

Google App Engine

Construindo serviços na nuvem

Edição atualizada



Casa do
Código

PAULO SIÉCOLA

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Vivian Matsui

Carlos Felício

[2020]

Casa do Código

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-5519-159-6

EPUB: 978-85-5519-160-2

MOBI: 978-85-5519-161-9

Caso você deseje submeter alguma errata ou sugestão, acesse
<http://erratas.casadocodigo.com.br>.

AGRADECIMENTOS

Gostaria de agradecer ao Adriano Almeida, pela oportunidade de publicar um livro na Casa do Código, e também agradecer à Vivian Matsui e à Bianca Hubert, pelo empenho e dedicação nas revisões didáticas.

Agradeço aos meus professores e mestres, pois, sem eles, não poderia compartilhar este conhecimento.

Agradeço aos meus pais, que são a fonte da minha motivação para o trabalho.

Obrigado a meu Deus, pelos dons que de Ti recebi. Espero estar utilizando-os com sabedoria e equilíbrio.

Finalmente, agradeço à minha amada esposa Isabel, pela paciência, apoio e incentivo sem igual.

SOBRE O LIVRO

O **Google App Engine** é uma plataforma de computação nas nuvens que permite a execução de aplicações Web na **infraestrutura do Google**. Tudo isso de forma fácil e escalável, sem a necessidade de manutenção em sistemas operacionais ou servidores.

Ele possui várias opções de utilização **gratuitas**, baseadas em cotas e limites, que permitem o desenvolvimento de pequenas aplicações para testes e estudos sem gastar nenhum centavo! Isso torna a plataforma muito atrativa se você está começando e deseja aprender mais sobre ela.

Este livro aborda vários aspectos dessa plataforma, principalmente aqueles relativos ao desenvolvimento de aplicações em Java para interagir com seus recursos, e também a administração deles através das ferramentas disponibilizadas pelo Google App Engine, ou GAE, como é comumente chamado.

Os principais tópicos deste livro são:

- Conceitos básicos do Google App Engine;
- Como desenvolver e hospedar **serviços REST** com Jersey no GAE;
- Como trabalhar com o Google **Datastore**;
 - Entidades
 - Consultas
 - Administração
- Como desenvolver uma aplicação no GAE utilizando o **Firebase Cloud Messaging** para enviar mensagens a

- aplicativos móveis;
- Como **agendar tarefas** no GAE para invocar um serviço em sua aplicação;
 - Como utilizar OAuth, como mecanismo de autenticação de usuários, para proteger o acesso aos serviços da sua aplicação;
 - Como gerar, visualizar e gerenciar os logs das aplicações Java hospedadas no GAE;
 - Como trabalhar com *memory cache* para armazenar dados temporários em memória de forma rápida, mas não persistente;
 - Como visualizar e gerenciar os erros gerados pela aplicação.

Ao longo da leitura, alguns projetos Java serão criados para o GAE. O código-fonte deles está no GitHub, no seguinte endereço: <https://github.com/siecola/GAEBookV3Exemplo1>.

Para o desenvolvimento desses projetos, você poderá usar a IDE **IntelliJ IDEA Community Edition**, juntamente com algumas ferramentas fornecidas pelo Google. Há um capítulo dedicado à preparação do ambiente de desenvolvimento.

A quem se destina este livro

Este livro é útil para desenvolvedores de aplicações Web que desejam conhecer sobre a plataforma de computação nas nuvens Google App Engine. Será possível aprender a trabalhar com suas tecnologias, ferramentas e técnicas para construir sistemas arquitetados para serem escaláveis. Para administradores de sistema, este livro traz tópicos essenciais para aqueles que desejam

administrar aplicações que serão hospedadas no Google App Engine, pois há uma boa parte do conteúdo dedicado a isso.

É interessante que o leitor possua familiaridade com **Java e Programação Orientada a Objetos**, bem como com a IDE IntelliJ IDEA Community Edition, para poder aproveitar com mais intensidade o material apresentado, e se aventurar nos exercícios propostos. Porém, o livro aborda todos os conteúdos de forma didática, construindo os **exemplos desde o início** e detalhando os conceitos de modo que possam ser compreendidos por programadores com qualquer nível de experiência.

SOBRE O AUTOR

Paulo César Siécola é Mestre em Ciência da Computação pelo Instituto de Matemática e Estatística da Universidade de São Paulo (2011). Possui graduação em Engenharia Elétrica pelo Instituto Nacional de Telecomunicações – INATEL (2005).

Atualmente, é Team Leader na WatchGuard Brasil e professor em cursos de Pós-Graduação no INATEL. Tem experiência em desenvolvimento de software em **C, Java, Swift e C#**, atuando principalmente nos seguintes temas: desenvolvimento Web, sistemas embarcados e desenvolvimento de aplicativos móveis para Android e iOS.

Sumário

1 O que é Google App Engine	1
1.1 Sandbox do ambiente padrão	3
1.2 Armazenamento de dados	3
1.3 Agendamento de tarefas	5
1.4 Cotas e limites	6
1.5 Console de administração	7
2 Preparando o ambiente de desenvolvimento	9
2.1 Instalando o JDK 8	10
2.2 Instalando o Apache Maven	10
2.3 Instalando o IntelliJ IDEA Community	11
2.4 Instalando o Git	12
2.5 Instalando o Google Cloud SDK	13
2.6 Instalando o Google Cloud Code para o IntelliJ IDEA	14
2.7 Instalando o Postman	17
3 Desenvolvendo a primeira aplicação para o GAE	19
3.1 Criando uma aplicação no GAE	20
3.2 Mais configurações do Google Cloud SDK	25

Sumário	Casa do Código
3.3 Mais configurações do IntelliJ IDEA	26
3.4 Estrutura do projeto	30
3.5 Configurando o projeto para executar na máquina local	38
3.6 Configurando o projeto para ser depurado na máquina local	41
3.7 Executando ou depurando a aplicação localmente	47
3.8 Adicionando um serviço REST para teste	49
3.9 Publicando a aplicação no GAE	52
3.10 Experimentando o console do GAE	56
4 Construindo serviços REST	59
4.1 Entendendo a classe de serviço	60
4.2 Anotando a classe de serviço	61
4.3 Anotando os métodos da classe de serviço	62
4.4 Testando o serviço local com o Postman	64
4.5 Testando o serviço no GAE com o Postman	67
4.6 Criando ambientes de testes no Postman	67
4.7 Visualizando as mensagens de log no console do GAE	70
5 Criando um serviço REST completo	73
5.1 Criando o modelo de produtos	75
5.2 Criando a classe do serviço	76
6 Armazenando dados no Google Cloud Datastore	87
6.1 O que é o Google Cloud Datastore	87
6.2 Preparando o projeto para trabalhar com Datastore	89
6.3 Testando na máquina local	97
6.4 Índices do Datastore	99
6.5 Administrando o Datastore no GAE	101

7 Gerando mensagens de log	107
7.1 Configurando o projeto para geração de logs	107
7.2 Métodos para geração de logs	110
7.3 Visualizando as mensagens de log no GAE	114
8 Protegendo serviços com HTTP Basic Authentication	121
8.1 O que é HTTP Basic Authentication	123
8.2 Configurando o projeto com HTTP Basic Authentication	125
8.3 Testando o serviço de produtos com HTTP Basic Authentication	131
8.4 Adicionando anotações para controle de permissões e papéis	134
9 Adicionando o serviço de usuários	137
9.1 Criando o modelo de usuários	138
9.2 Preparando o modelo de usuários para o mecanismo de autenticação	141
9.3 Criando a camada de repositório de usuário	144
9.4 Configurando o Spring Security para acessar a base de usuários	160
9.5 Criando o serviço de usuários	165
9.6 Testando o serviço de usuário com o Postman	175
10 Enviando mensagens com o Firebase Cloud Messaging	178
10.1 O que é Firebase Cloud Messaging	179
10.2 Estratégia	182
10.3 Configurando o projeto no GAE para utilizar o FCM	184
10.4 O aplicativo móvel para Android	189
10.5 Enviando mensagens a aplicativos móveis com o FCM	192

10.6 Testando o envio de mensagens pelo FCM	198
11 Agendando tarefas no GAE	201
11.1 Como funcionam as tarefas agendadas no GAE	201
11.2 Criando um serviço agendado	202
11.3 Configurando a tarefa	204
11.4 Acompanhando a execução do console do GAE	206
12 Utilizando Memory Cache	209
12.1 O que é MemCache	210
12.2 Utilizando JCache	211
12.3 Usando MemCache no mecanismo de autenticação	213
12.4 Visualizando o MemCache do console do GAE	217
13 Protegendo serviços com OAuth 2.0	221
13.1 O que é OAuth 2.0	222
13.2 Alterando a aplicação gae_exemplo1	223
13.3 Testando o mecanismo de autenticação OAuth 2.0	230
14 Algo mais sobre Google App Engine	236

Versão: 24.3.14

CAPÍTULO 1

O QUE É GOOGLE APP ENGINE

O Google App Engine é uma plataforma de computação nas nuvens que permite a execução de aplicações Web na infraestrutura do Google, de forma fácil e escalável, sem a necessidade de manutenção em sistemas operacionais e servidores. Todas as aplicações ficam hospedadas sob o domínio *appspot.com*, e podem ser acessadas por todo o mundo ou apenas por pessoas/aplicações autorizadas.

O GAE faz parte do conjunto de serviços chamado **Google Cloud Platform**. As aplicações podem ser desenvolvidas em Java, Python, PHP e Go, com um ambiente de *runtime* específico para cada linguagem.

O estilo de cobrança do Google App Engine é o *pay as you go*, ou seja, você só paga pelo o que a aplicação usar de recursos da plataforma. Porém, é possível iniciar o desenvolvimento sem pagar nada, com limites razoáveis de banda, armazenamento e número de acessos por mês. Esses limites são suficientes para provas de conceitos, estudos e até mesmo aplicações de pequeno porte.

Durante este livro, será utilizada a linguagem de programação

Java para o desenvolvimento de aplicações de exemplo, que serão hospedadas no Google App Engine. Por isso, o foco das descrições a partir desse ponto será exclusivo para essa linguagem.

É possível escolher dentre duas formas de utilização do GAE, em termos de ambiente de execução: **padrão** e **flexível**. No ambiente flexível, a aplicação é executada dentro de um *container Docker*. No ambiente padrão, a aplicação é executada dentro de um *sandbox*, como será detalhado a seguir. Neste livro, usaremos somente o ambiente padrão de execução. A seguir, veja uma tabela que descreve algumas diferenças entre os dois ambientes:

Característica	Flexível	Padrão
Ambiente de execução	Container Docker	Sandbox
Tempo de escalabilidade	Lento	Rápido
Possibilita execução de código nativo e outros frameworks	Sim	Não
Permite execução com limites gratuitos	Não	Sim
Tempo de inicialização da instância	Em torno de minutos	Em torno de segundos
Estilo de cobrança	Por instâncias ativas	Por serviços ou requisições

Como pode ser observado, a escolha do ambiente de execução deve considerar a necessidade da aplicação. Por exemplo, se você precisa executar uma biblioteca Linux específica, a melhor opção seria o ambiente flexível. Em um outro caso, se você necessita que sua aplicação entre em execução somente através de uma requisição REST, a melhor opção seria o ambiente padrão, pois, durante **quase** todo o restante do tempo, a instância ficaria parada, sem cobrança por tempo de uso.

Como dito, o ambiente escolhido para o livro será o padrão, pois não temos nenhum requisito que necessite do ambiente flexível. Além disso, você poderá acompanhar 90% dos exemplos deste livro sem ter de pagar pelos serviços do GAE.

1.1 SANDBOX DO AMBIENTE PADRÃO

As aplicações no Google App Engine rodam em um ambiente seguro e controlado, que limita e controla o acesso ao sistema operacional onde está sendo executado. Tais limitações permitem que a infraestrutura controle melhor a alocação de recursos e as máquinas para as diversas aplicações. Alguns exemplos dessas limitações são:

- As aplicações só podem ser acessadas por meio de requisições HTTP ou HTTPS em portas padrões.
- Não é possível escrever em arquivos. Para leitura, só é possível ler arquivos que foram carregados pela própria aplicação. Para armazenamento de dados, a aplicação deve usar os mecanismos oferecidos pela plataforma, que serão vistos mais adiante.
- A aplicação só é executada em resposta a alguma requisição HTTP ou por meio de uma tarefa que foi agendada, devendo responder em até 60 segundos.

1.2 ARMAZENAMENTO DE DADOS

O App Engine fornece quatro formas de armazenamento de dados:

- **Google Cloud Datastore:** fornece um esquema de

armazenamento de dados do tipo NoSQL, com mecanismos de buscas e operações atômicas. Ele pode ser utilizado por aplicações que não serão portadas para outro tipo sistema de armazenamento, como banco de dados relacionais (por exemplo, o MySQL).

- **Google Cloud SQL:** provê um serviço de banco de dados relacional, semelhante ao MySQL. Esta é uma opção a ser escolhida caso haja a possibilidade de a aplicação precisar ser portada para outra plataforma que possui um sistema de armazenamento com banco de dados relacional. É importante ter essa informação para poder decidir entre esse sistema e o Google Cloud Datastore.
- **Google Cloud Storage:** fornece um serviço de armazenamento completamente gerenciável de objetos e arquivos com tamanhos da ordem de terabytes. Ele pode ser usado como repositório de arquivos ou de websites, armazenamento de backups ou logs. Os dados armazenados nesse serviço podem ser gerenciados por uma aplicação, utilizando bibliotecas específicas, pelo console do Google Cloud Platform, ou como sistema de arquivos mapeado em um sistema operacional.
- **Blobstore:** é usado para a aplicação armazenar e fornecer objetos chamados *blobs*, que são muito maiores do que os tipos de dados que podem ser armazenados no serviço de Datastore. Pode ser utilizado em conjunto com o Google Cloud SQL ou o Datastore, para casos em que a aplicação deve armazenar uma foto, um vídeo ou um arquivo muito grande que não será guardado em um banco de dados, por exemplo.

Neste livro, usaremos mais o mecanismo Google Cloud Datastore para armazenamento dos dados das aplicações de exemplo que serão criadas. Isso porque se trata de um serviço gratuito para o volume de dados que será usado, além de ser simples e, ao mesmo tempo, eficiente.

Obviamente, uma aplicação pode usar mais de um mecanismo de armazenamento de dados, por exemplo, utilizar o Datastore como banco de dados principal (com dados dos usuários, transações etc.) e o Blobstore para fotos.

A escolha entre os mecanismos de armazenamento deve levar em conta fatores como:

- **Custo de armazenamento:** o Datastore possui preços menores e até planos gratuitos, com certos limites, em relação ao Cloud SQL;
- **Tipo do dado a ser armazenado:** a escolha entre um banco relacional ou um NoSQL deve ser levada em conta entre usar o Datastore ou o Cloud SQL;
- **Possibilidade de portar a aplicação para outras plataformas:** utilizar o Cloud SQL pode tornar a aplicação mais fácil de ser portada para outras plataformas, com pequenas (ou, até mesmo, nenhuma) modificações;
- **Tamanho do dado a ser armazenado:** o armazenamento de fotos e arquivos grandes requer serviços como o Blobstore ou o Cloud Storage.

1.3 AGENDAMENTO DE TAREFAS

Tarefas podem ser agendadas para serem executadas, sem a

necessidade de responderem a requisições HTTP externas. Essa facilidade permite que a aplicação execute procedimentos agendados (a cada hora ou a cada dia), como requisições a recursos externos ou processos de limpeza do sistema de armazenamento.

1.4 COTAS E LIMITES

Para começar a desenvolver com o Google App Engine, basta criar uma conta do Google e publicar a aplicação. A partir daí, qualquer pessoa poderá acessar de qualquer lugar e sem nenhum custo. Porém, existem alguns limites para o uso gratuito da plataforma, que são:

- Registrar até 10 aplicações por usuário;
- Cada aplicação pode usar até 1 GB de armazenamento de dados (NoSQL) com limite de 50 mil operações de leitura/escrita por dia;
- 1 GB de tráfego de entrada e saída por dia.

Existem outros limites que podem ser consultados na página do Google App Engine, porém esses são os mais significativos. Ademais, o Google costuma mudar tais limites, seguindo as tendências de mercado, normalmente aumentando algumas das opções gratuitas.

Você também pode habilitar o mecanismo e a cobrança da sua aplicação, o que significa que nenhum serviço ou funcionalidade vai parar caso atinja o limite gratuito; mas é claro, será cobrado por isso.

Para maiores detalhes sobre as cotas do GAE, consulte a página <https://cloud.google.com/appengine/quotas>.

1.5 CONSOLE DE ADMINISTRAÇÃO

O Google App Engine possui um console de administração Web, no qual podem ser realizadas várias operações de gerenciamento das aplicações. Ele pode ser acessado em <https://console.cloud.google.com/appengine>.

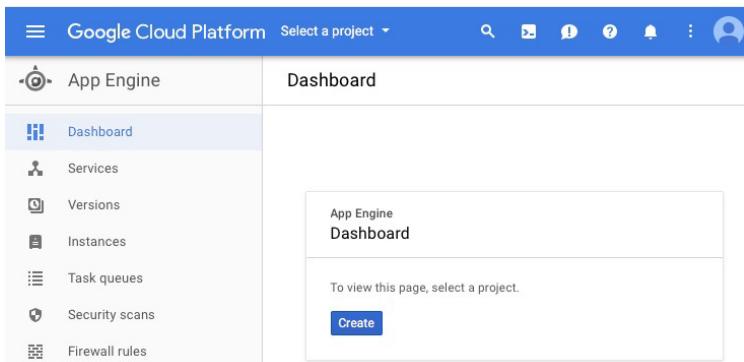


Figura 1.1: Página inicial

Nessa tela, você poderá acessar todas as opções do GAE para gerenciar seus projetos, que podem ser selecionados pela opção na barra superior, ao lado do título da página.

Caso você seja elegível para receber um crédito para usar serviços pagos do Google Cloud Platform, uma oferta aparecerá no topo dessa página. Caso contrário, você poderá utilizar outro e-mail do Google e aproveitar tais créditos. Será necessário cadastrar um cartão de crédito, caso você aceite tal oferta.

Neste livro, somente será utilizado o Google Cloud SQL como um serviço pago, por isso, caso não queira usar seu cartão de crédito, fique tranquilo para recusar a oferta e continuar utilizando os **limites gratuitos do GAE**.

Você usará muito esse console de administração das aplicações que forem publicadas no Google App Engine.

Conclusão

Agora que você já sabe um pouco sobre o que o Google App Engine pode oferecer como plataforma, você poderá preparar seu ambiente de desenvolvimento seguindo as instruções do próximo capítulo.

CAPÍTULO 2

PREPARANDO O AMBIENTE DE DESENVOLVIMENTO

Chegou a hora de preparar seu ambiente de desenvolvimento! Para desenvolver aplicações para o Google App Engine em Java, são necessários os seguintes programas e pacotes:

- IntelliJ IDEA (2019 ou superior), sendo possível utilizar a versão *Community* ou a *Ultimate*;
- Git;
- Java Development Kit 8 64 bits;
- Apache Maven 3.5 ou superior;
- Google Cloud SDK;
- Postman.

As versões apresentadas dos programas e pacotes formam uma combinação recomendada para o Google App Engine SDK e para a linguagem Java. Obviamente, se uma versão mais nova do SDK surgir, talvez seja possível usarmos também outras versões superiores de pacotes, do IntelliJ IDEA ou mesmo do Java. Tente utilizar essas versões recomendadas, pelo menos durante os exercícios do livro, para que você não tenha contratemplos

indesejados.

A seguir, veja os links e as instruções de instalação e configuração de cada um dos itens listados.

2.1 INSTALANDO O JDK 8

As aplicações para GAE, desenvolvidas neste livro, serão construídas com a linguagem Java. Por isso, é preciso instalar as ferramentas necessárias para trabalharmos com essa linguagem. A principal ferramenta é o Java SE Development Kit 8 (ou somente **JDK 8**), um pacote para desenvolvedores poderem construir aplicações em Java, na versão 8.

Para instalar o Java SE Development Kit 8, entre no seguinte link:

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>. Então, baixe o arquivo de acordo com o seu sistema operacional e siga as instruções de instalação.

2.2 INSTALANDO O APACHE MAVEN

Para construir os projetos para o GAE, será necessário ter o Apache Maven versão 3.5 ou superior. Ele vai facilitar muito a construção do projeto, bem como o gerenciamento das suas dependências.

Todo projeto construído com o Apache Maven possui um arquivo de configuração, chamado `pom.xml`. Nele são colocadas todas as bibliotecas necessárias para o projeto. Com base nestas configurações, o Apache Maven busca as dependências nele

configuradas, em seu repositório. Isso evita que você tenha de adicionar arquivos .jar manualmente em seu projeto.

Ao longo dos capítulos, você vai trabalhar com esse arquivo, adicionando bibliotecas ao seu projeto.

Para instalar essa ferramenta, proceda com os seguintes passos:

1. Baixe-o no link <https://maven.apache.org/download.cgi>, de acordo com o seu sistema operacional.
2. Siga as instruções de instalação de acordo com o descrito em: <https://maven.apache.org/install.html>. Nesse mesmo link, há algumas dicas para os usuários do sistema operacional Windows, que deverão ser seguidas.
3. Ainda nesse link, não se esqueça de configurar as variáveis de ambiente para que o Maven possa ser executado corretamente. Nessa página, há instruções de como fazer isso para os sistemas operacionais Windows, Linux e Mac OS X.

2.3 INSTALANDO O INTELLIJ IDEA COMMUNITY

Neste livro, os projetos serão construídos com a ferramenta IntelliJ IDEA Community. Ela é um ambiente integrado de desenvolvimento (IDE) para construção de projetos em Java e em outras linguagens. Com ela, você vai desenvolver, testar e instalar suas aplicações no GAE, sem a necessidade de digitar comandos em um terminal.

