException {
    final AmazonS3Client amazonS3Client = new AmazonS3Client(
        new EnvironmentVariableCredentialsProvider());
    amazonS3Client.withRegion(Regions.US_EAST_1);

    S3Object s3Object = amazonS3Client.getObject(bucketName, objectKey);

    StringBuilder stringBuilder = new StringBuilder();
    BufferedReader bufferedReader = new BufferedReader(
        new InputStreamReader(s3Object.getObjectContent()));
    String content = null;
    while ((content = bufferedReader.readLine()) != null) {
        stringBuilder.append(content);
    }
    return stringBuilder.toString();
}

```

Esse trecho de código é bem semelhante ao que foi utilizado no capítulo anterior para fazer o *download* do arquivo. Repare que o cliente de acesso ao S3 também é criado manualmente, assim como foi feito com o cliente do DynamoDB.

Agora volte ao método `handleRequest` para completar sua lógica para tratar o evento do S3, de forma muito semelhante como foi feito no consumidor do exemplo do capítulo anterior:

```

@Override
public String handleRequest(S3Event input, Context context) {
    LambdaLogger logger = context.getLogger();

    logger.log("Starting execution...");
    for (S3EventNotification.S3EventNotificationRecord record :
        input.getRecords()) {

        String key = record.getS3().getObject().getKey();
        logger.log("Object key: " + key);

        try {
            String objectContent = downloadObject(record.getS3().getBucket()
                .getName(), key);
            ObjectMapper objectMapper = new ObjectMapper();

            Invoice invoice = objectMapper.readValue(objectContent,
                Invoice.class);
        }
    }
}

```