O Google tem adotado essa ferramenta e desenvolvido

plugins para facilitar o desenvolvimento de várias aplicações para seu ecossistema – dentre eles, o GAE. Uma das grandes vantagens de utilizá-la é a facilidade de se fazer depuração remota em aplicações desenvolvidas e hospedadas no GAE.

O link para a instalação do IntelliJ IDEA Community é: <https://www.jetbrains.com/idea/download>. Selecione a opção para baixar a versão **Community** e siga as instruções de instalação.

Caso você não tenha habilidades em trabalhar com essa IDEA, há alguns tutoriais para iniciantes neste link: <https://www.jetbrains.com/idea/documentation/>.

2.4 INSTALANDO O GIT

O Git será utilizado como repositório do código que for desenvolvido. Alguns sistemas operacionais já possuem o mínimo necessário pré-instalado, mas caso você deseje instalá-lo manualmente, siga as instruções no seguinte link, de acordo com a sua plataforma: <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>.

Para verificar se você já possui o Git instalado em sua máquina, execute o seguinte comando no terminal:

```
git --version
```

Caso o comando seja reconhecido, você já possui o Git instalado em seu sistema operacional. Caso contrário, siga o procedimento de instalação no link fornecido.

Com o próprio IntelliJ IDEA, é possível executar todos os comandos necessários para gerenciar seu código no repositório,

utilizando uma interface gráfica simples e intuitiva. Há alguns vídeos específicos para isso nos tutoriais apresentados na seção anterior, em <https://www.jetbrains.com/idea/documentation/>.

2.5 INSTALANDO O GOOGLE CLOUD SDK

O Google Cloud SDK é um conjunto de ferramentas para trabalhar com a plataforma de Cloud do Google, como o GAE. Com ele, será possível instalar e depurar as aplicações desenvolvidas para o GAE de forma integrada com o IntelliJ IDEA.

Para baixar e instalar, siga as instruções contidas em <https://cloud.google.com/sdk/>, de acordo com o seu sistema operacional. Por enquanto, **não é necessário executar as instruções para inicialização do SDK**, que estão descritas na sua página de instalação.

É necessário configurar as variáveis de ambiente do seu sistema operacional para incluir a pasta `bin` da instalação do Google Cloud SDK. Isso é necessário para que a sua execução seja feita de qualquer caminho, inclusive de dentro do IntelliJ IDEA. O procedimento varia entre os sistemas operacionais, mas é muito semelhante ao que foi executado no último passo da instalação do Apache Maven – basta alterar o caminho onde o Google Cloud foi instalado.

Depois de instalar o Google Cloud SDK, execute o seguinte comando em seu terminal, **fora do diretório em que o SDK foi instalado**, como instruído na documentação do Google, para adicionar a extensão do Google Cloud SDK para Java:

```
gcloud components install app-engine-java
```

Aguarde essa instalação antes de prosseguir para a seção seguinte.

2.6 INSTALANDO O GOOGLE CLOUD CODE PARA O INTELLIJ IDEA

Agora é hora de configurar o IntelliJ IDEA para podermos trabalhar com o Google App Engine. Dessa forma, será possível desenvolver, testar e publicar as aplicações no GAE. Para isso, execute os passos descritos a seguir:

1. Abra o IntelliJ IDEA.
2. Dentro dele, acesse as configurações da IDE pelo menu `File -> Settings` (no Windows), ou `IntelliJ IDEA -> Preferences` (no Mac OS X).

A mesma tela pode ser acessada na janela de boas-vindas do IntelliJ IDEA, como mostra a figura a seguir:

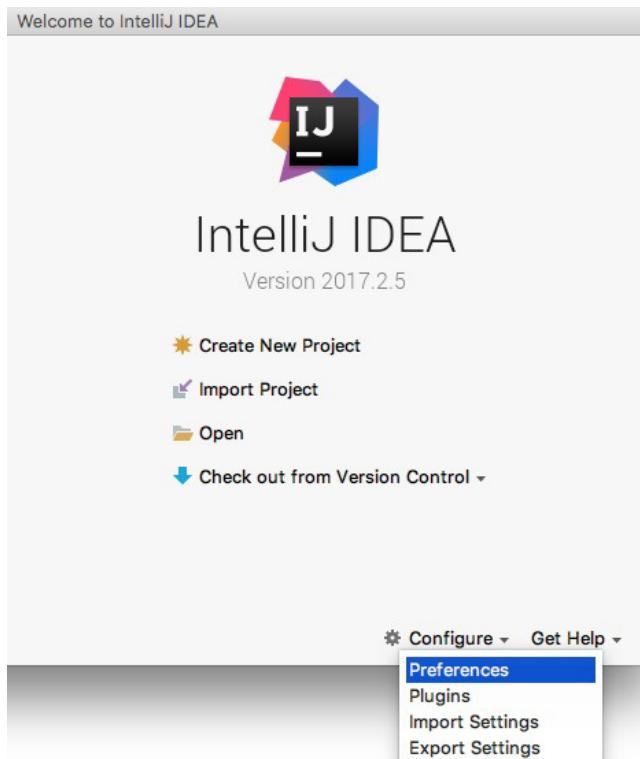


Figura 2.1: Instalando o Google Cloud Code

3. Na aba de navegação da janela que aparecer, escolha a opção **Plugins**.
4. Digite **Google Cloud Code** na caixa de pesquisa e clique no link **Search in repositories**.
5. Na janela que aparecer com o resultado da pesquisa, selecione o **Google Cloud Code** e clique no botão **Install** para iniciar o processo de instalação. Veja a figura a seguir:

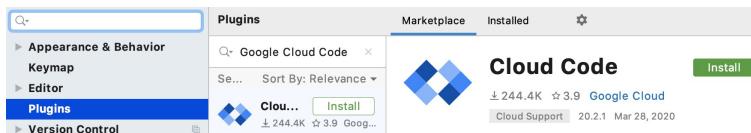


Figura 2.2: Instalando o Google Cloud Code

6. Após o processo de instalação, reinicie o IntelliJ IDEA.
7. Depois de o IntelliJ IDEA reiniciar, acesse novamente o menu de configurações – no Windows, em File -> Settings e, no Mac OS X, em IntelliJ IDEA -> Preferences -, e escolha a opção Cloud Code -> Cloud SDK . Nessa tela, configure o local onde o pacote Cloud SDK foi instalado, como na figura a seguir:

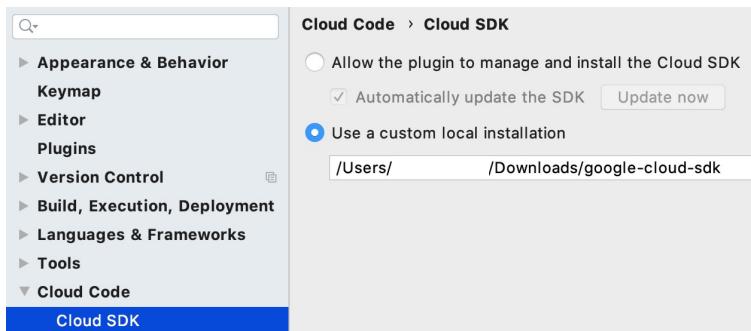


Figura 2.3: Configurando o Google Cloud Code

Se você tiver dúvidas de onde instalou o Google Cloud SDK, digite o seguinte comando no terminal: `gcloud info` . Ele mostrará o seu local.

8. Por último, configure a localização do Java SE Development

Kit. Para isso, acesse o menu File -> Project Structure , então selecione o JDK instalado em seu sistema operacional. Caso não apareça nenhuma opção, clique no botão New , escolha JDK e aponte para o local onde você o instalou.

2.7 INSTALANDO O POSTMAN

O Postman é uma aplicação que permite testar API de forma fácil e rápida. Como neste livro serão desenvolvidas muitas API REST, é necessário utilizar uma boa ferramenta para esse trabalho.

A seguir, veja algumas das vantagens do Postman:

- Permite salvar as configurações das requisições para serem executadas posteriormente, sem a necessidade de escrevê-las toda vez que abrir a ferramenta;
- Permite agrupar as requisições em coleções, deixando o ambiente mais organizado;
- Possibilita trabalhar com variáveis de ambiente, para facilitar os testes no ambiente de desenvolvimento e na nuvem, sem a necessidade de criar requisições específicas para cada endereço da aplicação.

Novamente, se você já está acostumado a trabalhar com outra ferramenta, não é necessário utilizar o Postman. Este livro vai usá-lo simplesmente para aproveitar as facilidades que ele traz.

Para instalar o Postman, acesse <https://www.getpostman.com>, e siga as instruções de acordo com o seu sistema operacional.

Conclusão

Pronto! Com esses passos, seu ambiente de desenvolvimento já está totalmente configurado para trabalhar com o Google App Engine. No próximo capítulo, você vai construir sua primeira aplicação em Java e publicá-la no GAE.

CAPÍTULO 3

DESENVOLVENDO A PRIMEIRA APLICAÇÃO PARA O GAE

Para entender como funciona o processo de criação de uma aplicação em Java, e depois como é feita a publicação no GAE, você construirá um primeiro projeto, chamado `gae_exemplo1`. Nos capítulos seguintes, você vai incrementá-lo, adicionando:

- Um serviço **REST**;
- Persistência de dados no **Google Cloud Datastore**;
- Mensagens de **log**;
- **Autenticação** HTTP Basic;
- Interação com o **Firebase Cloud Messaging**.

Com isso, você também aprenderá técnicas interessantes de trabalhar com o console de administração do Google App Engine, como:

- Monitorar a execução de uma aplicação;
- Visualizar os acessos, com monitoramento das mensagens de erro;
- Visualizar as mensagens de log geradas pela aplicação;

- Gerenciar o Google Cloud Datastore para visualizar (e até editar) os dados gravados pela aplicação;
- Gerenciar as versões da aplicação;
- Configurar a aplicação para poder utilizar o Firebase Cloud Messaging;
- Fazer depuração remota com o IntelliJ IDEA.

Esse projeto será feito com o Spring Boot (<https://projects.spring.io/spring-boot>), uma forma rápida e fácil de se construir projetos Web em Java, utilizando o framework Spring. Com o Spring Boot, será possível criar os serviços REST apenas com anotações em classes Java, sem a necessidade de configurar arquivos .xml. Os arquivos .xml que o projeto terá serão apenas por parte do Maven, ou do próprio GAE.

O Google App Engine já possui total compatibilidade para projetos criados com o Spring Boot. Não se preocupe.

Criando a conta no Google Cloud

Caso você ainda não tenha se registrado no Google Cloud para trabalhar com o GAE, basta criar uma conta do Google e acessar o link mostrado na próxima seção. Utilize-a para criar os projetos e seguir o conteúdo do livro.

3.1 CRIANDO UMA APLICAÇÃO NO GAE

Para começar a criar o projeto `gae_exemplo1`, execute os passos a seguir:

1. Acesse o console de administração do GAE no endereço <https://console.cloud.google.com/appengine>

2. Caso você ainda não possua nenhum projeto criado, aparecerá um botão com o texto `Select a project`, localizado no canto superior esquerdo da página. Clique nesse botão para ser direcionado à página de criação de projetos.
3. Caso você já possua algum projeto criado, clique no menu `drop down`, localizado no mesmo lugar. Uma tela como a da figura a seguir aparecerá, listando todos os seus projetos.



Figura 3.1: Criando um novo projeto no GAE

Então, selecione o botão `New Project` para ser direcionado à página de criação de projetos.

4. Na tela que aparecer, escolha o nome do seu projeto. Você verá o `Project ID`, logo abaixo no nome do projeto que digitou, que deverá ser associado à criação do projeto no IntelliJ IDEA. Ele também será único na plataforma do GAE, pois fará parte da URL de acesso quando ele for publicado.

New Project

Project name * _____

exemplo1 ?

Project ID: elegant-rock-272515. It cannot be changed later. [EDIT](#)

Location * _____

No organization [BROWSE](#)

Parent organization or folder

[CREATE](#) [CANCEL](#)

Figura 3.2: Escolhendo o nome do projeto

Guarde o valor do Project ID para quando for criar o projeto no IntelliJ IDEA.

5. Clique no botão `create` para prosseguir com o processo de criação.
6. Depois de o passo anterior ser concluído, aparecerá uma tela de boas-vindas, como a seguir:

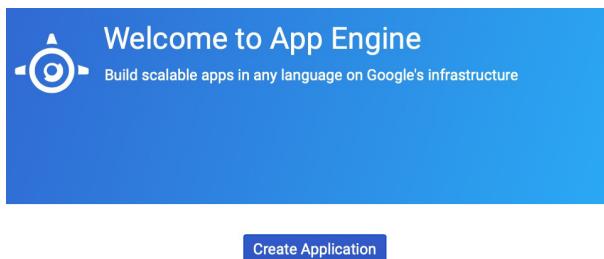


Figura 3.3: Telas de boas vindas do projeto

Clique no botão `Create Application` para prosseguir com a criação da aplicação dentro do App Engine.

- Nesse momento, você será consultado sobre a região em que deseja criar a sua aplicação:

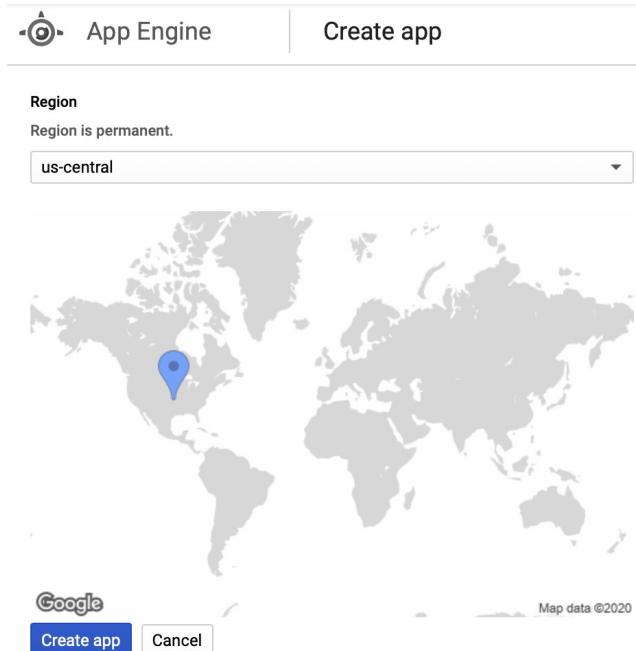


Figura 3.4: Escolhendo a região do projeto

A escolha da região deve levar em conta alguns fatores como **custo de operação e proximidade geográfica dos usuários**. Algumas regiões são mais caras que outras. Tente levar pelo menos esses dois aspectos em consideração quando estiver criando uma aplicação de produção real.

Para o caso desse exercício, mantenha a região indicada na

página e clique no botão `Create` .

8. A próxima tela questiona sobre a linguagem e o tipo de ambiente:

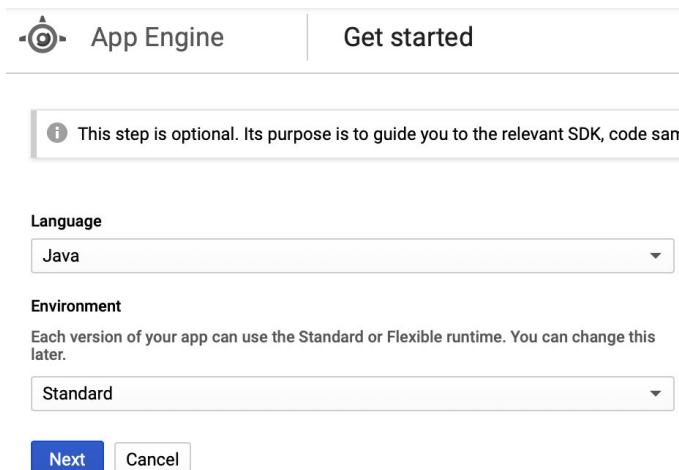


Figura 3.5: Linguagem e ambiente

Escolha **Java** e **Standard**. Em seguida, clique em `Next` .

9. Depois de o projeto ser criado, aparecerá uma janela ao lado direito da página, oferecendo um tutorial para a criação da aplicação com o Maven. Se tiver curiosidade, você pode até segui-lo, mas os passos serão descritos aqui, por isso, pode clicar no botão `I'll do this later` .

Quando a primeira versão da sua aplicação for publicada, aparecerão várias informações na tela do projeto, que serão detalhadas mais adiante.

3.2 MAIS CONFIGURAÇÕES DO GOOGLE CLOUD SDK

Agora que você já possui uma conta para acessar o Google Cloud Platform, é necessário realizar um passo a mais para configurar o seu ambiente de desenvolvimento: a autorização do Google Cloud Tools (`gcloud tool`). Isso é necessário para que ele acesse a sua conta e manipule os projetos criados por você na plataforma, como realizar a publicação dos seus projetos, por exemplo.

Para isso, execute os passos a seguir:

1. Abra o terminal do seu sistema operacional.
2. Digite o comando `gcloud auth login`. Caso ele não seja reconhecido pelo seu computador, volte ao capítulo anterior e verifique se essa ferramenta foi instalada e, principalmente, se você adicionou o caminho da pasta `bin` da instalação do Google Cloud Tools nas variáveis de ambiente do seu sistema operacional.
3. O navegador Web padrão do seu computador abrirá, solicitando as credenciais de acesso ao Google Cloud Platform . Digite o e-mail e a senha usados para a criação do projeto nos passos da seção anterior.
4. Por fim, para garantir que todos os componentes do Google Cloud Tools estão atualizados, execute o seguinte comando no terminal:

```
gcloud components update
```

Com essas configurações, será possível gerenciar e publicar sua aplicação dentro do IntelliJ IDEA.

3.3 MAIS CONFIGURAÇÕES DO INTELLIJ IDEA

Como dito anteriormente, o IntelliJ IDEA é um ambiente integrado de desenvolvimento (ou IDE), para construção de projetos em Java e em outras linguagens. Com ele, você vai desenvolver, testar e instalar suas aplicações no GAE, sem a necessidade de digitar comandos em um terminal.

Todos os projetos deste livro usarão o Maven como gerenciador de dependências. Porém, é necessário utilizar uma versão mínima dele para que tudo funcione perfeitamente, que é a 3.5. Caso você ainda não o tenha instalado, volte ao capítulo anterior e siga os passos específicos para tal.

Dependendo da versão do IntelliJ IDEA que você instalou, pode acontecer dele usar uma versão anterior do Maven. Por isso, abra o IntelliJ IDEA e vá até a sua seção de configurações/preferências, depois execute os passos a seguir:

1. Na caixa de pesquisa, digite `maven`.
2. Na janela que aparecer, certifique-se de que a versão do Maven é igual ou superior a 3.5.
3. Caso a versão seja inferior, configure o campo `Maven home directory` para apontar para a pasta onde você instalou a versão do Maven, conforme as instruções do capítulo anterior:

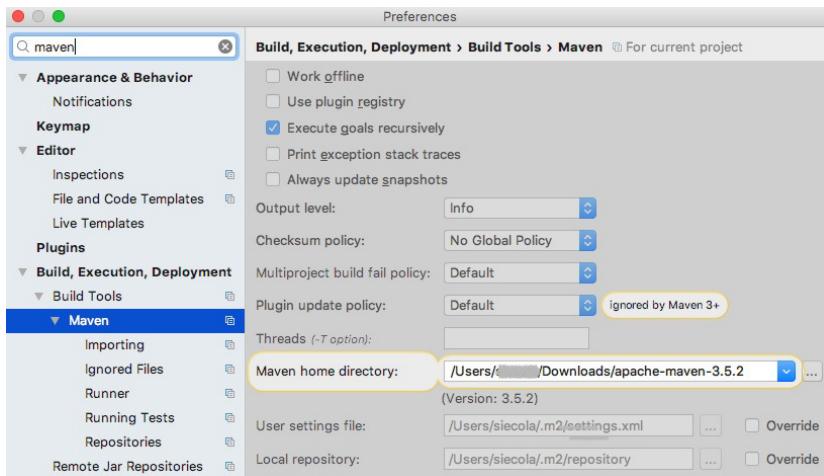


Figura 3.6: Configurando a versão do Maven

Ok! Agora só falta começar a criar o projeto!

Criando o projeto Spring Boot com o Spring Initializr

Como dito no início deste capítulo, o projeto a ser criado inicialmente será bem simples, para que você aprenda os primeiros passos com o IntelliJ IDEA até a publicação da aplicação no GAE. Mais adiante, outras funcionalidades serão acrescentadas, como um serviço REST.

O projeto será construído com o Spring Boot, e a forma mais simples de se começar uma criação é utilizar a ferramenta **Spring Initializr**. Esta é uma página Web onde você pode configurar tudo o que precisa em sua aplicação inicialmente e baixar um arquivo compactado, com toda a estrutura do seu projeto configurada.

Para começar, acesse a página da ferramenta no seguinte

endereço: <https://start.spring.io>. Logo no topo, selecione as opções para criar um projeto baseado em Maven e em Java, como já foi descrito no capítulo anterior. Além disso, selecione o Spring Boot na versão 2.2.6.

No canto esquerdo da página, preencha os seguintes campos:

- **Group:** esse campo corresponde, por exemplo, ao nome da sua empresa ou de seu site. O exemplo utilizado aqui será `br.com.siecola` .
- **Artifact:** ele representa o nome do seu projeto. O exemplo usado aqui será `gae_exemplo1` .

Esses dois campos vão formar o pacote base da sua aplicação, que nesse caso será `br.com.siecola.gae_exemplo1` . Você pode usar outros valores, se quiser, como o endereço de seu site ou de sua empresa, mas tenha a cautela de fazer as alterações quando esses valores aparecerem em configurações do projeto mais adiante.

Agora configure o campo `Packaging` com a opção `War` , que é o formato que o GAE aceita no momento da publicação da aplicação. Esse é um formato muito comum de distribuição de aplicações Java para Web, utilizado em servidores de aplicação como o Tomcat.

No campo `Java` , tenha certeza de selecionar a versão 8.

No canto superior direito, clique no botão `Add dependencies...` e selecione a opção `Web` , para poder incluir as bibliotecas necessárias para se construir um projeto Web com o Spring Boot – algo necessário para a construção de serviços REST na primeira aplicação de exemplo.

Veja como devem ficar as suas configurações:

The screenshot shows the Spring Initializr web application. At the top, there are sections for 'Project' (Maven Project selected), 'Language' (Java selected), and 'Dependencies' (Spring Web selected). Under 'Project Metadata', fields are filled with Group: br.com.siecola, Artifact: gae_exemplo1, Name: gae_exemplo1, Description: Demo project for Spring Boot, Package name: br.com.siecola.gae_exemplo1, Packaging: War selected, and Java version: 8 selected. Below these, there are checkboxes for Spring Boot versions: 2.0.0.M3, 2.3.0 (SNAPSHOT), 2.2.7 (SNAPSHOT) (selected), 2.1.14 (SNAPSHOT), and 2.1.13.

Figura 3.7: Configurando projeto Spring Boot

Por fim, clique no botão `Generate`. A ferramenta vai gerar seu projeto com as configurações realizadas, e o seu navegador Web vai realizar o download do arquivo compactado. Descompacte-o em um local de sua preferência, na sua máquina de desenvolvimento.

Com o projeto descompactado em sua máquina, abra o IntelliJ IDEA e acesse o menu `File -> Open`, apontado para **dentro da pasta descompactada** do projeto. A pasta correta que você deve abrir no IntelliJ é a que contém o arquivo `pom.xml`.

Feito isso, o IntelliJ entenderá que esse é um projeto baseado em Maven e vai exibir sua estrutura correta, além de baixar todas

as dependências definidas no arquivo `pom.xml`. Mais adiante, você adicionará novas bibliotecas a ele, alterando o arquivo `pom.xml`.

3.4 ESTRUTURA DO PROJETO

Dentro do IntelliJ IDEA, a estrutura de diretórios e arquivos do projeto criado deverá ficar como mostra a figura a seguir:

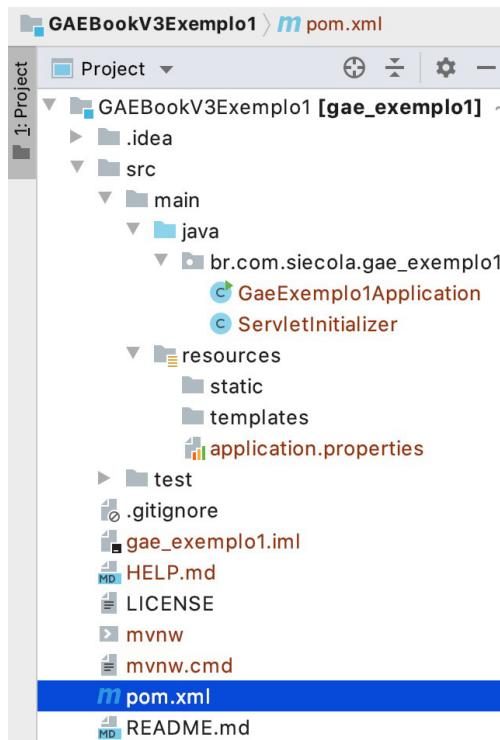


Figura 3.8: Estrutura do projeto

Dentro dessa estrutura, há vários arquivos que foram criados pela ferramenta `Spring Initializr`, seguindo a arquitetura

escolhida no processo de criação. No momento, os interessantes são:

- **Arquivo pom.xml** : esse é o arquivo base de um projeto Maven. Nele serão inseridas todas as dependências do projeto, bem como algumas configurações para que ele seja executado localmente.
- **Arquivo application.properties** : por enquanto, ele deverá estar vazio, mas nele podem ser feitas algumas configurações do projeto Spring Boot, como a porta do servidor nível de log. No momento, ele ficará vazio, pois as configurações necessárias da aplicação, a ser publicada no GAE, serão feitas em outros lugares.

Outros arquivos, de classes e/ou de configurações, serão adicionados ao projeto à medida que ele for ganhando novas funcionalidades.

Para que esse projeto Spring Boot execute corretamente, como uma aplicação válida do Google App Engine, é necessário adicionar dois arquivos na pasta webapp/WEB-INF . São eles:

- **Arquivo src/main/webapp/WEB-INF/appengine-web.xml** : nele estão as configurações de *deploy and run* da aplicação, bem como a sua versão. Veja como deve ser sua estrutura básica:

```
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
  <version>1</version>
  <threadsafe>true</threadsafe>
  <runtime>java8</runtime>
  <system-properties>
    <property name="java.util.logging.config.file" va
```

```
lue="WEB-INF/logging.properties"/>
</system-properties>
</appengine-web-app>
```

Após a criação desse arquivo com esse conteúdo, o IntelliJ IDEA deverá questionar se você deseja configurar o projeto para adicionar o framework do Google App Engine:

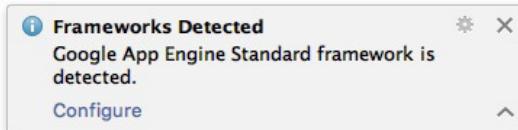


Figura 3.9: Configure o framework do GAE

Clique no link `Configure` para que seu projeto passe a ser reconhecido pelo IntelliJ IDEA como um projeto que utiliza o framework do Google App Engine Standard. Caso você não tenha feito esse passo, no momento em que essa mensagem aparecer, é possível realizar a mesma configuração, clicando com o botão direito sobre o projeto e acessando o menu `Open Module Settings`. Nessa tela, simplesmente clique no botão `+` e escolha a opção `Google App Engine Standard`.

Essa configuração é necessária para você utilizar todos os recursos do *plugin* do Google Cloud Tools no IntelliJ IDEA, por exemplo, publicar seu projeto no GAE pela sua interface gráfica, como você verá mais adiante.

- **Arquivo `src/main/webapp/WEB-INF/logging.properties`** : nele são configuradas as opções de log da aplicação. Repare que ele é citado no

arquivo `appengine-web.xml`. Por enquanto, ele pode ficar com o seguinte conteúdo:

```
.level = INFO
```

Depois de criar as pastas `webapp/WEB-INF` dentro da pasta `src/main`, inclua esses dois arquivos em seu projeto no caminho `src/main/webapp/WEB-INF`, com o conteúdo apresentado aqui. Veja como deverá ficar ao final:

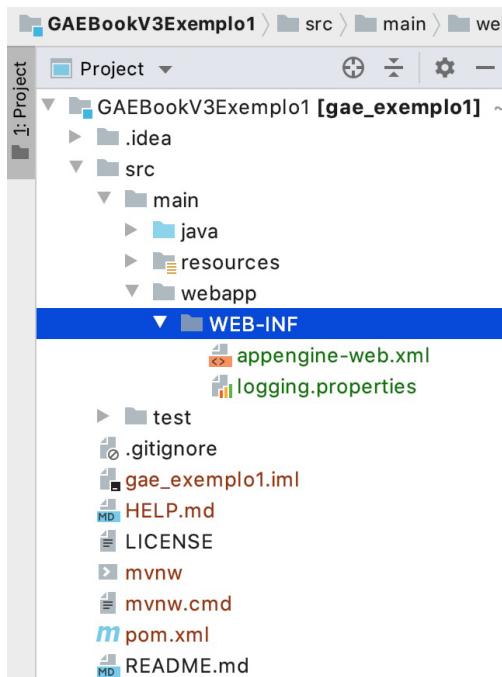


Figura 3.10: Nova estrutura do projeto

Arquivo pom.xml

Todo projeto criado com o Maven tem um arquivo chamado `pom.xml`. Nele são adicionadas todas as dependências de

bibliotecas e configurações de execução da aplicação. Caso esse arquivo ainda não esteja aberto no IntelliJ IDEA, abra-o para realizar algumas configurações específicas para o projeto recém-criado.

A primeira configuração a ser feita é a versão do SDK do App Engine. É importante utilizar a mesma versão que foi instalada em seu computador, com os passos descritos no capítulo anterior. Para verificar qual é exatamente a versão, abra um terminal e digite o seguinte comando:

```
gcloud info
```

Na resposta a esse comando, localize o seguinte componente:

```
app-engine-java: [1.9.79]
```

Nesse exemplo, a versão do componente app-engine-java que está instalado é a 1.9.79 . De posse dessa versão, abra o arquivo pom.xml e adicione a seguinte linha ao final da seção properties :

```
<appengine.sdk.version>1.9.79</appengine.sdk.version>
```

No lugar da versão 1.9.79 desse exemplo, coloque a versão do componente app-engine-java exibido na execução do comando gcloud info .

Para que o projeto recém-construído tenha todas as dependências necessárias, ainda é necessário realizar algumas configurações no arquivo pom.xml , bem como adicionar algumas dependências:

1. Na seção de plugins do arquivo pom.xml , delimitado pelas tags <build><plugins> , inclua o plugin do Google Cloud

Platform para simplificar o processo de execução e instalação da aplicação no GAE:

```
<!-- Configuração para execução local-->
<plugin>
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>appengine-maven-plugin</artifactId>
    <version>1.3.1</version>
</plugin>
```

2. Na seção de dependências do arquivo `pom.xml`, delimitado pela tag `<dependencies>`, exclua o servidor Tomcat, pois o GAE utiliza o Jetty. A configuração total da seção de dependências deverá ficar como no trecho a seguir:

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
            <exclusion>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-tomcat</artifactId>
            </exclusion>
        </exclusions>
    </dependency>

    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>com.google.appengine</groupId>
        <artifactId>appengine-api-1.0-sdk</artifactId>
        <version>${appengine.sdk.version}</version>
    </dependency>
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jul-to-slf4j</artifactId>
    <scope>provided</scope>
</dependency>
</dependencies>
```

Para que o IntelliJ IDEA possa reconfigurar o projeto com as dependências corretas para essa versão, clique com botão direito sobre o projeto e escolha a opção Maven -> Reimport . Aguarde até que todas as dependências sejam baixadas.

Preparando para a execução local

Antes de executar a aplicação, confira se está tudo certo com o seu projeto, ou seja, se ele pode ser compilado e *empacotado* corretamente. Como ele foi montado com o Maven, o IntelliJ IDEA fornece uma forma amigável de executar alguns de seus comandos. Para isso, acesse o menu View -> Tool Windows -> Maven Projects .

Você verá uma aba lateral listando seu projeto e os comandos disponíveis para ele, como mostra a figura a seguir:

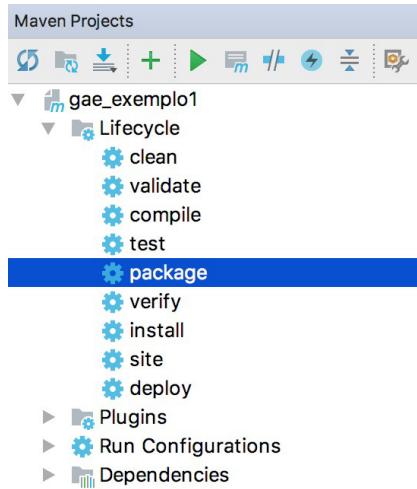


Figura 3.11: Comandos Maven

Repare dentro da seção `Lifecycle` que existem as opções `clean` e `package`. Como os nomes sugerem, a primeira opção remove todas as pastas de saída utilizadas pelo processo de compilação, e a segunda gera um diretório contendo seu projeto compilado.

Execute o comando `clean`, com um duplo clique nele. Em seguida, execute o comando `package`. Isso fará com que todo o projeto seja compilado e preparado para **ser executado**.

Repare no canto esquerdo do IntelliJ IDEA que uma nova pasta chamada `target` aparecerá na estrutura do seu projeto. Se tudo der certo, ela terá uma estrutura semelhante a essa:

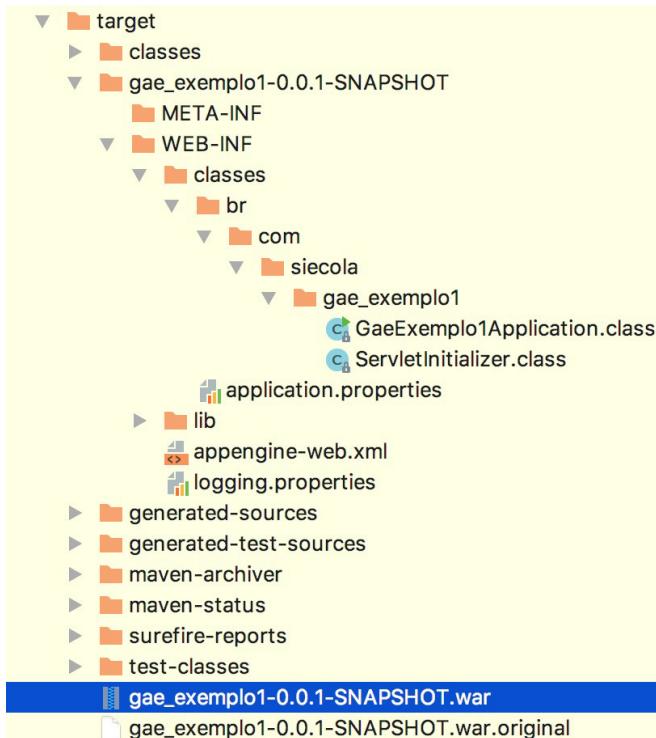


Figura 3.12: Pasta target do projeto

Caso essa pasta não tenha aparecido, verifique o log de execução desse comando para ver se aconteceu algum erro no processo de compilação. Caso tenha ocorrido, você também pode voltar alguns passos deste capítulo para verificar se fez tudo corretamente.

3.5 CONFIGURANDO O PROJETO PARA EXECUTAR NA MÁQUINA LOCAL

Com as configurações realizadas até o momento, falta apenas

um passo para que a aplicação seja executada na máquina local, dentro do IntelliJ IDEA. Logo, é necessário criar uma configuração de execução, pelo menu `Run -> Edit Configurations`.

Na tela que aparecer, clique no botão `+` e selecione a opção `Maven`. Nessa tela, dê um nome à nova configuração (por exemplo, `Run`) e configure o parâmetro `Command Line` com o seguinte valor: `appengine:run`. Veja como ela deverá ficar:

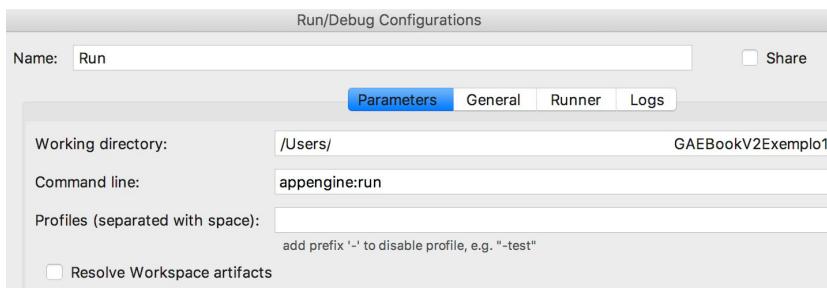


Figura 3.13: Configuração para execução local

Depois de realizar essas configurações, clique no botão `OK` para salvar a nova configuração de execução e fechar a janela. Agora você pode selecionar essa nova configuração de execução no IntelliJ IDEA, na barra de botões do canto superior direito:



Figura 3.14: Barra de execução

Selecione a configuração que você criou e clique no botão com o símbolo de `Play`. Sua aplicação deverá começar a ser executada.

Obviamente, ainda não há nada dentro dela para ser acessado ou testado, mas você deverá ver um log de execução, ao final do

processo, semelhante a esse:

```
[INFO] GCLOUD: 2017-12-30 16:21:13.361:INFO:oejsh.ContextHandler: main: Started c.g.a.t.d.j.DevAppEngineWebAppContext@4726927c{/,{ file:///Users/paulosiecola/pcs/projetos_particulares/livro_gae/GAEB ookV3Exemplo1/target/gae_exemplo1-0.0.1-SNAPSHOT/,AVAILABLE}{}/Use rs/paulosiecola/pcs/projetos_particulares/livro_gae/GAEBookV3Exem plo1/target/gae_exemplo1-0.0.1-SNAPSHOT}
[INFO] GCLOUD: 2017-12-30 16:21:13.373:INFO:oejs.AbstractConnecto r:main: Started NetworkTrafficSelectChannelConnector@6ffa56fa{HTTP/1.1,[http/1.1]}{localhost:8080}
[INFO] GCLOUD: 2017-12-30 16:21:13.383:INFO:oejs.Server:main: Sta rted @5541ms
[INFO] GCLOUD: Dec 30, 2017 6:21:13 PM com.google.appengine.tools .development.AbstractModule startup
[INFO] GCLOUD: INFO: Module instance default is running at http:/ /localhost:8080/
[INFO] GCLOUD: Dec 30, 2017 6:21:13 PM com.google.appengine.tools .development.AbstractModule startup
[INFO] GCLOUD: INFO: The admin console is running at http://local host:8080/_ah/admin
[INFO] GCLOUD: Dec 30, 2017 4:21:13 PM com.google.appengine.tools .development.DevAppServerImpl doStart
[INFO] GCLOUD: INFO: Dev App Server is now running
```

Sua aplicação deve estar rodando no seguinte endereço, como mostra o log anterior: <http://localhost:8080/>. Porém, como dito anteriormente, ainda não há nada dentro dela que possa ser acessado. No momento, se você acessar essa página, aparecerá apenas um erro informando que nada foi encontrado.

O intuito do livro não é construir aplicações com páginas Web no GAE, mas, sim, serviços REST. Então, veremos mais sobre isso adiante.

Console de administração local

Há também uma página que exibe um console de administração do GAE local, que exibe algumas informações da

infraestrutura simulada em sua máquina. Enquanto sua aplicação estiver em execução, ele pode ser acessado do seguinte endereço: http://localhost:8080/_ah/admin.

Veja como ele é:



Figura 3.15: Console de administração local

Nos capítulos adiante, você terá a oportunidade de trabalhar um pouco mais com esse console.

3.6 CONFIGURANDO O PROJETO PARA SER DEPURADO NA MÁQUINA LOCAL

O Google App Engine fornece toda a infraestrutura necessária para execução e depuração de aplicações. Entretanto, em tempo de desenvolvimento, é muito mais confortável utilizar alguma forma de testar o código na própria máquina local, dentro do ambiente de desenvolvimento.

Para isso, é necessário realizar algumas configurações no IntelliJ IDEA, bem como no arquivo pom.xml do projeto, para que você possa depurar a sua aplicação localmente:

1. Abra o arquivo pom.xml e localize a seção plugins dentro de builds :

```
<build>
    <!-- for hot reload of the web application-->
    <outputDirectory>${project.build.directory}/${project.
    build.finalName}/WEB-INF/classes</outputDirectory>
    <plugins>
```

2. Logo abaixo da tag plugins , adicione um novo item:

```
<!-- Configuração para depuração local-->
<plugin>
    <groupId>com.google.appengine</groupId>
    <artifactId>appengine-maven-plugin</artifactId>
    <version>${appengine.sdk.version}</version>
    <configuration>
        <jvmFlags>
            <jvmFlag>-Xdebug</jvmFlag>
            <jvmFlag>-agentlib:jdwp=transport=dt_socket,ad
            dress=5005,server=y,suspend=n</jvmFlag>
        </jvmFlags>
    </configuration>
</plugin>
```

Essa é parte da configuração da JVM para executar a aplicação em modo de depuração remota, dentro do IntelliJ, com o servidor de desenvolvimento do Google Cloud Tools. Caso tenha dúvidas de como esse arquivo deva ficar no final, consulte o repositório no seguinte endereço: <https://github.com/siecola/GAEBookV3Exemplo1>.

3. No IntelliJ IDEA, crie uma configuração de execução/depuração do Maven, para iniciar o servidor de

desenvolvimento onde a aplicação será executada. Para isso, acesse o menu Run -> Edit Configuration . Na tela que aparecer, clique no botão + e escolha a opção Maven :

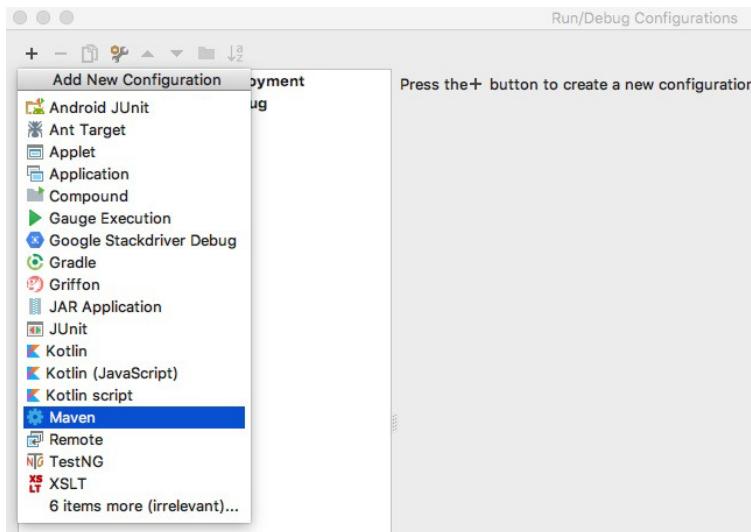


Figura 3.16: Adicionando a configuração de execução Maven

4. Crie essa nova configuração com o nome de `appengine:devserver_start` , que deverá ser o mesmo valor para o parâmetro Command line nessa tela. Isso fará com que o IntelliJ IDEA execute esse comando antes de iniciar sua aplicação.
5. Ainda nessa tela, configure o parâmetro Working directory apontando para a pasta onde está o seu projeto. São apenas esses dois ajustes nessa tela de configuração. Veja como deve ficar:

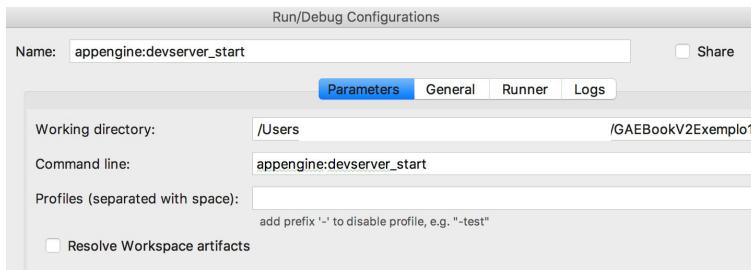


Figura 3.17: Ajustando a configuração de execução Maven

6. Crie uma configuração para depuração remota no menu **Run -> Edit Configuration**. Na tela que aparecer, clique no botão **+** e escolha a opção **Remote** :

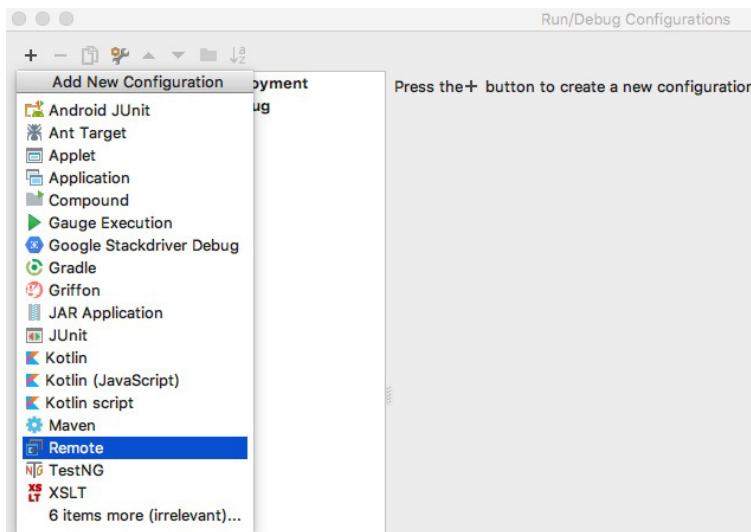


Figura 3.18: Adicionando a configuração de execução remota

Esse é o último passo para fazer com que sua aplicação seja

executada na sua máquina de desenvolvimento local.

7. Nessa nova configuração de execução, insira um nome (por exemplo, `remote debug`), para que você possa escolhê-lo quando for executá-lo. Ainda nessa tela, configure o parâmetro `Search sources using module's classpath` para apontar para a sua aplicação.

Agora o que falta é adicionar a primeira configuração de execução, que foi o servidor de desenvolvimento local `appengine:devserver_start` , para ela ser lançada antes. Isso é feito na seção `Before launch` , na parte inferior dessa tela, ao clicarmos no botão `+` e escolhermos a configuração `appengine:devserver_start` . Veja como deve ficar:

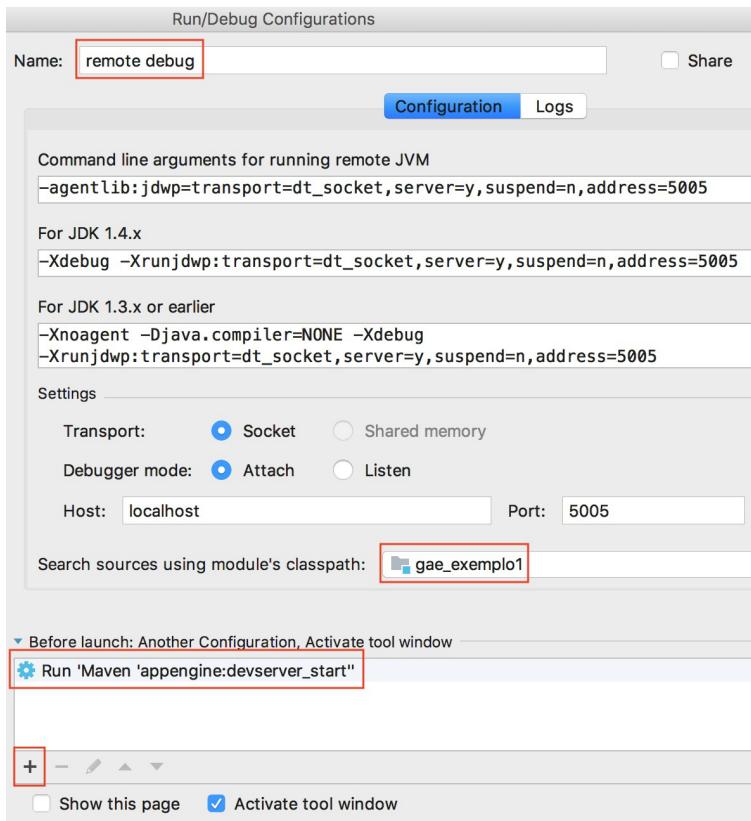


Figura 3.19: Ajustando a configuração de execução remota

Resumindo, esses passos foram necessários para que você pudesse depurar a sua aplicação de dentro do IntelliJ IDEA, sem a necessidade de executar comandos pelo terminal do seu sistema operacional.

3.7 EXECUTANDO OU DEPURANDO A APLICAÇÃO LOCALMENTE

Até o momento, o arquivo pom.xml possui duas configurações distintas de execução local da aplicação:

- Uma para execução, propriamente dita, sem a possibilidade de inserção de pontos de parada (ou *breakpoints*);

```
<!-- Configuração para execução local-->
<plugin>
  <groupId>com.google.cloud.tools</groupId>
  <artifactId>appengine-maven-plugin</artifactId>
  <version>1.3.1</version>
</plugin>
```

Essa configuração só deve estar ativa quando você for somente EXECUTAR A APLICAÇÃO, ou seja, quando **não** for colocar *breakpoints* para depurar o seu código. Quando for depurar a aplicação, comente esse trecho do arquivo pom.xml para que a depuração funcione adequadamente.

- Outra para depuração da aplicação pelo servidor de desenvolvimento do Google Cloud Tools.

```
<!-- Configuração para depuração local-->
<plugin>
  <groupId>com.google.appengine</groupId>
  <artifactId>appengine-maven-plugin</artifactId>
  <version>${appengine.sdk.version}</version>
  <configuration>
    <jvmFlags>
      <jvmFlag>-Xdebug</jvmFlag>
```

```
<jvmFlag>-agentlib:jdwp=transport=dt_socket,address=5005,server=y,suspend=n</jvmFlag>
</jvmFlags>
</configuration>
</plugin>
```

Essa configuração só deverá estar ativa quando você for somente DEPURAR A SUA APLICAÇÃO, ou seja, quando for colocar *breakpoints* para depurar o seu código. Caso contrário, comente esse trecho de código no arquivo `pom.xml` para que o modo de execução funcione adequadamente.

Essas duas configurações não devem existir simultaneamente no arquivo `pom.xml`, como dito anteriormente. Por isso, tenha o cuidado de deixar sempre uma, de acordo com a operação que você for fazer: executar ou depurar a aplicação.

Para depurar a aplicação, vá à barra de ferramentas, localizada no canto superior direito do IntelliJ IDEA, e selecione a opção `remote debug`, que você criou no início deste capítulo:



Figura 3.20: Executando a aplicação

Depois de selecionar essa configuração de execução, clique no botão `Debug remote debug`, que é o único marcado em verde na figura anterior. Isso fará com que tudo o que foi feito até o momento comece a ser executado, ou seja:

- O servidor de desenvolvimento será executado antes da aplicação;
- A aplicação vai conectar-se ao servidor de desenvolvimento e executar seu código.

Nesse modo de depuração, você pode colocar um ponto de parada (ou *breakpoint*) no código, para poder depurar passo a passo, exatamente como em qualquer aplicação Java. Isso é importante para você conseguir executar a aplicação linha a linha e visualizar o conteúdo das variáveis.

MODO DE DEPURAÇÃO

Habilite o modo de depuração apenas quando você realmente desejar depurar sua aplicação na sua máquina local, executando o código passo a passo. Por isso, deixe a configuração para depuração local, do arquivo `pom.xml`, comentada e só habilite quando precisar depurar seu código.

3.8 ADICIONANDO UM SERVIÇO REST PARA TESTE

Agora que a aplicação está pronta para ser executada localmente no IntelliJ IDEA, crie um serviço REST para poder interagir de alguma forma com ela; uma espécie de **Hello World**, mas com requisições HTTP. Nos próximos capítulos, você criará um serviço de gerenciamento de produtos, um pouco mais complexo que esse exemplo.

Com esse serviço de teste, você será capaz de enviar uma requisição HTTP para a aplicação e ver sua resposta. Da mesma forma, quando publicá-la no GAE, você poderá ter certeza de que ela realmente está em execução e visualizar gráficos de requisições em seu console.

Para começar, crie um novo pacote na aplicação, chamado controller , onde ficarão todas as classes que implementarão os serviços REST. Dentro desse pacote, crie uma nova classe chamada HelloController . Veja como deve ficar a estrutura do projeto:

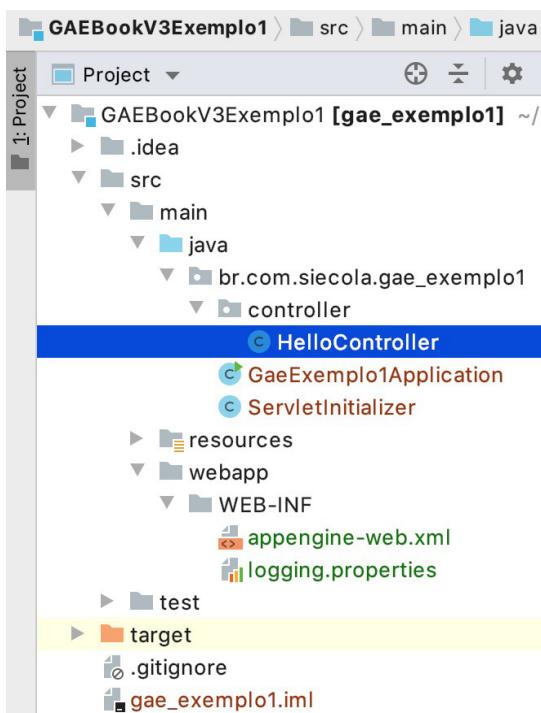


Figura 3.21: Novo controller de teste

Nessa classe, será implementado o serviço REST de teste. Nesse

primeiro momento, não se importe com os detalhes dessa implementação, pois isso será totalmente detalhado no próximo capítulo. Veja como ela deve ficar:

```
package br.com.siecola.gae_exemplo1.controller;

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;

import java.util.logging.Logger;

@RestController
@RequestMapping(path="/api/test")
public class HelloController {

    private static final Logger log = Logger.getLogger(HelloController.class.getName());

    @GetMapping("/dog/{name}")
    public String hello(@PathVariable String name) {
        log.info("Dog: " + name);

        return "Hello World - " + name;
    }
}
```

Caso você tenha criado o projeto com outro pacote base, lembre-se de alterar o nome do pacote na primeira linha do arquivo.

Execute a aplicação pela configuração de execução simples (não a de depuração), que nesse exemplo foi criado com o nome de Run , e acesse o seguinte endereço do seu navegador Web: <http://localhost:8080/api/test/dog/Matilde>. Você deverá ver a

seguinte mensagem: Hello World - Matilde .

Fantástico! Você criou seu primeiro serviço REST na aplicação.

Como dito anteriormente, todos os detalhes dessa implementação – principalmente sobre a formação do endereço que foi acessado – serão explicados no próximo capítulo.

Você também deverá ver uma mensagem de log no console do IntelliJ IDEA, no seguinte formato:

```
[INFO] GCLOUD: INFO: Dev App Server is now running  
[INFO] GCLOUD: Dec 31, 2017 10:49:48 AM br.com.siecola.gae_exempl  
01.controller.HelloController hello  
[INFO] GCLOUD: INFO: Dog: Matilde
```

3.9 PUBLICANDO A APLICAÇÃO NO GAE

Agora que sua aplicação está montada e executando normalmente na sua máquina local, chegou a hora de publicá-la no GAE, para que ela se torne disponível na internet, para qualquer pessoa acessar! O IntelliJ IDEA possui todas as ferramentas necessárias para isso, por meio de menus e telas de configuração, sem a necessidade de digitar comandos em um terminal.

Para começar o processo de publicação ou *deploy*, pare a execução da aplicação e execute os passos descritos a seguir:

1. No IntelliJ IDEA, acesse o menu Tools -> Cloud Code -> App Engine -> Deploy to App Engine . A seguinte tela

deverá aparecer:

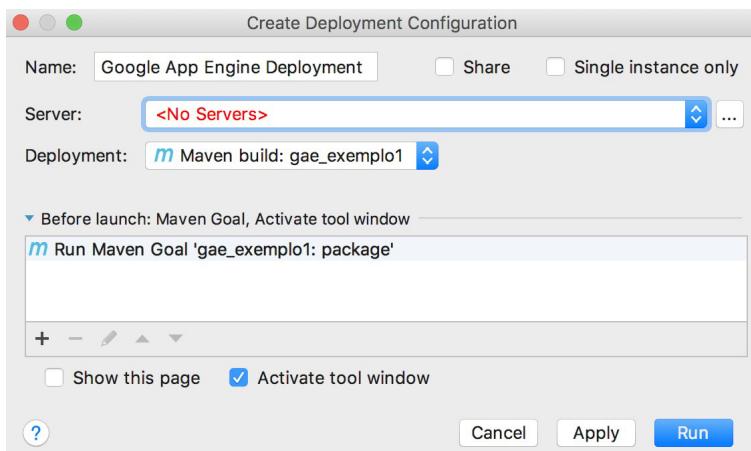


Figura 3.22: Publicando a aplicação – Passo 1

Se essa é a primeira vez que você está publicando sua aplicação, é provável que você tenha de realizar alguns passos a mais, como configurar o servidor de execução, que na figura anterior aparece em vermelho.

2. Clique no botão com os 3 pontos logo após a caixa de seleção do servidor e adicione um novo, como mostra a figura a seguir:

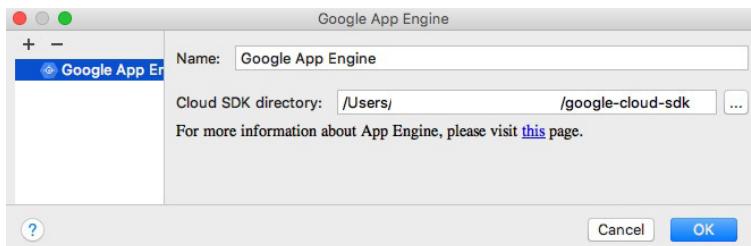


Figura 3.23: Publicando a aplicação – Passo 2

Clique em `OK` para voltar à tela anterior.

3. De volta à primeira tela, selecione o `drop down Project` para selecionar o projeto. Caso você ainda não tenha efetuado o login da sua conta do GAE, faça-o utilizando a mesma conta que usou no início deste capítulo.
4. Depois de ter efetuado o login, escolha o projeto criado no início deste capítulo. Ele será utilizado para publicar sua aplicação.
5. Tenha certeza de marcar as seguintes caixas de seleção:
 - *Update dos, dispatch, cron, queues and datastore indexes;*
 - *Promote the deployed version to Receive all traffic;*
 - *Stop previous version.*

Todos os itens mencionados aqui serão detalhados mais adiante, mas eles se referem à atualização de arquivos de configuração e gerenciamento de versões da aplicação, durante o seu processo de *deploy*.

6. Deixe os demais campos dessa tela com as configurações que já estavam e clique em `Run` para iniciar o processo de publicação de sua aplicação no projeto selecionado. Veja como essa tela deverá ficar:

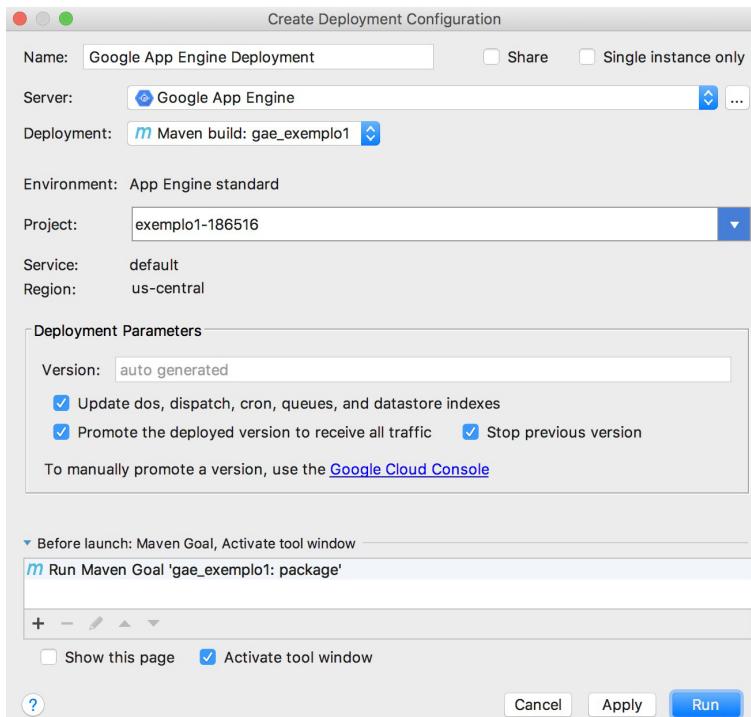


Figura 3.24: Última tela de publicação no GAE

7. Ao final do processo de publicação, você verá o endereço em que ele foi publicado, no log de execução do IntelliJ IDEA. Algo como o exemplo a seguir:

```
Deployed service [default] to [https://exemplo1-186516.appspot.com]
```

```
You can stream logs from the command line by running:  
$ gcloud app logs tail -s default
```

```
To view your application in the web browser run:  
$ gcloud app browse --project=exemplo1-186516  
[2017-12-31 11:01:03] Maven build: gae_exemplo1. Project: exemplo1-186516. Version: auto' has been deployed successfu
```

11y.

8. Clique no link para poder acessar a aplicação em execução no GAE, porém, não se esqueça de concatenar o endereço do serviço REST de teste criado – que nesse exemplo seria: <https://exemplo1-186516.appspot.com/api/test/dog/Matilde>. Você deverá ver a mesma mensagem no seu navegador Web, quando testou a aplicação na sua máquina local.

Tudo pronto! Sua aplicação já está rodando na infraestrutura do Google App Engine!

3.10 EXPERIMENTANDO O CONSOLE DO GAE

O console de administração do GAE, acessado em <https://console.cloud.google.com/appengine>, oferece várias ferramentas de configuração e administração das aplicações publicadas nele. Acesse esse link e escolha sua aplicação para ver seu dashboard.

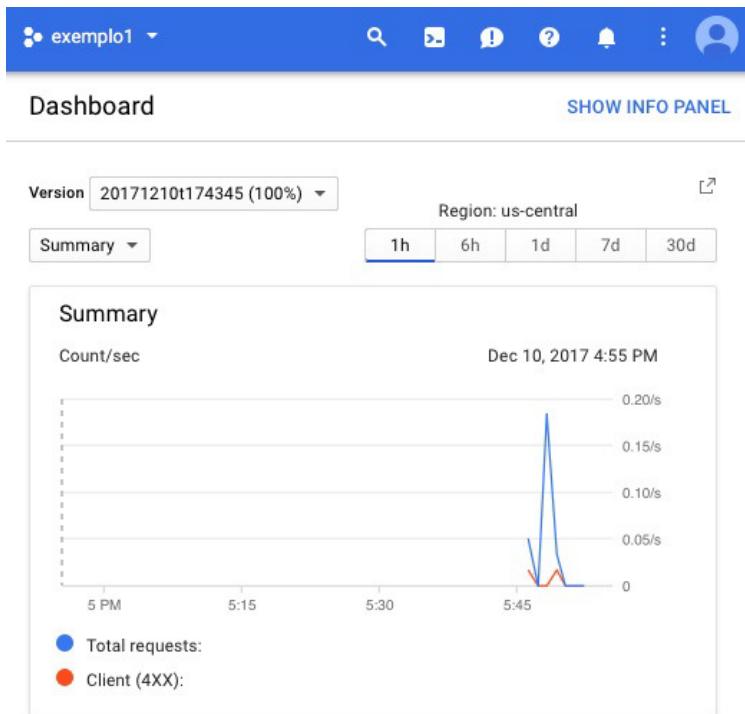


Figura 3.25: Console de administração

Aqui estão algumas ferramentas importantes desse console:

- Um *dashboard* com as principais informações gerais sobre a aplicação selecionada, com um gráfico de requisições;
- Informações sobre as versões publicadas;
- Informações sobre as instâncias em execução;
- Quais tarefas estão agendadas;
- Detalhes de utilização dos recursos da plataforma e as cotas disponíveis.

À medida que você estudar os capítulos do livro, você vai aprender mais sobre o console de administração do GAE e como ele pode ajudar.

Conclusão

Neste capítulo, você aprendeu a publicar seu primeiro projeto no Google App Engine! Além disso, pode conhecer um pouco do seu console de administração Web, uma ferramenta que pode ajudar muito no gerenciamento das aplicações publicadas no GAE.

No próximo capítulo, você verá como criar um serviço REST nesse primeiro projeto, utilizando o framework Jersey.

CAPÍTULO 4

CONSTRUINDO SERVIÇOS REST

No capítulo anterior, você aprendeu a criar e publicar uma aplicação Spring Boot no ambiente padrão do GAE. Além disso, adicionou um serviço REST simples para poder testar sua aplicação, para que ela respondesse às requisições HTTP.

Você deve ter percebido que não foi necessário realizar nenhuma configuração especial na aplicação para que o serviço REST passasse a funcionar, como uma configuração de arquivos `xml`, por exemplo. O motivo da simplicidade de se implementar um serviço REST é porque o projeto foi desenvolvido com o Spring Boot. Ele é uma forma de criar aplicações baseadas no *framework* Spring, mas muito mais orientada a convenções de estruturas e anotações do que configurações em arquivos `xml`.

O que possibilitou a criação de serviços REST nesse projeto, que você começou a construir no capítulo anterior, foi a biblioteca `spring-boot-starter-web`. Ela foi adicionada ao arquivo `pom.xml` no momento da criação do projeto pela interface do Spring Initializr:

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

```
<artifactId>spring-boot-starter-web</artifactId>
<exclusions>
  <exclusion>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
  </exclusion>
</exclusions>
</dependency>
```

Com apenas essa biblioteca do Spring Boot, você pode criar quantos serviços REST forem necessários em seu projeto, apenas seguindo a convenção de anotações (que será detalhada neste capítulo).

4.1 ENTENDENDO A CLASSE DE SERVIÇO

Para fins de organização em seu projeto, é interessante fazer a divisão dos serviços REST para serem implementados em classes. Dessa forma, cada método público dessa classe vai representar uma operação do serviço.

Para entender melhor, observe a tabela a seguir:

Mundo REST	Mundo Java
Serviço REST	Implementado por uma classe, adequadamente anotada
Endereço do serviço REST	Anotação principal da classe
Operação de um serviço REST	Implementado por um método público da classe, adequadamente anotado
Endereço da operação do serviço REST	Anotação do método

Tenha sempre essa representação em mente quando for criar serviços e operações REST em uma aplicação Spring Boot, ou até mesmo em outros frameworks de mercado.

Falando especificamente de um projeto construído com Spring Boot, o mapeamento da classe e seus métodos para serviços e operações REST, respectivamente, são feitos de forma automática, simplesmente através das anotações (que serão detalhadas a seguir), sem a necessidade de configurar arquivos XML do projeto.

No serviço de teste apresentado no capítulo anterior, a classe que implementa o serviço REST é a seguinte:

```
public class HelloController
```

Essa classe representa o serviço test que foi criado. Ela contém um único método, que é:

```
public String hello(@PathVariable String name)
```

Ele representa a operação hello do serviço test. Tanto a classe quanto o método possuem anotações, detalhadas a seguir.

4.2 ANOTANDO A CLASSE DE SERVIÇO

Para que uma classe seja reconhecida com um serviço REST, em uma aplicação Spring Boot, é necessário anotá-la adequadamente. A primeira anotação é a que diz que ela é um controller, que implementará um serviço REST, o `@RestController`.

Ela deve ser colocada logo acima da declaração da classe. Assim, ela faz com que o Spring Boot faça o seu mapeamento para ser reconhecida como uma classe que vai implementar um serviço REST.

A outra anotação necessária é a que define qual será o endereço base do serviço. No exemplo de teste, ficou assim:

```
@RequestMapping(path="/api/test")
```

Isso faz com que todas as operações implementadas nessa classe sejam acessadas pelo endereço base /api/test . Logo, a definição da classe HelloController deve ficar como no trecho a seguir:

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

@RestController
@RequestMapping(path="/api/test")
public class HelloController {
```

Feito isso, todos os métodos públicos dessa classe, devidamente anotados, poderão ser mapeados como operações do serviço Hello – como será detalhado na seção seguinte.