```
        saveInvoice(invoice);
        logger.log("Invoice saved");
    } catch (IOException e) {
        logger.log("Failed to download object - " + e.getMessage());
    }
}
return null;
}
```

Nesse caso, os registros são lidos do evento do S3, os arquivos são baixados e interpretados. Cada nota fiscal extraída de cada arquivo da notificação do S3 é persistida na tabela do Dynamo, utilizando os métodos privados criados anteriormente.

Agora que o projeto foi construído, crie o arquivo **zip**, como foi feito nos exemplos anteriores desse capítulo, para mais adiante subir o código para a função Lambda que será criada no AWS.



O código fonte desse projeto pode ser encontrado [aqui²¹](#).

16.3.2 - Criando a função Lambda

A criação dessa função Lambda no console do AWS será um pouco diferente das demais, pois ela irá interagir com outros serviços, com o S3 e o DynamoDB. Isso fará com que as configurações de permissões seja mais elaborada. Além disso, será necessário criar um novo *bucket* no S3, assim como uma nova tabela no DynamoDB.

16.3.3 - Criando a tabela no DynamoDB

Para criar a tabela que será utilizada nesse exemplo, acesse o console do DynamoDB e crie uma nova, chamada `invoice-lambda`, seguindo os mesmos passos descritos na seção 14.1 do capítulo 14.

É nessa tabela que as notas fiscais importadas pela função Lambda serão salvas.

16.3.4 - Criando o bucket no S3

Para criar o *bucket* onde os arquivos de notas fiscais serão colocados, vá até o console do S3 e crie um novo com o nome `<prefixo>-invoice-lambda`, substituindo o prefixo por algo que possa identificá-lo de forma única. Lembre-se de escolher a região US East (N. Virginia), que onde todos os recursos utilizados ao longo desse livro estão sendo criados.

Depois de configurado o nome e a região, clique no botão `Create`, localizado no canto inferior esquerdo da página, para iniciar o processo de criação do *bucket*.

²¹https://github.com/siecola/aws_lambda03

16.3.5 - Criando a função Lambda

Agora que todos os recursos adicionais foram criados para a execução dessa função Lambda, é necessário criá-la de fato. Para isso, vá até seu console na AWS e inicio processo de criação de uma nova função.