4.3 ANOTANDO OS MÉTODOS DA CLASSE DE SERVIÇO

Para que um método de uma classe de serviço seja reconhecido como uma operação de tal serviço, é necessário decorá-lo com algumas anotações. Nesse caso, existem várias anotações possíveis, bem como algumas variações destas.

No exemplo do serviço, criado no capítulo anterior, foi criada uma operação de nome dog que recebe um parâmetro de nome name . A anotação que definiu tais atributos foi a @GetMapping("/dog/{name}") . Dessa forma, essa operação pode ser acessada pelo seguinte endereço: /api/test/dog/<name> , através do verbo HTTP GET , sendo que a primeira parte – ou seja, /api/test – foi definida na anotação da classe.

Existe ainda uma outra informação nessa anotação, que é o fato de existir um parâmetro (de nome `name`) definido no caminho de acesso à operação. Ele será passado como parâmetro de entrada do método que vai tratar a requisição a essa operação. Isso é feito na declaração do método, como pode ser visto no trecho a seguir:

```
public String hello(@PathVariable String name)
```

Repare que, na assinatura do método, há a anotação `PathVariable`, antes da declaração do parâmetro `name` do tipo `String`. Ela indica que o valor desse parâmetro virá como parâmetro no caminho da requisição HTTP a esse método.

O nome do atributo (nesse caso, `name`) deve ser igual ao definido na anotação `GetMapping`, para que o Spring saiba atribuir os valores corretamente, em caso de haver mais de um parâmetro na requisição. Veja como ficou a implementação desse método, com essas anotações:

```
@GetMapping("/dog/{name}")
public String hello(@PathVariable String name) {
    log.info("Dog: " + name);

    return "Hello World - " + name;
}
```

Essa operação simplesmente imprime o parâmetro `name` no log e retorna a mensagem `Hello World -` concatenada com esse valor. Essa mensagem é o valor de retorno da operação em questão, que é recebida por quem faz a requisição a essa operação.

A mensagem de log pode ser visualizada no console do IntelliJ IDEA e também no console do GAE, como veremos nas seções a seguir. No capítulo seguinte, você verá outras anotações para definir outros atributos para receberem requisições HTTP de

outros verbos.

4.4 TESTANDO O SERVIÇO LOCAL COM O POSTMAN

No capítulo anterior, você deve ter testado o serviço `Hello` utilizando um navegador Web da sua máquina local. Isso até pode ser útil para testes pequenos e rápidos, mas sem customizações de parâmetros. Porém, é aconselhável usar um cliente no qual seja possível configurar, por exemplo, qual verbo HTTP será usado para fazer a requisição. Além disso, em determinadas requisições, pode ser necessário enviar uma mensagem sem corpo.

Existem vários clientes para testarmos serviços REST, e este livro adota o Postman, que é bem completo para as necessidades das aplicações aqui desenvolvidas. Além disso, ele é muito utilizado por empresas desenvolvedoras de API, sendo que algumas delas até fornecem as chamadas ***collections***, para que você as importe no Postman e já saia testando seus serviços.

A instalação do Postman já foi detalhada no segundo capítulo. Por isso, se tiver alguma dúvida de como fazê-la, volte até ele para deixar tudo configurado a partir desse ponto.

Com a aplicação executando no IntelliJ IDEA, abra o Postman. Você deverá ver uma tela como a da figura a seguir:

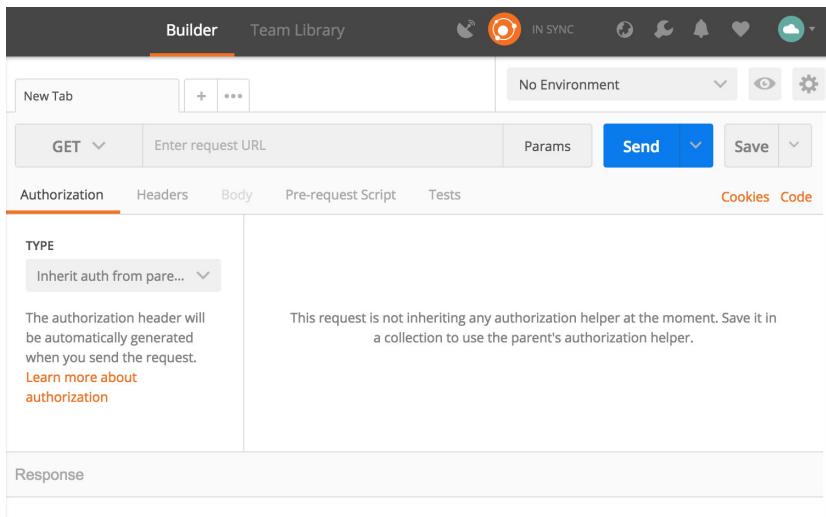


Figura 4.1: Tela inicial do Postman

Logo abaixo do nome da aba, `New Tab`, você pode escolher o verbo HTTP a ser usado na requisição que fará. No exemplo da figura, ele está configurado para fazer uma requisição utilizando o `GET`.

Ao lado do verbo HTTP, é possível digitar o endereço da requisição a ser feita, que no exemplo é o serviço de teste da aplicação (em execução na sua máquina local, ou seja, <http://localhost:8080/api/test/dog/Matilde>). Lembre-se de que o último parâmetro do caminho da requisição é o valor do atributo `name`, que será passado para o método que implementa a operação.

Feito isso, clique no botão `Send` para enviar essa requisição para a sua aplicação em execução na máquina local:

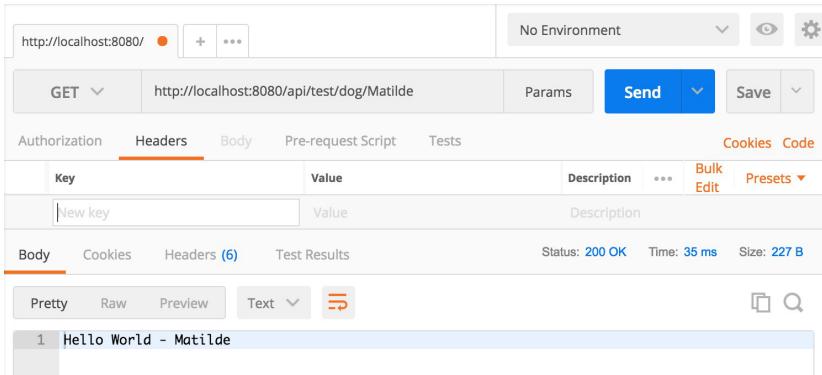


Figura 4.2: Primeira requisição com o Postman

A resposta aparece na seção `Body` do Postman, juntamente com outras informações, como:

- Código de retorno, que nesse caso deve ser **200 OK**, indicando que a requisição foi processada com sucesso;
- Tempo de execução, que no exemplo levou **35 ms**;
- Tamanho da mensagem de resposta, contendo o corpo da mensagem de retorno, além de cabeçalhos.

No capítulo seguinte, você aprenderá a fazer requisições mais complexas com o Postman, como também entenderá a importância de usarmos um cliente para requisições HTTP em vez de somente um navegador Web.

Experimente alterar o último parâmetro da URL para testar a operação com nomes diferentes. Veja que a mensagem de log exibida no IntelliJ IDEA vai refletir o valor recebido como parâmetro da operação.

4.5 TESTANDO O SERVIÇO NO GAE COM O POSTMAN

No capítulo anterior, você publicou a aplicação com o serviço `Hello` de teste, e testou com o seu navegador Web. Agora você vai acessá-lo usando o Postman, de uma forma semelhante com a que fez na seção anterior.

Dentro do campo da URL de destino da requisição, altere a parte do endereço com `http://localhost:8080/`, para o endereço no qual sua aplicação foi publicada no GAE.

Se tiver dúvidas de qual é o endereço correto, consulte a seção `Dashboard` da sua aplicação, no console do GAE. No canto superior direito dessa página, está o endereço público de acesso a ela.

Trocando o endereço local pelo endereço que sua aplicação está publicada, você pode repetir os passos da seção anterior e acessar o serviço `Hello` de teste da sua aplicação no GAE. Veja que, dessa forma, basta alterar o endereço base para que você consiga usar o Postman para acessar tanto a aplicação local quanto a aplicação hospedada no GAE.

4.6 CRIANDO AMBIENTES DE TESTES NO POSTMAN

Embora seja muito simples apenas alterar o endereço base da

sua aplicação na URL de requisição do Postman (para poder acessar a execução local ou no GAE), é possível deixar esse processo mais simples ainda, com **ambientes de teste** no Postman. Dessa forma, você apenas precisará alterar o ambiente – por exemplo, `GAE_local` ou `GAE_cloud` – para alternar as requisições entre a execução local no GAE.

Para configurar um novo ambiente de teste no Postman, clique no botão `Environment quick look`, localizado no canto superior direito da tela:

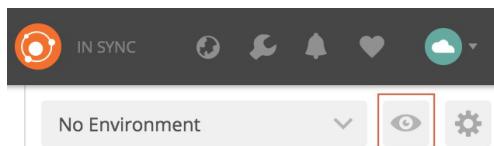


Figura 4.3: Criando ambientes no Postman

No *popup* que aparecer, clique no botão `Add`, na seção `Environment`. Na janela aberta, dê um nome ao novo ambiente (por exemplo, `GAE_local`). Nessa mesma tela, crie uma variável para o ambiente, como `address`, com o valor `http://localhost:8080` – que é o endereço que a aplicação roda quando está executando em sua máquina local. A tela de criação do novo ambiente deverá ficar como a figura a seguir:

The screenshot shows the 'Manage Environments' section of the Postman application. At the top, there's a header bar with the title 'MANAGE ENVIRONMENTS' and a close button ('X'). Below the header, there are two tabs: 'Manage Environments' (which is selected, indicated by an orange underline) and 'Environment Templates'. A sub-header 'Edit Environment' is displayed above a table. The table has a single row for the environment 'GAE_local'. The table columns are 'Key', 'Value', and 'Bulk Edit'. Under 'Key', there are two entries: 'address' with a checked checkbox and 'New key' with a 'Value' field. The 'Value' for 'address' is set to 'http://localhost:8080'.

Key	Value	Bulk Edit
<input checked="" type="checkbox"/> address	http://localhost:8080	
New key	Value	

Figura 4.4: Criando um ambiente local no Postman

Agora repita o processo para criar um ambiente para acessar a sua aplicação publicada no GAE, mas faça duas configurações diferentes:

- Dê um outro nome ao novo ambiente (por exemplo, `GAE_cloud`);
- Configure a chave `address` com o valor da URL da sua aplicação publicada no GAE.

OK! Mas como usamos esses ambientes com a variável `address` no Postman? Muito simples, no campo onde você digitou a URL da requisição no Postman, altere o endereço base (que deve estar como `http://localhost:8080/api/test/dog/Matilde`) para `\{\{address\}\}/api/test/dog/Matilde`. Perceba que agora a URL é composta pela variável que você criou nos ambientes `GAE_local` e `GAE_cloud`.

Para alternar entre os ambientes, selecione-os na barra superior direita, do lado esquerdo do botão `Environment quick`

look . Você deverá ver os dois que criou. Então, basta escolher um para pode testar no ambiente local ou no *cloud*.

Você também pode salvar essa requisição ao serviço Hello de teste no Postman. Para isso, clique no botão Save e digite um nome para ela. Nessa mesma tela para salvar a requisição, é possível criar uma nova coleção para deixar todas as requisições desse serviço agrupadas. Esse é um passo opcional, mas interessante para deixar tudo mais organizado.

4.7 VISUALIZANDO AS MENSAGENS DE LOG NO CONSOLE DO GAE

A operação do serviço Hello de teste imprime uma mensagem de log quando é executada. É possível monitorar os logs de execução da sua aplicação no console do GAE, com a ferramenta Logging .

Para localizar essa ferramenta, vá até o Dashboard e clique no botão ao lado do título Google Cloud Platform , no canto superior esquerdo. No menu que aparecer, vá até a seção Operations , clique na opção Logging e, em seguida, em Logs Viewer . Você deverá ver uma tela semelhante a esta:

The screenshot shows the GAE Logging viewer interface. At the top, there are several dropdown menus and buttons: 'Filter by label or text search' (with a placeholder 'Hello'), 'GAE Application' (set to 'All logs'), 'Any log level' (set to 'INFO'), 'Last hour' (selected), and 'Jump to now'. Below these is a table header: 'Showing logs from the last hour ending at 10:38 AM (UTC-8)' and 'View Options'. The table lists four log entries:

Date	Time	Log Level	Method	Status	Size	Latency	User Agent	Request URL
2017-12-31	10:30:51.602	INFO	GET	200	118 B	24 ms	PostmanRu...	/api/test/dog/Matilde
2017-12-31	10:37:26.240	INFO	GET	200	119 B	31 ms	PostmanRu...	/api/test/dog/Doralice
2017-12-31	10:37:13.566	INFO	GET	200	117 B	107 ms	PostmanRu...	/api/test/dog/Hannah
2017-12-31	10:37:35.417	INFO	GET	404	207 B	369 ms	PostmanRu...	/api/test/dogs/Hannah

At the bottom of the table, there are two buttons: 'Load older logs' and 'Load newer logs'.

Figura 4.5: Monitorando logs no console do GAE

Nos filtros na parte superior da tela, selecione a sua aplicação e também a opção All logs . Você deverá ver todos os logs de todas as requisições feitas ao serviço Hello de teste. Em cada linha, você pode ver as seguintes informações:

- Data e hora da requisição;
- Verbo HTTP utilizado na requisição;
- Código de retorno da requisição;
- Tamanho, em bytes, da requisição;
- Tempo de processamento;
- Tipo do cliente que fez a requisição;
- Endereço destino, como todos os parâmetros.

Você pode clicar em uma dessas linhas para obter maiores detalhes sobre a requisição recebida, bem como visualizar a mensagem de log e o local no código em que ela foi gerada.

The screenshot shows a log entry from December 31, 2017, at 10:37:31 UTC-8. The log message is: "GET /api/test/dog/Hannah 200 117 B 107 ms PostmanRu... /api/test/dog/Hannah". Below this, the log content is expanded:

```
187.73.169.0 - - [31/Dec/2017:16:37:31 -0200] "GET /api/test/dog/Hannah HTTP/1.1" 200 117 - "PostmanRuntime/7.1.2" "exemplol-186516.appspot.com" ms=107 cpu_ms=173 cpm_usd=1.3075e-8 loading_request=0 instance=00c61b17c83ff55ebd21753c62eb7a3825ee82f9e6a40a0b08b67e90bc30b1077cc6a20ccefe1730 app_engine_release=1.9.54 trace_id=-
```

The expanded log object includes the following fields:

- httpRequest: ...
- insertId: "5a492e6c000427a33f01b84c"
- labels: ...
- logName: "projects/exemplol-186516/logs/appengine.googleapis.com%2Frequest_log"
- operation: ...
- protoPayload: {
 - receiveTimestamp: "2017-12-31T18:37:32.278360657Z"
 - resource: ...
 - severity: "INFO"
 - timestamp: "2017-12-31T18:37:31.566300Z"}

At the bottom, another log entry is partially visible:

```
2017-12-31 10:37:31.666 UTC-8 br.com.siecola.gae_exemplol.controller.HelloController hello: Dog: Hannah (HelloController.java:18)
```

Figura 4.6: Detalhes da mensagem de log

Fique à vontade para explorar e se acostumar com essa

ferramenta, pois ela será muito útil para monitorar a sua aplicação quando ela estiver em execução no GAE.

Conclusão

Neste capítulo, você aprendeu:

- Como preparar o projeto com o Spring Boot para trabalhar com serviços REST;
- Criar um **serviço REST simples**;
- Conceitos básicos de REST e como anotar uma classe e seus métodos para serem entendidos como serviços e operações, com seus caminhos de acesso e parâmetros;
- Um pouco mais sobre o console do GAE;
- Como utilizar o **Postman** para acessar um serviço REST;

No próximo capítulo, você criará um serviço um pouco mais complexo, com várias operações e com um modelo de dados com vários atributos. Dessa forma, você aprenderá mais sobre a construção de serviços REST para o GAE e sobre o seu console.

CAPÍTULO 5

criando um serviço REST completo

No capítulo anterior, você criou seu primeiro serviço REST e publicou no GAE. Ele tinha apenas uma operação, que recebia uma string como parâmetro e retornava outra, concatenada com o parâmetro de entrada. Foi simples, porém útil para testar o projeto e começar a aprender alguns conceitos.

Este capítulo explica a criação de serviços REST um pouco mais complexos no GAE. O que será criado é um serviço de gerenciamento de produtos, mas ainda sem persistência dos dados, para que o foco fique somente na parte da criação do serviço em si. No próximo capítulo, será mostrado como usar o Datastore do GAE.

O serviço de gerenciamento de produtos terá as seguintes operações com os respectivos endereços de acesso:

- `GET /api/products/{code}` : para recuperar um produto de código `{code}` ;
- `GET /api/products` : para recuperar a lista de todos os produtos;
- `POST /api/products` : para inserir um novo produto;

- `PUT /api/products/{code}` : para alterar um produto existente, passando seu código na URL e as alterações no corpo da requisição;
- `DELETE /api/products/{code}` : para apagar o produto com o código passado na URL.

Como pode ser visto, o serviço será o que se chama de CRUD (*Create, Read, Update and Delete*) de produtos.

O formato de dados usado é o JSON, muito difundido para utilização com serviços REST, por ser mais leve e de fácil interpretação.

FORMATO JSON

Se você não está acostumado com o formato JSON, acesse seu site (<http://json.org/>) para aprender mais sobre como os dados são representados.

Um outro site interessante para ajudar na compreensão de dados em formato JSON e também ajudar a montar representações, principalmente de objetos complexos, é o Online JSON Viewer (<http://jsonviewer.stack.hu/>).

Esse projeto começará com um código bem simples, de fácil entendimento para quem está começando. Porém, evoluirá para um modelo bem avançado, utilizando técnicas modernas proporcionadas por projetos com o Spring Boot.

5.1 CRIANDO O MODELO DE PRODUTOS

Para criar o serviço e as operações listadas, é necessário primeiramente criar o modelo de produtos. Ele será parte do contrato que define como os dados serão trafegados nesse serviço. Para começar, crie um novo pacote na aplicação `gae_exemplo1`, com o nome de `br.com.siecola.gae_exemplo1.model`.

Nesse pacote, serão criados todos os modelos usados pelos serviços da aplicação. A organização desses modelos em um único pacote é puramente para fins de organização da estrutura do código do projeto. Dentro desse novo pacote, crie a classe do modelo de produtos, com o nome de `Product`.

Na classe `Product`, crie seus atributos e os `getters` e `setters`, como no trecho de código a seguir:

```
import java.io.Serializable;

public class Product implements Serializable {

    private String productID;
    private String name;
    private String model;
    private int code;
    private float price;

    //getters and setters
}
```

Esse será o modelo do produto utilizado pelo serviço de gerenciamento de produtos. Todas as operações usarão essa classe, logo, uma alteração aqui afeta diretamente a compatibilidade desse serviço.

No capítulo *Armazenando dados no Google Cloud Datastore*, o

mesmo modelo de produto será usado para persistir suas informações no Datastore.

5.2 CRIANDO A CLASSE DO SERVIÇO

Agora, é necessário criar o serviço de gerenciamento de produtos. No momento, ele será responsável por expor as operações desse serviço. Para começar, no pacote `br.com.siecola.gae_exemplo1.controller`, crie a classe `ProductController` para implementar o novo serviço de CRUD de produtos.