Na primeira parte da configuração da nova função Lambda, configure seu nome como `aws_lambda03` e escolha o Runtime como Java, como mostra a figura a seguir:

Create function [Info](#)

Choose one of the following options to create your function.

Author from scratch

Start with a simple Hello World example.



Basic information

Function name

Enter a name that describes the purpose of your function.

aws_lambda03

Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime [Info](#)

Choose the language to use to write your function.

Java 8

Nome da função Lambda

Na sessão Permissions dessa mesma tela, escolha a opção Create a new role from AWS policy templates para configurar um novo papel para essa função Lambda poder acessar o S3 e o DynamoDB.

Em seguida, digite um nome para o novo papel, por exemplo `aws_lambda03_permissions` e adicione os seguintes políticas a esse papel:

- Amazon S3 object read-only permissions
- Simple microservice permissions

Veja como deve ficar essa configuração, na figura a seguir:

Permissions Info

Lambda will create an execution role with permission to upload logs to Amazon CloudWatch Logs. You can

▼ Choose or create an execution role

Execution role

Choose a role that defines the permissions of your function. To create a custom role, go to the [IAM console](#).

- Create a new role with basic Lambda permissions
- Use an existing role
- Create a new role from AWS policy templates

 Role creation might take a few minutes. The new role will be scoped to the current function.

Role name

Enter a name for your new role.

`aws_lambda03_permissions`

Use only letters, numbers, hyphens, or underscores with no spaces.

Policy templates - *optional* Info

Choose one or more policy templates.

Amazon S3 object read-only permissions 
S3

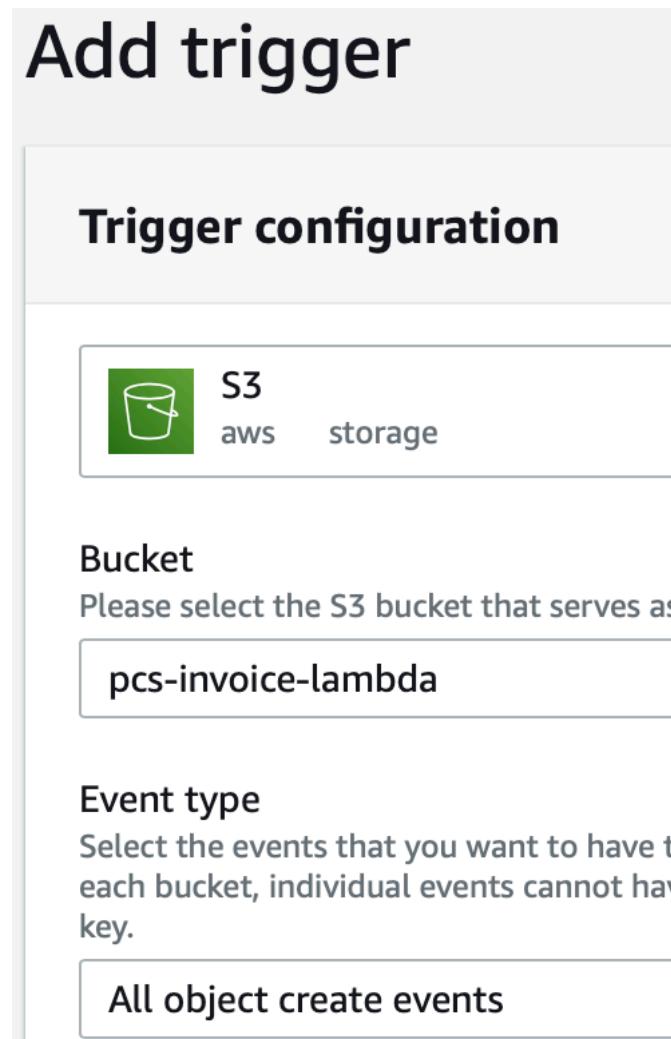
Simple microservice permissions 
DynamoDB

Permissões da função Lambda

Essas duas políticas farão com que a função tenha permissão de acessar o S3 e o DynamoDB.

Para finalizar essa sessão, clique no botão `Create function`, localizado no canto inferior direito.

Depois que a função for criada, configure seu *trigger* clicando no botão `+ Add trigger`. Nessa tela, configure a fonte do *trigger* para o S3, escolhendo o *bucket* criado na seção anterior. Além disso, escolha o tipo de evento como `All object create events`, como mostra a figura a seguir:



Dessa forma, toda vez que um novo objeto for inserido no *bucket*, a função Lambda será executada.

Para finalizar, clique no botão `Add`.

Veja como deve ficar o diagrama da função:

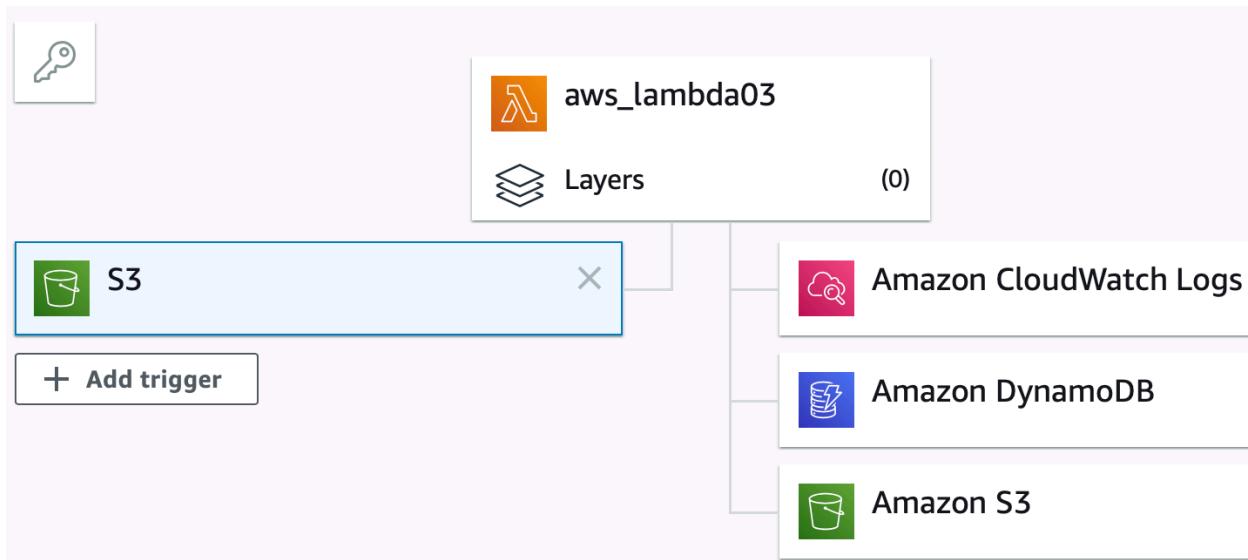


Diagrama da função

Perceba que as permissões adicionadas ao papel criado para essa função aparecem do lado direito, juntamente com a padrão, que é a escrita de logs no CloudWatch Logs.



O S3 será configurado automaticamente para conectar seu evento de novo arquivo e chamar essa função Lambda. Por isso não é necessário realizar nenhuma configuração adicional ao *bucket* criado.

Para finalizar, na seção *Function code*, escolha o arquivo **zip** gerado na seção 16.3.1 para fazer o *upload* para a função. Lembre-se também de configurar o *handler* da função com o valor `br.com.siecola.aws_lambda03.S3EventJob::handleRequest`.

Com essas últimas configurações, clique no botão *Save* localizado no canto superior direito para finalizar a criação da função e colocá-la para funcionar.

16.3.6 - Testando a função Lambda

Para testar essa nova função, pegue o mesmo arquivo de nota fiscal utilizado no capítulo 15 e acesse o *bucket* S3 criado nesse capítulo, clicando em seu link, na lista de *buckets*. Na tela que aparecer, clique no botão *Upload*, escolha o arquivo de nota fiscal e clique no botão *Upload*.

Essa ação fará com que a função Lambda seja iniciada. Ela por sua vez irá tratar o evento do S3, baixar o arquivo, interpretá-lo e salvar a nota fiscal na tabela `invoice-lambda` do DynamoDB.

Gere outros arquivos e repita o processo. As notas fiscais importadas poderão ser vistas na tabela do DynamoDB, como mostra a figura a seguir:

	id ⓘ	customerName	invoiceNumber	productId	quantity	totalValue
<input type="checkbox"/>	5b7a4cea-8284-4ef	Doralice	XYZ-456	3	1	500