Coloque a anotação `@RestController` na declaração da classe, para indicar ao Spring que ela implementa um serviço REST. Além disso, adicione também a anotação `@RequestMapping` com o caminho no qual esse serviço estará disponível, como no trecho a seguir:

```
import br.com.siecola.gae_exemplo1.model.Product;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.List;

@RestController
@RequestMapping(path="/api/products")
public class ProductController {

}
```

Isso fará com que o serviço de gerenciamento de produtos seja acessado pela URL `/api/products`.

Agora crie o método privado `createProduct`, que será

utilizado somente para criar um produto *dummy* a partir de seu código, pois até o momento não há uma tabela onde os produtos são armazenados.

```
private Product createProduct (int code) {  
    Product product = new Product();  
    product.setProductID(Integer.toString(code));  
    product.setCode(code);  
    product.setModel("Model " + code);  
    product.setName("Name " + code);  
    product.setPrice(10 * code);  
    return product;  
}
```

A ideia aqui é apenas mostrar como criar as operações e interagir com objetos complexos, como parâmetros de entrada e saída.

Operação para recuperar um produto pelo código

Com o básico preparado nessa classe, agora é o momento de passar a criar os métodos que vão expor as operações do serviço de gerenciamento de produtos. A primeira operação será a de retornar um produto pelo seu código.

Por isso, crie o método `getProduct` para implementar tal operação, recebendo o código do produto como parâmetro do caminho da URL da requisição. Veja o trecho a seguir:

```
@GetMapping("/{code}")  
public ResponseEntity<Product> getProduct(@PathVariable int code)  
{  
    Product product = createProduct(code);  
    return new ResponseEntity<Product>(product, HttpStatus.OK);  
}
```

Essa operação de recuperar um produto específico,

implementada pelo método `getProduct`, será acessada pela URI `/api/products/{code}`. O parâmetro `{code}`, passado na URL, representa o código do produto. O retorno dessa operação é um objeto complexo do tipo `Product` e será representado no formato JSON no corpo da resposta à requisição, como no exemplo a seguir:

```
{  
    "productID": "3",  
    "name": "Nome 3",  
    "model": "Model 3",  
    "code": 3,  
    "price": 30.0  
}
```

A anotação do método é a mesma usada no serviço de teste, do capítulo anterior, que define o caminho da operação bem como seu parâmetro. O método recebe o código do produto desejado como parâmetro pela URL. Ele também possui um retorno, que nesse caso é um objeto `ResponseType`.

Ao longo de todo o livro, você verá que trabalhar com esse tipo de objeto, em retornos de operações, torna o código mais simples e mais versátil, já que você pode passar a informação desejada juntamente a um código de retorno.

A primeira linha do método simplesmente cria um objeto em memória do tipo `Product`, para ser retornado pela operação. A última linha do método cria um objeto do tipo `ResponseEntity`, passando o produto criado, juntamente ao código de retorno `HTTP 200 OK`. Isso indica que a operação foi realizada com sucesso.

Com o método criado, execute a aplicação e teste-a com o Postman. Nesse momento, você pode criar uma outra aba nele e salvá-la com o nome que desejar. Veja um exemplo de como fica,

buscando pelo produto de código 1:

The screenshot shows the Postman interface with the following details:

- Request URL:** {{address}}/api/products/1
- Method:** GET
- Headers:** Authorization (with a placeholder value), Headers tab selected.
- Body:** Body tab selected, showing a JSON response:

```
1 {  
2   "productID": "1",  
3   "name": "Name 1",  
4   "model": "Model 1",  
5   "code": 1,  
6   "price": 10  
7 }
```
- Response Status:** 200 OK
- Time:** 139 ms
- Size:** 285 B

Figura 5.1: Recuperar produto pelo código

Repare que o corpo da resposta é exatamente a representação do modelo `Product` em formato JSON. Veja também que o código de resposta, `HTTP 200 OK`, é o que foi definido como segundo parâmetro da resposta do objeto do tipo `ResponseEntity` (do método `getProduct`).

Operação para listar todos os produtos

Vamos continuar com o exemplo. Crie a operação para retornar todos os produtos em uma lista, como no trecho a seguir, ainda usando os produtos criados apenas em memória:

```
@GetMapping  
public ResponseEntity<List<Product>> getProducts() {  
    List<Product> products = new ArrayList<>();  
    for (int j = 1; j <= 5; j++) {  
        products.add(createProduct(j));  
    }  
    return ResponseEntity.ok(products);  
}
```

```
    }

    return new ResponseEntity<List<Product>>(products, HttpStatus.OK);
}
```

Essa operação, que não recebe nenhum parâmetro, pode ser acessada pela URL `api/products` com o método HTTP GET . Ela devolve uma lista de produtos criados no momento de sua chamada. O formato devolvido será uma lista no formato JSON, semelhante ao exemplo a seguir:

```
[  
  {  
    "productID": "1",  
    "name": "Name 1",  
    "model": "Model 1",  
    "code": 1,  
    "price": 10.0  
},  
  {  
    "productID": "2",  
    "name": "Name 2",  
    "model": "Model 2",  

```

Execute a sua aplicação e teste pelo Postman. Você verá que a resposta será semelhante a essa, porém com 5 produtos na lista, que foram criados pelo método `getProducts` .

Operação para inserir um produto

A operação para inserir um novo produto é bem diferente, pois ela deve recebê-lo como parâmetro no método que implementa tal operação. Veja como ela deverá ficar no exemplo a seguir:

```
@PostMapping  
public ResponseEntity<Product> saveProduct(@RequestBody Product product) {  
    product.setProductID(Integer.toString(product.getCode()));  
  
    return new ResponseEntity<Product>(product, HttpStatus.CREATED);  
}
```

Obviamente, o tratamento desse método não faz nada útil. O intuito aqui é apenas demonstrar como deve ficar a declaração do método com sua anotação e também o retorno da criação do produto, com o código de status HTTP 201 Created .

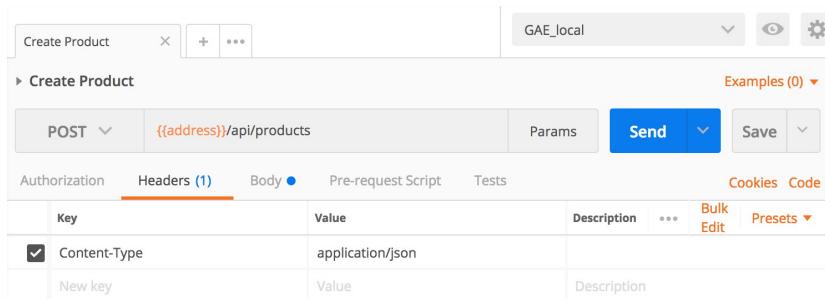
Repare na anotação `PostMapping` , na declaração do método. Ela indica que essa operação deve ser acessada pelo verbo `HTTP POST` . Da mesma forma, há uma nova anotação no parâmetro `Product product` que o método recebe, a `@RequestBody` . Ela indica que o conteúdo do parâmetro de entrada deverá ser extraído do corpo da mensagem de requisição.

O modelo do produto a ser inserido no corpo da requisição deve ser como mostra o trecho adiante:

```
{  
    "name": "Name 6",  
    "model": "Model 6",  
    "code": 6,  
    "price": 60.0  
}
```

Além disso, é necessário adicionar um cabeçalho na requisição, informando o modelo de dados da informação presente em seu corpo. Tal cabeçalho é o seguinte: Content-Type: application/json .

Isso diz ao servidor que a informação presente no corpo da requisição deve ser tratada como um JSON. Veja como deve ficar no Postman, em relação a essa configuração do cabeçalho:



The screenshot shows the Postman interface for a 'Create Product' request. The top bar includes 'Create Product' (with a close button), a '+' icon, and an '***' icon. To the right are dropdowns for 'GAE_local' and other settings. Below the bar, the request name is 'Create Product' and the method is 'POST' to '({{address}})/api/products'. There are 'Params', 'Send', and 'Save' buttons. The main area shows the 'Headers (1)' tab selected, with a single entry: 'Content-Type' set to 'application/json'. Other tabs include 'Body', 'Pre-request Script', and 'Tests'. A 'Cookies' and 'Code' section is also visible.

Figura 5.2: Configurando o cabeçalho no Postman

Agora, veja como você deve colocar a informação no corpo da requisição, inserindo-a na aba Body do Postman:

The screenshot shows the Postman interface for a 'Create Product' request. The top bar includes tabs for 'Create Product', 'GAE_local', and various icons. The main area shows a 'POST' request to '({{address}})/api/products'. The 'Body' tab is selected, containing JSON data:

```
1 {  
2     "productID": "1",  
3     "name": "Name 1",  
4     "model": "Model 1",  
5     "code": 1,  
6     "price": 10  
7 }
```

Below the body, the status is 'Status: 201 Created' with a time of '5 ms' and a size of '290 B'. The 'Pretty' view of the response shows the same JSON data.

Figura 5.3: Criando no produto no Postman

Perceba que você deve colocar as informações do novo produto no formato JSON, no corpo da requisição. Você também deve configurar o verbo `HTTP POST`. Note também que o código de retorno foi o `HTTP 201 Created` =, exatamente o que foi definido na última linha do método `saveProduct` :

```
return new ResponseEntity<Product>(product, HttpStatus.CREATED);
```

Operação para apagar um produto

Agora crie a operação para apagar um produto pelo seu código, como no trecho a seguir:

```
@DeleteMapping(path = "/{code}")
```

```
public ResponseEntity<Product> deleteProduct(@PathVariable("code"
    int code) {
    Product product = createProduct(code);
    return new ResponseEntity<Product>(product, HttpStatus.OK);
}
```

A novidade aqui é a anotação `@DeleteMapping`, que indica que a operação deve ser acessada com o verbo `HTTP DELETE`. O parâmetro `path` dessa anotação nos diz que a operação será acessada pelo endereço raiz do serviço, ou seja, pela URL `api/products/{code}`.

Operação para alterar um produto

Por último, crie a operação de alterar um pedido. Este receberá o código como parâmetro na URL, e o produto com as informações a serem alteradas no corpo da requisição. A operação retorna o produto alterado como resposta:

```
@PutMapping(path = "/{code}")
public ResponseEntity<Product> updateProduct(@RequestBody Product
    product,
                                                @PathVariable("code"
    int code) {
    product.setProductID(Integer.toString(product.getCode()));
    product.setName("New name");
    return new ResponseEntity<Product>(product, HttpStatus.OK);
}
```

A anotação `PutMapping` define que a operação deverá ser acessada com o verbo `HTTP PUT`. Seu atributo `path` define que a URL será `api/products/{code}`. Por fim, o produto alterado é passado no corpo da requisição. Isso significa que você deverá configurar o Postman de forma muito semelhante ao que fez na criação do produto, com apenas duas diferenças:

- Configurar o verbo HTTP para `PUT`;

- Passar o código do produto a ser alterado como parâmetro no caminho da requisição.

O cabeçalho e o corpo da mensagem deverão ser os mesmos em relação à requisição de criação do novo produto.

Como dito anteriormente, esse serviço de gerenciamento não persiste as informações em lugar nenhum, até o momento. Mas no próximo capítulo, será mostrado como isso pode ser feito com o Google Cloud Datastore.

IMPORTS DE BIBLIOTECAS

Se tiver dúvidas sobre quais imports utilizar, consulte o código-fonte do projeto no repositório <https://github.com/siecola/GAEBookV3Exemplo1>.

Em resumo, as novas operações com suas URLs de acesso são:

- GET /api/products/{code} : para recuperar um produto de código {code} ;
- GET /api/products : para recuperar a lista de todos os produtos;
- POST /api/products : para inserir um novo produto;
- PUT /api/products/{code} : para alterar um produto existente, passando seu código na URL e as alterações no corpo da requisição;

- `DELETE /api/products/{code}` : para apagar o produto com o código passado na URL.

Conclusão

Neste capítulo, você aprendeu a como criar um serviço de CRUD de produtos, com um modelo de dados complexo, e as anotações que podem ser usadas para definir o caminho e os verbos HTTP usados para acessar as operações do serviço. Também vimos qual é o formato de dados esperado como parâmetro de entrada e o que será devolvido como resposta.

No próximo capítulo, você aprenderá a usar o Google Cloud Datastore, para criar uma tabela de produtos e poder persisti-los, em vez de utilizar uma lista em memória.

CAPÍTULO 6

ARMAZENANDO DADOS NO GOOGLE CLOUD DATASTORE

Os serviços de gerenciamento de produtos, criados no projeto `gae_exemplo1`, demonstraram como é possível trabalhar com objetos complexos – por exemplo, parâmetros de entrada e saída das operações de um serviço REST. Isso é algo importante de ser bem entendido. Entretanto, não utilizamos nenhum mecanismo de persistência de dados. A não ser que você esteja construindo um serviço que apenas realiza cálculos ou consulta outro serviço, realizar operações com um banco de dados para salvar estados ou entidades é quase sempre necessário.

A plataforma do Google App Engine possui algumas formas diferentes de persistir dados, que variam de acordo com o seu tamanho, custo de armazenamento, desempenho e compatibilidade com outros sistemas de armazenamento.

6.1 O QUE É O GOOGLE CLOUD DATASTORE

Este capítulo foca no Google Cloud Datastore, um banco de dados NoSQL com grande desempenho e altamente escalável.

Aqui vão algumas características importantes desse serviço de armazenamento do Google:

- Transações atômicas;
- Alta disponibilidade de leituras e escritas;
- Alto desempenho e escalabilidade;
- Mecanismo flexível de consulta de dados;
- Sem necessidade de gerenciamento de servidores ou aplicações de banco de dados.

Para você ter uma ideia melhor de como funciona o Google Cloud Datastore, segue uma tabela comparativa com o mundo dos bancos de dados relacionais, para você se acostumar com os termos usados nele. Essa comparação foi retirada do site de documentação do Google Cloud Datastore (<https://cloud.google.com/datastore/docs/concepts/overview>):

Conceito	Datastore	Banco de dados relacional
Categoria do objeto	Tipo (<i>Kind</i>)	Tabela (<i>Table</i>)
Um objeto	Entidade (<i>Entity</i>)	Linha (<i>Row</i>)
Dado individual de um objeto	Propriedade (<i>Property</i>)	Campo (<i>Field</i>)
Identificação única de um objeto	Chave (<i>Key</i>)	Chave primária (<i>Primary Key</i>)

É importante aprender esses conceitos do Google Cloud Datastore (*Kind*, *Entity*, *Property* e *Key*), pois, quando você for escrever o código em Java para salvar ou pesquisar um objeto, você precisará saber isso para fazer as devidas associações. Nas seções a seguir, você verá como cada um deles funciona na prática.

A forma de acesso ao Google Cloud Datastore apresentada

neste capítulo é conhecida como *low level API* – ou seja, a forma mais básica e direta de interagir com esse serviço, sem camadas de software ou frameworks. É importante começar aprendendo dessa forma, para entendermos os conceitos com maior profundidade. Existem bibliotecas que abstraem o acesso ao Datastore, que serão vistas em capítulos adiante.

6.2 PREPARANDO O PROJETO PARA TRABALHAR COM DATASTORE

Para demonstração e exemplificação das técnicas de armazenamento e consulta de dados com o Google Cloud Datastore, usaremos o mesmo projeto `gae_exemplo1` e o mesmo serviço de gerenciamento de produtos. Entretanto, todos os métodos serão alterados para salvar ou buscar os produtos no Google Cloud Datastore.

Em primeiro lugar, comece alterando o modelo de produtos na classe `Product` do pacote `br.com.siecola.gae_exemplo1.model`. Também adicione o atributo `id`, como no trecho a seguir, juntamente ao seu *getter* e *setter*:

```
private long id;
```

A unidade fundamental do sistema de armazenamento com o Datastore é uma *Entity* que possui uma identidade imutável, representada por uma chave e por parâmetros que podem ser alterados. Uma entidade pode ser criada, apagada, alterada ou recuperada pelo seu identificador, e localizada por meio de consultas de seus outros parâmetros.

Agora, na classe `ProductController`, crie o método privado `productToEntity` para fazer a conversão de um objeto `Product` para um objeto `Entity`, como no trecho a seguir. Ele será usado nos métodos em que for necessário converter um produto (do tipo `Product`) para uma entidade (do tipo `Entity`) para salvar no Datastore.

```
private void productToEntity (Product product, Entity productEntity) {  
    productEntity.setProperty("ProductID", product.getProductID());  
    productEntity.setProperty("Name", product.getName());  
    productEntity.setProperty("Code", product.getCode());  
    productEntity.setProperty("Model", product.getModel());  
    productEntity.setProperty("Price", product.getPrice());  
}
```

NO MOMENTO, A SIMPLICIDADE EM FAVOR DA DIDÁTICA

Por enquanto, esses métodos serão criados na classe `ProductController`, para que os conceitos novos fiquem focados no que é necessário para se trabalhar com o Google Cloud Datastore. Em alguns capítulos mais adiante, você aprenderá outras formas mais avançadas para isolar o `controller` do restante do código de acesso ao Datastore.

De maneira análoga, crie o método privado `entityToProduct` para fazer a conversão de um objeto `Entity` para um objeto do tipo `Product`.

```
private Product entityToProduct (Entity productEntity) {  
    Product product = new Product();  
    product.setId(productEntity.getKey().getId());  
    product.setProductID((String) productEntity.getProperty("Prod
```

```

        uctID"));
        product.setName((String) productEntity.getProperty("Name"));
        product.setCode(Integer.parseInt(productEntity.getProperty("C
ode"))
                .toString()));
        product.setModel((String) productEntity.getProperty("Model"))
;
        product.setPrice(Float.parseFloat(productEntity.getProperty("P
rice"))
                .toString()));
    return product;
}

```

Ele será usado nos métodos onde será necessário buscar uma entidade do Datastore e converter no objeto `Product` a ser retornado nas operações do serviço.

Operação para inserir um produto

Ainda na classe `ProductManager`, altere o método `saveProduct`, que é quem implementa a operação para inserir um novo produto. Dessa vez, o produto que chegar como parâmetro no corpo da requisição HTTP POST será salvo no Google Cloud Datastore com a execução do código a seguir:

```

@PostMapping
public ResponseEntity<Product> saveProduct(@RequestBody Product p
roduct) {

    DatastoreService datastore = DatastoreServiceFactory
        .getDatastoreService();

    Key productKey = KeyFactory.createKey("Products",
                                         "productKey");
    Entity productEntity = new Entity("Products", productKey);

    this.productToEntity (product, productEntity);

    datastore.put(productEntity);
}

```

```
        product.setId(productEntity.getKey().getId());  
  
        return new ResponseEntity<Product>(product, HttpStatus.CREATED);  
    }  

```

A primeira linha do método obtém uma instância do serviço Datastore para ser utilizada – nesse caso, para inserção de um novo produto. A segunda linha instancia uma chave que será usada na criação da entidade produto. Essa chave ficará incompleta até que a entidade seja inserida no Datastore, onde será possível obter a identificação única de sua criação – o que é realizado na penúltima linha do método.

A terceira linha cria uma nova entidade do tipo `Products` (nesse caso, uma analogia ao nome da tabela é válida), com a chave criada na linha anterior. A quarta linha preenche as propriedades da entidade `productEntity` em um esquema chave-valor (que seriam os nomes dos campos e seus valores), e é finalizada pela inserção do novo produto com a penúltima linha do método.

Evidentemente, o trecho mostrado é simples o suficiente para apenas mostrar a essência do serviço Datastore, pois algumas verificações poderiam ser feitas. Por exemplo, poderíamos ver se algum campo mandatório está presente no objeto produto recebido, ou se já há algum cadastrado com o mesmo código.

Para resumir e fazer uma relação com a tabela comparativa da seção anterior, tem-se que:

- O **tipo** que está sendo colocado no Datastore é o de nome `Products` ;
- A **entidade** é o produto recebido como parâmetro do método `saveProduct` ;

- As **propriedades** possuem os nomes: `ProductID` , `Name` , `Code` , `Model` , `Price` ;
- O valor da **chave** da entidade salva foi atribuída ao atributo `id` , de `product` .

Operação para listar todos os produtos

Para listar todos os produtos cadastrados, precisamos criar uma consulta no Datastore para que ele retorne todas as entidades do tipo `Products` , baseado ou não em uma ordenação por uma de suas propriedades.

No exemplo a seguir, isso foi feito com base na propriedade `Code` , ordenando a lista de produtos de forma crescente de acordo com esse atributo. O método `getProducts` deve então ficar da seguinte forma:

```
@GetMapping
public ResponseEntity<List<Product>> getProducts() {
    List<Product> products = new ArrayList<>();
    DatastoreService datastore = DatastoreServiceFactory
        .getDatastoreService();

    Query query;
    query = new Query("Products").addSort("Code",
        Query.SortDirection.ASCEN
DING);

    List<Entity> productsEntities = datastore.prepare(query).as
List(
        FetchOptions.Builder.withDefaults());

    for (Entity productEntity : productsEntities) {
        Product product = entityToProduct(productEntity);

        products.add(product);
    }
}
```

```
        return new ResponseEntity<List<Product>>(products, HttpStatus.OK);
    }
```

O que aparece de novo nesse método é justamente a busca de todas as entidades, realizada pelo objeto do tipo `Query`, que usa a propriedade `Code` para organizá-las de forma crescente. Com isso, o método `asList` do objeto `datastore` retorna a lista de todas as entidades (produtos) de nome `Products`.

Operação para recuperar um produto pelo código

Para recuperar apenas um produto do `Datastore`, usando seu código como parâmetro de pesquisa, basta alterar o método `getProduct`, como no trecho a seguir:

```
@GetMapping("/{code}")
public ResponseEntity<Product> getProduct(@PathVariable int code)
{
    DatastoreService datastore = DatastoreServiceFactory
        .getDatastoreService();

    Query.Filter codeFilter = new Query.FilterPredicate("Code",
        Query.FilterOperator.EQ
UAL, code);

    Query query = new Query("Products").setFilter(codeFilter);

    Entity productEntity = datastore.prepare(query).asSingleEntity();

    if (productEntity != null) {
        Product product = entityToProduct(productEntity);

        return new ResponseEntity<Product>(product, HttpStatus.OK
    );
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
```

Para localizar uma entidade específica a partir de uma de suas propriedades, é necessário utilizar um filtro. Nesse exemplo, criaremos um filtro para a propriedade `code`, para que seja igual ao código passado como parâmetro pela URL. Após isso, faremos a busca, requisitando somente uma entidade (produto).