<input type="checkbox"/>	638411ac-0c4e-462	Matilde	ABC-123	2	3	1250

Notas fiscais importadas

Também é possível monitorar o log de execução da função Lambda através do CloudWatch Logs, como mostrado na figura a seguir:

[CloudWatch](#) > [Log Groups](#) > [/aws/lambda/aws_lambda03](#) > 2019/09/04/[LATEST]243e

Filter events		
	Time (UTC +00:00)	Message
	2019-09-04	
▶	00:56:41	START RequestId: e780ae33-846c-4b45-951c-2bf0e7657b9
▶	00:56:42	Starting execution...
▶	00:56:42	Object key: invoice.txt
▶	00:56:50	Invoice saved
▶	00:56:50	END RequestId: e780ae33-846c-4b45-951c-2bf0e7657b9d

Logs de execução da função

É importante ressaltar que nessa abordagem, não existem recursos alocados enquanto uma nota fiscal não é importada, ou seja, não existe cobrança durante o tempo ocioso da função Lambda.

16.4 - Monitorando a execução de funções Lambdas

Logo abaixo do nome da função Lambda existe uma aba chamada Monitoring. Nessa seção é possível ver gráficos como:

- Número de invocações da função ao longo do tempo;

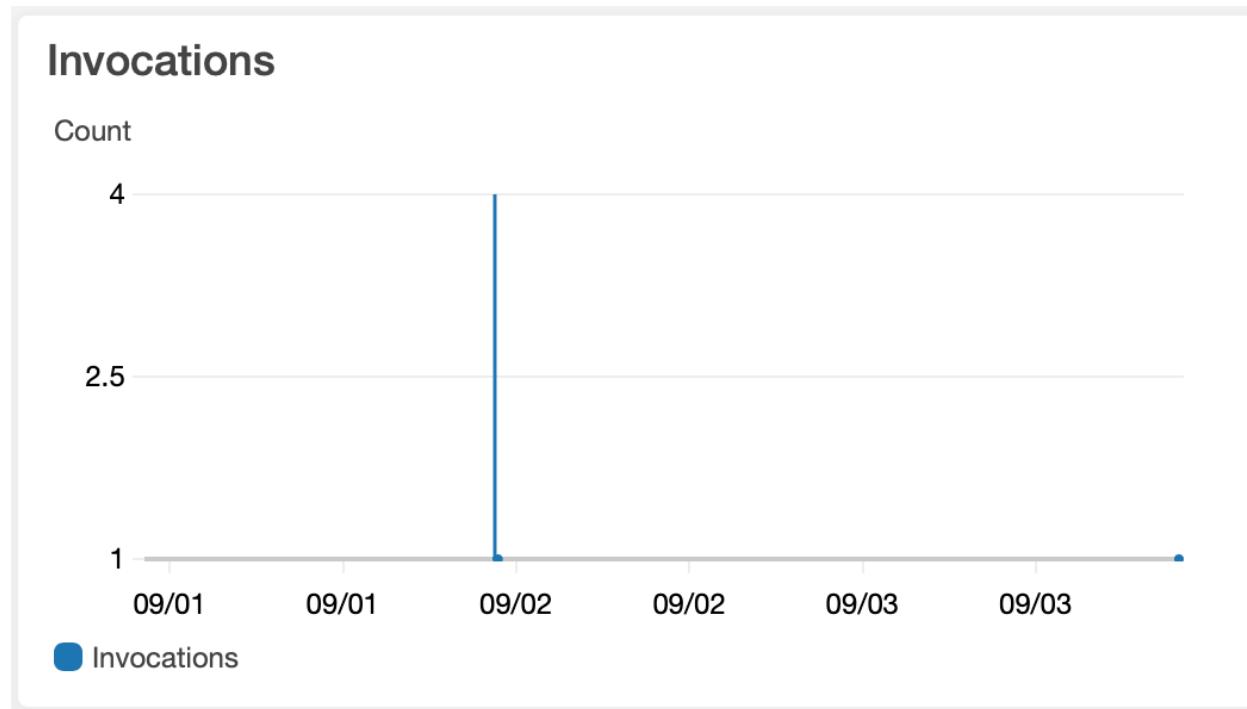


Gráfico de invocações da função

- Duração de execução das invocações.

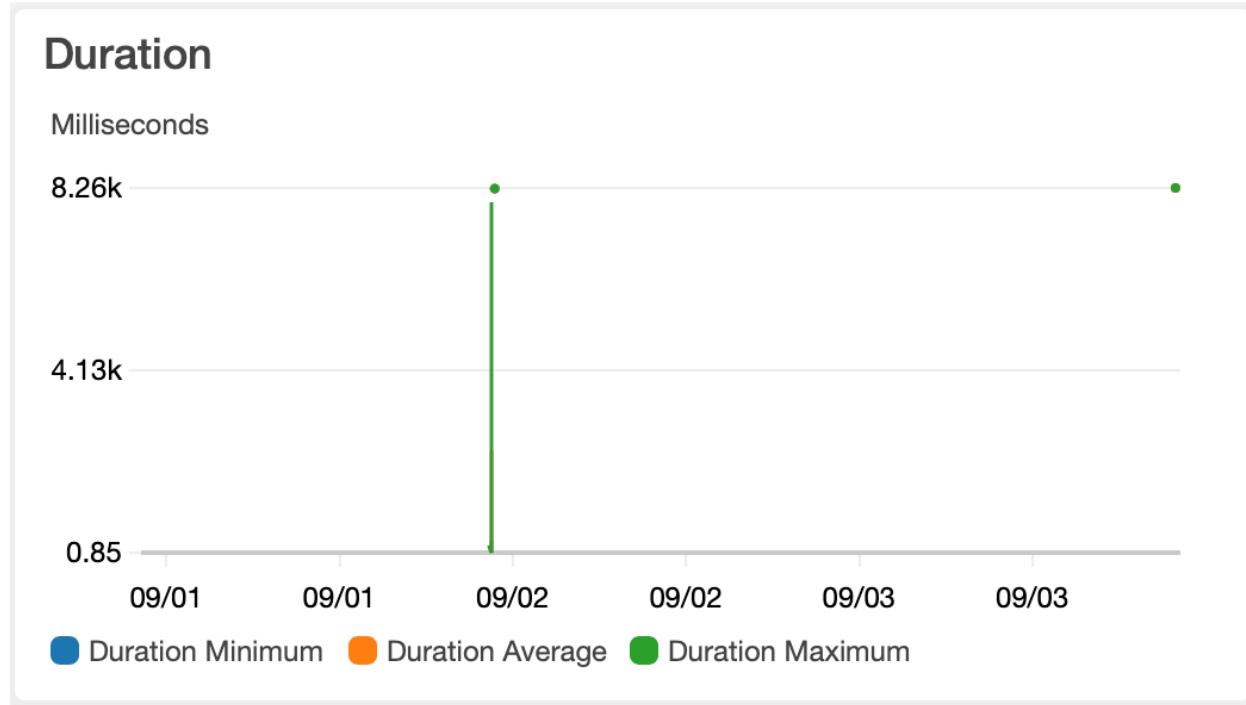


Gráfico com tempo de execuções

Além disso é possível disso é possível acessar diretamente os logs de cada execução, na seção CloudWatch Logs Insights, bem como visualizar o tempo gasto em cada uma e seu consumo de memória, como pode ser visto na figura a seguir:

: DurationInMS	: BilledDurationInMS	: MemorySetInMB	: MemoryUsedInMB
8263.91	8300	512	140
8248.76	8300	512	141

Execuções de uma função

Essas informações são muito úteis para encontrar possíveis problemas ou monitorar a função em tempo de processamento e consumo de memória.

16.5 - Conclusão

Nesse capítulo foi possível demonstrar 3 tipos de formas de invocar Lambdas, além de desenvolver uma função capaz de gravar dados em uma tabela DynamoDB.

A utilização de funções Lambda, como forma de desenvolver aplicações *serverless* é um assunto que está muito em evidência, principalmente pela facilidade de implementação, configuração e monitoramento, graças às ferramentas fornecidas pela AWS e outros *players* de *cloud computing*.