Caso ela seja encontrada, o objeto `Product` é preenchido com as informações e retornado como resposta, com o código `HTTP 200 OK` de resposta. Porém, caso não seja, o código `HTTP 404 - Not Found` é retornado com a resposta para indicar que ele não foi encontrado com o código fornecido.

Operação para alterar um produto

Para alterar uma entidade específica, devemos alterar o método `updateProduct`, como no trecho a seguir. Ele é semelhante ao método para buscar somente uma entidade, em termos da consulta para localizar tal produto a partir do código. A partir daí, as alterações são aplicadas ao produto, e este é salvo novamente no Datastore:

```
@PutMapping(path = "/{code}")
public ResponseEntity<Product> updateProduct(@RequestBody Product
product,
                                              @PathVariable("code"
int code) {
    DatastoreService datastore = DatastoreServiceFactory
        .getDatastoreService();

    Query.Filter codeFilter = new Query.FilterPredicate("Code",
                                                       Query.FilterOperator.
EQUAL, code);

    Query query = new Query("Products").setFilter(codeFilter);

    Entity productEntity = datastore.prepare(query).asSingleEnt
ity();
```

```

    if (productEntity != null) {
        productToEntity (product, productEntity);

        datastore.put(productEntity);

        product.setId(productEntity.getKey().getId());
        return new ResponseEntity<Product>(product, HttpStatus.OK
    );
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}

```

Repare na linha `datastore.put(productEntity);`. Ele é o responsável por salvar o produto no Datastore novamente, com as alterações mandadas pelo modelo recebido no corpo da requisição `HTTP PUT`.

Operação para apagar um produto

Por último, para apagar uma entidade, basta modificarmos o método `deleteProduct`, como o trecho seguinte:

```

@DeleteMapping(path = "/{code}")
public ResponseEntity<Product> deleteProduct(
    @PathVariable("code") int code) {

    DatastoreService datastore = DatastoreServiceFactory
        .getDatastoreService();

    Query.Filter codeFilter = new Query.FilterPredicate("Code",
        Query.FilterOperator
    .EQUAL, code);

    Query query = new Query("Products").setFilter(codeFilter);

    Entity productEntity = datastore.prepare(query).asSingleEnt
ity();

    if (productEntity != null) {

```

```
        datastore.delete(productEntity.getKey());  
  
        Product product = entityToProduct(productEntity);  
  
        return new ResponseEntity<Product>(product, HttpStatus.OK  
    );  
} else {  
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);  
}  
}
```

Repare que o produto é localizado da mesma forma como vimos nos métodos para alteração e busca por código. A diferença é que, caso o produto seja encontrado, o comando `datastore.delete(productEntity.getKey());` será executado para excluí-lo do Datastore. Caso contrário, a mensagem HTTP 404 Not Found será enviada como resposta.

IMPORTS DE BIBLIOTECAS

Se tiver dúvidas sobre quais imports utilizar, consulte o código-fonte do projeto, no repositório <https://github.com/siecola/GAEBookV3Exemplo1>.

6.3 TESTANDO NA MÁQUINA LOCAL

Para testar as alterações na sua máquina local de desenvolvimento utilizando o REST Console, basta seguir os mesmos passos mostrados no capítulo anterior, acessando os serviços pelos métodos e pelas URLs seguintes:

- GET /api/products/{code} : para recuperar um produto

- de código {code} ;
- GET /api/products : para recuperar a lista de todos os produtos;
 - POST /api/products : para inserir um novo produto;
 - PUT /api/products/{code} : para alterar um produto existente, passando seu código na URL e as alterações no corpo da requisição;
 - DELETE /api/products/{code} : para apagar o produto com o código passado na URL.

A única diferença é que o modelo do produto, retornado pelas operações, possui o atributo `id`, e ficará como mostra o JSON a seguir. Porém, não é necessário adicionar esse atributo nas operações para inserção e alteração de produtos, pois ele será gerado pelo Datastore quando o produto for inserido.

```
{  
    "id": 5629499534213120,  
    "productID": "1",  
    "name": "Nome 1",  
    "model": "Model 1",  
    "code": 1,  
    "price": 10.0  
}
```

RODANDO A APLICAÇÃO NO GAE

Publique a aplicação no GAE e faça alguns testes com o serviço de produtos. Faça inserções e pesquisas.

6.4 ÍNDICES DO DATASTORE

O Google Cloud Datastore trabalha com índices para cada consulta que é feita pela aplicação. Para tornar as buscas mais rápidas, você pode especificar esses índices por meio de um arquivo de configuração, chamado `datastore-indexes.xml`. Ele deve ser criado na pasta `WEB-INF`.

Esses índices são gerados com 3 características principais:

- Tipo da entidade a ser indexada;
- Propriedade da entidade usada na busca;
- Ordenação das entidades retornadas na busca.

No exemplo mostrado, a aplicação realiza consultas de entidades do tipo `Product`, baseadas na propriedade `Code` e ordenada de forma crescente (no caso do método `getProducts`). Para esse tipo de busca – que utiliza apenas uma propriedade como base de pesquisa –, não é necessário criar índices.

Porém, caso a busca também precisasse de uma ordenação pela propriedade `Model`, isso seria necessário, por exemplo. Dessa forma, para tornar buscas com mais de uma propriedade mais rápidas, basta criar o arquivo de configuração de índices, conforme os passos a seguir:

1. Crie o arquivo `datastore-indexes.xml` na pasta `WEB-INF`.
2. Crie o esqueleto desse arquivo, como mostra o trecho a seguir:

```
<?xml version="1.0" encoding="utf-8"?>
<datastore-indexes autoGenerate="true">
```

```
</datastore-indexes>
```

3. Dentro da tag `datastore-indexes`, crie o índice para a consulta realizada:

```
<datastore-index kind="Products" ancestor="false">
  <property name="Code" direction="asc" />
  <property name="Model" direction="asc" />
</datastore-index>
```

Sendo assim, cada elemento `datastore-index` representa um índice que auxilia o Datastore na busca para esse tipo de entidade, por meio das propriedades informadas.

4. Publique a aplicação novamente no Google App Engine para visualizar as entidades e o índice que foi criado. Lembre-se de marcar todas as opções da seção Deployment Parameters na tela de publicação, no GAE do IntelliJ IDEA.

Exercício proposto

Para você praticar um pouco mais sobre como interagir com os dados armazenados no Google Cloud Datastore pela sua aplicação, altere o serviço de gerenciamento de produtos do projeto `gae_exemplo1` para que ele realize as seguintes verificações:

- Na operação de alterar o produto, exija que o atributo `id` esteja presente e preenchido com um valor válido. Caso ele não tenha sido enviado, responda com `HTTP 400 Bad Request`.
- Antes de cadastrar ou alterar um produto, verifique se já não há outro produto cadastrado com o mesmo código. Caso já exista algum com o mesmo código, as operações

deverão responder com o código HTTP 400 Bad Request .

A resolução desse exercício está no repositório do livro, no endereço: <https://github.com/siecola/GAEBookV3Exemplo1>.

6.5 ADMINISTRANDO O DATASTORE NO GAE

O console do Google App Engine possui uma parte dedicada para administração do Datastore. Ele pode ser acessado ao clicarmos no menu suspenso no canto superior esquerdo da tela, na seção Datastore da divisão Storage . Essa seção do console do GAE fornece as ferramentas vistas a seguir.

Dashboard

Essa é opção do console de administração do Google Cloud Datastore. Nela você pode obter estatísticas sobre o acesso aos tipos e às entidades. Essas estatísticas são geradas a cada 24 horas.

Gerenciamento e pesquisa de entidades

Essa ferramenta pode ser acessada na opção Entities . Com ela, é possível:

- Visualizar os tipos existentes.
- Listar as entidades cadastradas de cada tipo, como mostra a figura:

[Query by kind](#) [Query by GQL](#)

Kind

Products

Filter entities

	Name/ID	Code	Model	Name	Price	ProductID
<input type="checkbox"/>	id=5629499534213120	1	Model 1	Nome 1	10.0	1
<input type="checkbox"/>	id=5707702298738688	2	Model 2	Nome 2	20.0	2

Figura 6.1: Listando entidades no Datastore

- Filtrar as entidades a partir de qualquer uma de suas propriedades:

[Query by kind](#) [Query by GQL](#)

Kind

Products

Filter entities

Code

is an integer

equal to

1

+ -

Apply filters
Clear filters

	Name/ID	Code	Model	Name	Price	ProductID
<input type="checkbox"/>	id=5629499534213120	1	Model 1	Nome 1	10.0	1

Figura 6.2: Filtrando entidades no Datastore

- Apagar uma entidade, selecionando-a e clicando no botão Delete .
- Alterar uma entidade, clicando em seu ID:

[←](#) Edit entity [REFRESH](#)

Namespace: [default]

Kind: Products

ID: Products name:productKey > Products id:5629499534213120

Key literal: Key('Products', 'productKey', 'Products', 5629499534213120)

Properties

Name	Type	Value	Indexed
Price	Floating point number	10	<input checked="" type="checkbox"/> x
Code	Integer	1	<input checked="" type="checkbox"/> x
ProductID	String	1	<input checked="" type="checkbox"/> x
Name	String	Nome 1	<input checked="" type="checkbox"/> x
Model	String	Model 1	<input checked="" type="checkbox"/> x

[+ Add property](#)

[Save](#) [Cancel](#)

Figura 6.3: Editando uma entidade no Datastore

- Realizar buscas usando o *Google Query Language* (ou GQL). Para saber mais, consulte https://cloud.google.com/datastore/docs/apis/gql/gql_reference.

[Query by kind](#) [Query by GQL](#)

```
SELECT * FROM Products
```

[Run query](#)[Clear query](#)[GQL query help](#)

	Name/ID	Code	Model	Name	Price	ProductID
<input type="checkbox"/>	id=5629499534213120	1	Model 1	Nome 1	10.0	1
<input type="checkbox"/>	id=5707702298738688	2	Model 2	Nome 2	20.0	2

Figura 6.4: Filtrando entidades no Datastore com GQL

- Criar novas entidades:

[← Create entity](#)Namespace [?](#)Kind [?](#)Key identifier [?](#)

Properties

Name	Type	Value	Indexed
Code	= Integer		<input checked="" type="checkbox"/> x
Model	= String		<input checked="" type="checkbox"/> x
Name	= String		<input checked="" type="checkbox"/> x
Price	= Floating point number		<input checked="" type="checkbox"/> x
ProductID	= String		<input checked="" type="checkbox"/> x

[+ Add property](#)[Create](#)[Cancel](#)

Figura 6.5: Criando entidades no Datastore com GQL

Visualização dos índices de pesquisa

Nesta seção, é possível visualizar os índices criados para cada entidade:

Indexes

Below are the composite indexes for this application. These indexes are managed in your app's index configuration file. [Learn more](#)

Kind ^	Indexes	Size	Entries	Status
Product	Code ▲ + Model ▲	—	—	Serving

Figura 6.6: Visualizando os índices

Veja que esse é um índice criado para o tipo `Product`, para uma busca por ordem crescente com os campos `Code` e `Model`.

Administração

Nesta seção, você pode fazer backup, restaurar, copiar e apagar entidades de forma massiva, como também desabilitar as operações de escrita no Datastore, na instância onde você está administrando.

VERSÃO SIMPLIFICADA DO CONSOLE LOCAL DO DATASTORE

Há uma versão simplificada do console de administração do Datastore que pode ser acessada na sua máquina de desenvolvimento, quando a aplicação está em execução, pelo endereço: http://localhost:8888/_ah/admin/datastore.

Conclusão

Neste capítulo, você aprendeu uma importante tarefa: como persistir dados no Google Cloud Datastore. Além disso, você também aprendeu técnicas de como pesquisar entidades salvas no Datastore, validações e como administrar tudo por meio do console de administração do GAE.

No próximo capítulo, você verá como gerar mensagens de log por meio de uma aplicação no GAE, e como visualizar essas mensagens em seu console de administração.

CAPÍTULO 7

GERANDO MENSAGENS DE LOG

Por muitas vezes, é necessário depurar ou registrar eventos em aplicações que estão sendo executadas no GAE, para que possamos analisá-los em um momento futuro. Em algumas situações, apenas as estatísticas, os gráficos de requisições, o trace e os logs gerados e exibidos pelo console do GAE (que são muito bons) não são suficientes para descobrir a causa raiz de um problema.

Para isso, é necessário lançar mão da geração de mensagens log pela aplicação. Então, essas mensagens poderão ser visualizadas no console do GAE. Para geração de logs pela aplicação, podemos usar o `java.util.logging.Logger`, como será mostrado neste capítulo.

7.1 CONFIGURANDO O PROJETO PARA GERAÇÃO DE LOGS

Neste capítulo, você gerará mensagens de log nas operações do serviço de gerenciamento de produtos, assim, elas serão visualizadas no console do GAE. Mas antes, é necessário realizar algumas configurações no projeto `gae_exemplo1`.

Abra o arquivo `src/main/webapp/WEB-INF/logging.properties`, onde são feitas as configurações para a geração, e veja que há a propriedade `.level`, configurada inicialmente com o valor padrão `INFO`.

```
.level = INFO
```

Há 5 níveis de severidade de log que podem ser configurados nessa propriedade:

1. DEBUG
2. INFO
3. WARNING
4. ERROR
5. CRITICAL

O nível `CRITICAL` é o mais severo e o `DEBUG`, o de menor severidade. Isso significa que é possível gerar mensagens com esses níveis, para depois serem filtrados na hora de gravação no mecanismo de armazenamento dessas mensagens de logs.

Nesses 5 níveis, há uma ordem na filtragem para que as mensagens apareçam. Por exemplo, configurando o nível de log como `WARNING`, as mensagens geradas como `INFO` e `DEBUG` não aparecerão, ao contrário das geradas com `WARNING`, `ERROR` e `CRITICAL`.

Na interface de monitoramento de logs do console do GAE, é possível selecionar as mensagens pelo seu nível de severidade, como veremos mais adiante. Como em qualquer outra aplicação, é muito importante saber definir o nível de severidade na geração das mensagens de log, bem como na configuração de filtragem, pois, em um primeiro momento, muitas mensagens sendo geradas

podem afetar o desempenho geral da aplicação.

No GAE, ainda há dois fatores que devem ser considerados, já que existem limites no uso da plataforma no modo gratuito, como:

- **Tempo de armazenamento das mensagens:** as mensagens de log ficam armazenadas por até 90 dias na plataforma. Caso sua aplicação esteja habilitada para cobranças (ou seja, não está no modo gratuito), essas mensagens poderão ficar armazenadas por até 365 dias.
- **Tamanho máximo do armazenamento das mensagens:** no modo gratuito, há um limite máximo de 1 GB de armazenamento das mensagens de log. Caso chegue ao limite, as primeiras mensagens vão sendo apagadas à medida que novas aparecem, em um esquema de reciclagem das mais antigas.

Nesse arquivo, também é possível definir o nível de log específico para classes pertencentes a um pacote, por exemplo:

```
br.com.siecola.gae_exemplo1.controller.level = DEBUG
```

No arquivo `src/main/webapp/WEB-INF/appengine-web.xml`, existe uma configuração responsável por indicar ao mecanismo de log qual arquivo de configuração deve ser utilizado:

```
<system-properties>
  <property name="java.util.logging.config.file" value="WEB-INF
/logging.properties"/>
</system-properties>
```

Repare que o arquivo é aquele citado no item anterior, onde pode ser feita a configuração do nível das mensagens de log a serem geradas.

7.2 MÉTODOS PARA GERAÇÃO DE LOGS

Agora que você já entendeu um pouco como funciona o mecanismo de log do GAE, é hora de entender como fazer para gerar tais mensagens no código. Como exemplo, será utilizado o serviço de gerenciamento de produtos, no qual serão colocados comandos dentro dos métodos da classe `ProductController`.

Dessa forma, toda vez que uma operação desse serviço for acessada, a aplicação vai gerar uma mensagem de log, que poderá ser visualizada no console do GAE, como veremos na próxima seção deste capítulo. A primeira coisa a fazer é importar a biblioteca `java.util.logging.Logger` na classe `ProductController`:

```
import java.util.logging.Logger;
```

Para utilizar os métodos de geração de log da classe `Logger`, crie um atributo privado e estático dentro da classe `ProductController`:

```
private static final Logger log = Logger.getLogger("ProductController");
```

A chamada `Logger.getLogger("ProductController")` cria ou localiza uma instância do mecanismo de log com o nome fornecido. Nesse caso, foi usado o próprio nome da classe.

Pronto! Agora é só usar o atributo `log` com um dos seguintes métodos:

```
log.finest("Mensagem de nível DEBUG");
log.finer("Mensagem de nível DEBUG");
log.fine("Mensagem de nível DEBUG");
log.config("Mensagem de nível DEBUG");
log.info("Mensagem de nível INFO");
```

```
log.warning("Mensagem de nível WARNING");  
log.SEVERE("Mensagem de nível ERROR");
```

Veja que as mensagens de nível `DEBUG` podem ser geradas pelos métodos `finest`, `finer`, `fine` ou `config`. Para a mensagem de nível `ERROR`, devemos usar o método `severe`. Já as mensagens de nível `INFO` e `WARNING` possuem os seus métodos próprios.

Ainda, ao serem utilizados os métodos de escrita na saída padrão, a mensagem será encarada como do nível `INFO`:

```
System.out.println("Mensagem INFO");
```

E ao usarmos métodos de escrita na saída de erro, a mensagem será encarada como `WARNING`:

```
System.err.println("Mensagem WARNING");
```

As mensagens do nível `CRITICAL` são reservadas para exceções que não forem tratadas. Ou seja, quando isso acontece, o mecanismo de log do GAE encarárá essas mensagens com o nível `CRITICAL`.

Vale lembrar de que as mensagens que efetivamente são gravadas no mecanismo de log serão aquelas que estiverem com um nível de log acima do filtro configurado na propriedade `.level` do arquivo `src/main/webapp/WEB-INF/logging.properties`.

Veja a seguir como pode ficar o método `deleteProduct`, recheado de mensagens de log para registrar tudo o que pode acontecer quando ele é chamado, com mensagens de níveis diferentes (dependendo da gravidade ou razão do evento).

```
@DeleteMapping(path = "/{code}")
```

```

public ResponseEntity<Product> deleteProduct(
    @PathVariable("code") int code) {

    //Mensagem 1 - DEBUG
    log.fine("Tentando apagar produto com código=[ " + code + " ]");

    DatastoreService datastore = DatastoreServiceFactory
        .getDatastoreService();

    Query.Filter codeFilter = new Query.FilterPredicate("Code",
        Query.FilterOperator.EQUAL, code);

    Query query = new Query("Products").setFilter(codeFilter);

    Entity productEntity = datastore.prepare(query).asSingleEntity();

    if (productEntity != null) {
        datastore.delete(productEntity.getKey());

        //Mensagem 2 - INFO
        log.info("Produto com código=[ " + code + " ] " +
            "apagado com sucesso");

        Product product = entityToProduct(productEntity);

        return new ResponseEntity<Product>(product, HttpStatus.OK);
    } else {
        //Mensagem 3 - ERROR
        log.severe ("Erro ao apagar produto com código=[ " + code +
                    " ]. Produto não encontrado!");

        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}

```

No início do método, veja que a primeira mensagem de log (Mensagem 1 - DEBUG) foi gerada com o método `fine` , o que significa que ela terá o nível `DEBUG` no GAE. A mensagem que diz

que o produto foi apagado com sucesso (Mensagem 2 - INFO) foi gerada com o nível INFO . Já a última (Mensagem 3 - ERROR), antes do lançamento da exceção lançada caso o produto não seja encontrado, foi construída com o severe , o que gera uma mensagem com o nível ERROR .

No caso desse método, haveria 3 possibilidades de configuração do filtro de severidade das mensagens:

- FINE : todas as 3 mensagens apareceriam – pois é o nível mais baixo, correspondente ao DEBUG –, o que faz com que todas as mensagens com nível acima desta sejam gravadas no log. Isso faz com que muitas mensagens apareçam, por isso é interessante configurar esse nível apenas para o pacote que você está depurando.
- INFO : somente as mensagens 2 e 3 seriam gravadas no log.
- ERROR : somente a mensagem 3 seria gravada no log, pois as outras (INFO e DEBUG) possuem níveis de severidade menor do que ERROR .

ESCOLHA DO NÍVEL DE SEVERIDADE DAS MENSAGENS

A escolha do nível de severidade das mensagens não é totalmente objetiva e pode variar de programador para programador. Entretanto, é importante ter bom senso na hora de fazer essa definição, levando em conta o que já foi dito neste capítulo, como quantidade de mensagens e o tamanho que elas vão ocupar em disco.

Exercício proposto

Para praticar mais as funcionalidades de geração de logs em uma aplicação no GAE, altere os outros métodos que implementam o serviço de produtos para gerarem mensagens de log – principalmente quando gerarem exceção. Execute a aplicação na sua máquina local de desenvolvimento e veja as mensagens sendo geradas na aba console do IntelliJ IDEA.

Altere o nível de severidade da configuração no arquivo `src/main/webapp/WEB-INF/logging.properties`, e veja seus efeitos no console de execução do IntelliJ IDEA.

7.3 VISUALIZANDO AS MENSAGENS DE LOG NO GAE

Nesta seção, você aprenderá a como visualizar as mensagens no console do GAE. Mas, antes, publique a aplicação e faça alguns testes com as operações em que você colocou as mensagens – por exemplo, a operação de apagar produtos.

No console do GAE, acesse o menu no canto superior direito, clicando na opção `Logging` (que fica no conjunto `StackDriver`). Logo no topo da página na qual os logs são exibidos, existem algumas opções para filtrar as mensagens que foram geradas pela aplicação. São elas:

- **Versão da aplicação:** aqui você pode selecionar, dentre as opções de aplicações do Google App Engine, qual versão você quer exibir os logs, como mostra a figura a seguir:

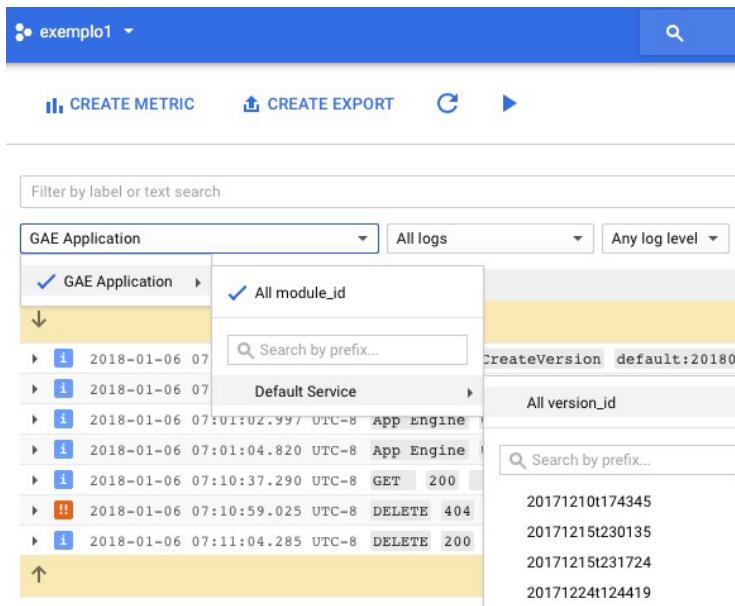


Figura 7.1: Logs no console do GAE

Repare que, como foram publicadas várias versões dessa aplicação, elas aparecem para que você possa selecioná-las. Aqui você também pode optar por ver todas as versões, selecionando a opção `All version_id`.

- **Tipo de log:** eles dividem-se em log de atividade – como publicação de novas versões ou alterações – e logs de requisições – que são os mais comuns, dado que uma aplicação no GAE basicamente responde a requisições HTTP.
- **Nível de severidade da mensagem de log:** esse é um filtro importante quando você possui muitas mensagens de log e deseja procurar apenas pelas mensagens com o nível

INFO, por exemplo, porque sabe que é nesse nível que está a informação que procura.

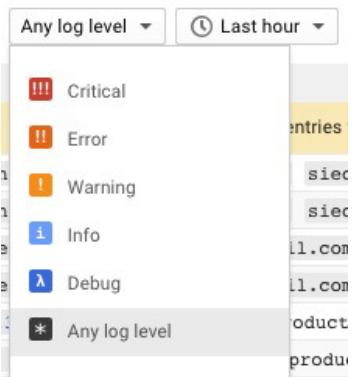


Figura 7.2: Nível de severidade dos logs

- **Período de geração da mensagem de log:** você pode escolher a data/hora da mensagem de log que deseja exibir. Isso é muito útil para analisar eventos que aconteceram com dias e/ou horários bem conhecidos.
- **Campo editável:** você pode digitar um texto para filtrar a mensagem de log, como no exemplo a seguir, no qual as mensagens são filtradas pela URL da requisição que as geraram:

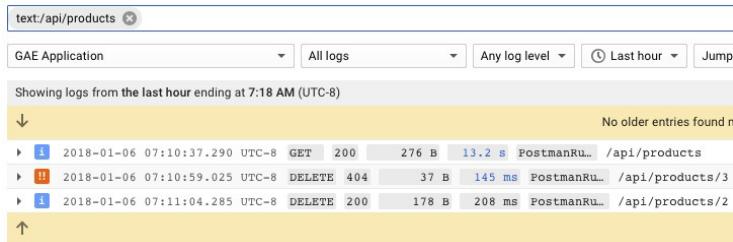


Figura 7.3: Filtrando mensagens de log

Repare que, como resultado do filtro, somente aparecem as mensagens de log que foram geradas pela requisição da URL `api/products/`. Isso é muito útil quando você deseja ver o que aconteceu quando um produto `x` foi acessado, por qualquer método.

Você também pode combinar filtros, como no exemplo a seguir, em que só aparecem mensagens de log de requisições GET na URL `api/products`:

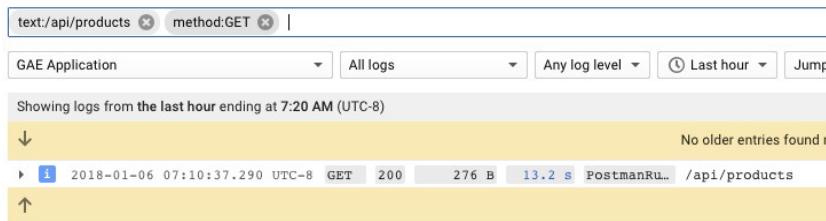


Figura 7.4: Combinando filtros de log

É possível expandir cada mensagem da lista para vermos mais detalhes do que acontece. O ícone localizado à esquerda da mensagem refere-se à sua severidade, como nas opções de configuração do filtro para essa opção. Nos detalhes da mensagem

expandida, é possível ver qual foi o método HTTP usado para a requisição, a URL de acesso, os detalhes do cliente, o código HTTP de retorno, o tempo de resposta, o consumo de CPU e algumas outras coisas mais.

Veja um exemplo em que foi gerada também uma mensagem pelo método `deleteProduct`, informando que o produto de código 2 foi apagado com sucesso:

The screenshot shows a list of log entries from Postman. The first two entries are standard log entries, while the third one is expanded to show detailed trace information. The expanded log entry includes fields like httpRequest, insertId, labels, logName, operation, protoPayload, resource, severity, and timestamp. Below the expanded log, there is a message from the ProductController indicating a successful deletion of a product with code 2.

Time	Method	Status	Latency	Client	URL
2018-01-06 07:10:37.290 UTC-8	GET	200	276 B	13.2 s	PostmanRun... /api/products
2018-01-06 07:10:59.025 UTC-8	DELETE	404	37 B	145 ms	PostmanRun... /api/products/3
2018-01-06 07:11:04.285 UTC-8	DELETE	200	178 B	208 ms	PostmanRun... /api/products/2

```
187.73.169.0 - - [06/Jan/2018:13:11:04 -0200] "DELETE /api/products/2 HTTP/1.1" 200 178 - "PostmanRuntime/7.1.1" "exemplol-186516.appspot.com" ms=208 cpu_ms=931 cpm_usd=1.9892999999999988 -> loading_request=0 instance=0c61b117ca57019d722350d1e44e29706f72c440e6b22a086ee9d0109179c88 fc0f5f5830f0 app_engine_release=1.9.54 trace_id=-
```

```
{
  httpRequest: {...},
  insertId: "5a50e708000b3002687ae4c7",
  labels: {...},
  logName: "projects/exemplol-186516/logs/appengine.googleapis.com%2Frequest_log",
  operation: {...},
  protoPayload: {...},
  receiveTimestamp: "2018-01-06T15:11:04.740394836Z",
  resource: {...},
  severity: "INFO",
  timestamp: "2018-01-06T15:11:04.285529Z"
}

2018-01-06 07:11:04.486 UTC-8 br.com.siecola.gae_exemplol.controller.ProductController delete
Product: Produto com código:[2] apagado com sucesso (ProductController.java:110)
```

Figura 7.5: Detalhe da mensagem de log

Há um link na linha da mensagem de log, logo à esquerda da URL da requisição. Se você clicar nele, será redirecionado para a página de `Trace`, onde poderá analisar com muito mais clareza tudo o que aconteceu:

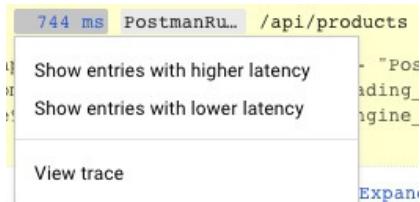


Figura 7.6: Acessando o menu de trace

Veja os detalhes da mensagem selecionada:

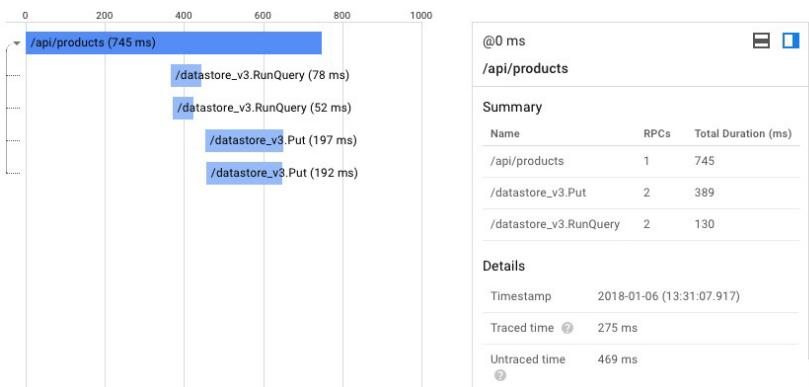


Figura 7.7: Trace da mensagem de log

Repare nessa página que é possível ter um gráfico com o tempo que cada operação levou para executar, desde o tratamento da requisição em si pelo método, como pelo tempo em que houve a consulta ao serviço de Datastore.

Saber utilizar bem as ferramentas de logs e trace pode economizar muito tempo na depuração de um erro, ou no processo de otimização de um tempo de resposta muito longo de uma requisição. **Não deixe de explorar essas funcionalidades que o GAE oferece!**

Conclusão

Neste capítulo, você aprendeu a gerar mensagens de log e visualizá-las no console do GAE, realizando filtros das mais diversas formas. Você viu como o GAE detalha as informações sobre o que aconteceu em cada requisição à aplicação, tornando o trabalho do desenvolvedor (ou do administrador da aplicação) mais fácil para descobrir um erro, ou otimizar o seu tempo de resposta.

Saber usar as ferramentas de análise de logs é um grande diferencial para um profissional que deseja trabalhar com essa plataforma. No próximo capítulo, você aprenderá a como proteger o acesso aos serviços da aplicação `gae_exemplo1`, utilizando um mecanismo de autenticação simples, o *HTTP Basic Auth*.

CAPÍTULO 8

PROTEGENDO SERVIÇOS COM HTTP BASIC AUTHENTICATION

O serviço de gerenciamento de produtos, criado e publicado no GAE, está acessível para qualquer usuário ou sistema, sem nenhum mecanismo de autenticação para evitar acessos não autorizados. Aliás, não há nenhum tipo de solicitação de autorização.

Isso significa que, caso você estivesse construindo um sistema proprietário, exclusivo para uso de um cliente, qualquer pessoa que tivesse conhecimento dele e estivesse conectada à internet poderia utilizá-lo. E se você, ou o seu cliente, estivesse pagando pela infraestrutura do GAE, poderia haver custos por excesso de requisições de usuários e locais desconhecidos.

Esse fato não seria um problema se você disponibilizasse uma API pública, ou seja, algo que fosse servir para qualquer sistema consumir de forma gratuita. Aí você pode pensar: "Mas assim eu teria de pagar para ter essa API publicada no GAE!?" . Não, se você não habilitar a cobrança na sua aplicação. O que pode acontecer é a sua aplicação chegar a um determinado limite diário de consumo do GAE, como número máximo de requisições ou tempo de CPU,

e ficar inacessível até o próximo período de contabilização (diário ou mensal). Ainda bem que, por padrão, as aplicações criadas no GAE não ficam com os mecanismos de cobrança habilitados, então você pode ficar despreocupado.

Mas se você vai desenvolver uma API e, por qualquer que seja o motivo, não deseja liberar o acesso a qualquer sistema, é possível implementar um mecanismo de autenticação e autorização. Há várias questões que devem ser consideradas na escolha de tal mecanismo, como:

- Nível de segurança desejado, pois obviamente existem mecanismos com diferentes técnicas que permite uma maior ou menor segurança;
- Compatibilidade com as aplicações clientes que vão consumir o serviço;
- Carga do servidor no processo de tratamento e autenticação das requisições;
- Quantidade de requisições a serem feitas pelo cliente para poder ser autenticado. O mecanismo HTTP Digest Authentication com MD5, por exemplo, oferece um nível de segurança consideravelmente elevado, mas requer que toda requisição seja negada pelo servidor na primeira vez, devolvendo uma chave ou semente aleatória. Assim, o cliente a refaz com o usuário e a senha gerados a partir dessa semente.

No capítulo *Protegendo serviços com OAuth 2.0*, será mostrado um mecanismo no qual o cliente adquire um token com suas credenciais de acesso. Então, ele usa esse token, que é válido apenas por um tempo determinado, para autenticar suas

requisições no servidor. Isso será feito com ajuda da plataforma do Google App Engine.

Por enquanto, neste capítulo, você conhecerá o mecanismo **HTTP Basic Authentication**, para iniciar a questão de autenticação e autorização de acesso de usuários a serviços REST. Dessa forma, será mostrado como construir tais mecanismos utilizando os recursos do Spring Boot.

No capítulo seguinte, você utilizará o serviço Google Cloud Datastore para fornecer a base de usuários e papéis que serão utilizados para definir quem poderá acessar as operações dos serviços da sua aplicação.

8.1 O QUE É HTTP BASIC AUTHENTICATION

O mecanismo HTTP Basic Authentication é muito simples de ser implementado, tanto do lado do servidor como do cliente. Ele apenas requer que, em toda requisição HTTP, o cliente envie o cabeçalho `Authorization` , como mostrado a seguir:

```
Authorization: Basic QWRtaW46QWRtaW4=
```

O campo `Authorization` deve conter: o esquema de autenticação, `Basic` , como diz o próprio nome do esquema; e as credenciais de acesso (usuário e senha), codificadas em Base64 no formato "usuário:senha" .

O servidor que trata a requisição e autoriza-a deve executar os passos descritos a seguir. A forma como ele executa tais passos depende do framework usado. Neste capítulo, usaremos o Spring Security, portanto, no momento é importante entender o **que** ele faz, e não **como** faz.

1. Verifica a existência do cabeçalho com o nome `Authorization`;
2. Lê e valida o esquema de autenticação, que deve ser do tipo `Basic`;
3. Lê a string com as credenciais em formato Base64;
4. Decodifica as credenciais em formato Base64 e obtém o usuário e a senha enviados pelo cliente;
5. De posse do usuário e da senha, o servidor busca em uma base de dados, para verificar se as credenciais estão corretas e o usuário tem permissão de acesso, baseado em regras ou papéis.

Como você pode ver, esse mecanismo é bem simples, com apenas alguns passos a serem executados por parte do servidor. Isso torna-o muito leve, mas com algumas desvantagens:

- Toda requisição possui o mesmo valor no cabeçalho `Authorization`, com os valores do usuário e da senha codificados sempre da mesma forma. Se alguma requisição for interceptada, esse cabeçalho pode ser utilizado, com o mesmo valor, para se passar pelo mesmo usuário original que a fez.
- O usuário e a senha não estão criptografados, o que permite que eles sejam facilmente decodificados aos valores originais.
- Se alguma requisição do cliente for interceptada (por exemplo, com um *sniffer* de rede), as credenciais de acesso estarão expostas e poderão ser decodificadas com quase nenhum esforço.

Se você possui uma preocupação com o fato de as requisições

dos clientes da sua API poderem ser interceptadas, é melhor utilizar um mecanismo de autenticação mais forte, como OAuth ou HTTP Digest. Há ainda quem recomende usá-lo em conjunto com algum mecanismo de criptografia da requisição como um todo, como o SSL. Entretanto, a análise sobre o que é mais ou menos custoso para o servidor é um assunto mais longo, e não está no escopo do livro.

Se o rigor da segurança não for uma preocupação grande, o HTTP Basic Authentication é um bom mecanismo de proteção de serviços REST. Aliás, ele é muito utilizado em aplicações com requisitos de baixa segurança e menor consumo de recursos do servidor.

8.2 CONFIGURANDO O PROJETO COM HTTP BASIC AUTHENTICATION

Agora que você já sabe o que é e como funciona esse mecanismo, é hora de usá-lo no projeto para proteger o serviço de gerenciamento de produtos. Primeiramente, começaremos com o básico: configurar o projeto para proteger todas as requisições a esse serviço, verificando um usuário e sua senha (fixos no código), para que os conceitos primordiais sejam entendidos.

Posteriormente, você vai adicionar regras de autorização de acesso às operações, dependendo do papel do usuário. No próximo capítulo, você criará uma base de dados de usuário, com o Google Cloud Datastore. Dessa forma, você aprenderá a unir um

importante recurso do Spring Boot com um poderoso serviço de infraestrutura de *cloud computing*, em uma aplicação no GAE.

Como o projeto foi desenvolvido utilizando o Spring Boot, configurá-lo para adicionar autenticação do tipo HTTP Basic é muito simples, pois **o Spring possui um módulo de segurança, chamado Spring Security** que cuida de tudo o que é necessário. Você apenas precisa adicioná-lo ao projeto, escrever algumas configurações e tudo estará funcionando perfeitamente.

Os detalhes de como ele funciona e como utilizá-lo serão descritos ao longo deste capítulo e também no capítulo *Protegendo serviços com OAuth 2*.

Adicionando as dependências ao projeto

Para adicionar o módulo de segurança do Spring, abra o arquivo `pom.xml` e, ao final da lista de dependências, adicione mais um item:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

O **Spring Boot Starter Security** traz tudo o que você precisa para a implementação da parte de autenticação e autorização dos serviços REST. À medida que você for construindo e configurando o mecanismo de autenticação da aplicação, você utilizará as partes dessa biblioteca.

Criando a classe de configuração de autenticação

Depois de adicionar o módulo de segurança do Spring, agora é

necessário configurá-lo adequadamente, pois ele possui algumas variações de uso e execução, como:

- O tipo de autenticação a ser utilizado, como HTTP Basic, OAuth 2.0;
- A fonte de usuários para serem autenticados.

Como de costume no Spring Boot, as configurações são feitas por meio de classes Java. A única que será necessária servirá para balizar o comportamento do mecanismo de autenticação, configurando os seguintes itens:

- Fonte das informações de usuários;
- Tipo de autenticação a ser utilizada, que nesse caso será HTTP Basic;
- Filtros para a definição de URL com acessos diferenciados.

No momento, a fonte das informações de usuários será apenas duas, criadas em memória, para facilitar o entendimento do mecanismo: uma será o usuário com papel `ADMIN` e a outra com papel `USER`, para que você exerça a validação de papéis durante as fases de autorização das requisições.

Para fazer tal classe de configuração, comece criando um novo pacote chamado `config`, dentro de `br.com.siecola.gae_exemplo1`. Nesse novo pacote, ficarão todas as classes que vão desempenhar algum papel de configuração da aplicação. Nele, crie a primeira classe de configuração, com o nome de `SpringSecurityConfigHttpBasic`, como no trecho a seguir:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.security.config.annotation.method.con  
figuration.EnableGlobalMethodSecurity;  
import org.springframework.security.config.annotation.web.builders.HttpSecurity;  
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;  
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;  
import org.springframework.security.config.http.SessionCreationPolicy;  
import org.springframework.security.core.userdetails.User;  
import org.springframework.security.core.userdetails.UserDetailsService;  
import org.springframework.security.provisioning.InMemoryUserDetailsManager;  
  
@Configuration  
@EnableWebSecurity  
@EnableGlobalMethodSecurity(prePostEnabled = true)  
public class SpringSecurityConfigHttpBasic extends WebSecurityCon  
figurerAdapter {
```

Perceba que ela estende de `WebSecurityConfigurerAdapter`, que possui alguns métodos que serão invocados pelo Spring para a configuração do mecanismo de autenticação. A anotação `@Configuration` diz que essa classe desempenha um papel de configuração da aplicação e, por isso, será invocada durante a inicialização.

A anotação `@EnableGlobalMethodSecurity` possibilita o uso de outras anotações, que serão vistas mais adiante, para configuração de autorizações específicas em classes e métodos de serviços.

Configurando o mecanismo de autenticação

Agora nessa nova classe, crie o método `configure(HttpSecurity http)`, que será invocado pelo

Spring para a configuração do mecanismo de autenticação. Ele vai informar, dentre outras coisas, qual será o método de autenticação exigido pelos acessos HTTP à aplicação:

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.csrf().disable()  
        .authorizeRequests()  
        .antMatchers("_ah/**").permitAll()  
        .anyRequest().authenticated()  
        .and().httpBasic()  
        .and().sessionManagement()  
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS);  
}
```

A invocação do método `httpBasic()` é a parte principal desse trecho de código, pois ela diz que o mecanismo de autenticação escolhido para esse momento é o **HTTP Basic**.

A chamada `.antMatchers("_ah/**").permitAll()` é uma forma de configurar os filtros do mecanismo de autenticação. Nesse caso, ele está dizendo que o acesso à página de configuração local do GAE da aplicação está liberado para qualquer usuário.

Existe outra forma de se fazer a configuração das autorizações de acesso, específicas a cada serviço, que é através de anotações nas classes que os implementam. A forma como isso pode ser feito será detalhado mais adiante, ainda nesse capítulo.

As demais linhas desse método configuram, dentre outras coisas, que todas as requisições deverão ser autenticadas para serem tratadas pela aplicação.

Configurando a fonte de dados de usuário

O último método a ser adicionado nessa classe é o que informa

a fonte de dados dos usuários. O mecanismo de autenticação necessita desses dados para conferir as credenciais de acesso informadas em cada requisição HTTP . O método a ser apresentado aqui configura dois usuários fixos no código, que ficarão armazenados em memória pela aplicação, como explicado anteriormente.

No capítulo seguinte, será adicionado um novo serviço de usuários, bem como uma nova tabela no Google Cloud Datastore, para armazenamento dos usuários e papéis que terão permissão de acesso à aplicação.

Veja como deve ficar esse método:

```
@Bean
public UserDetailsService userDetailsService() {
    User.UserBuilder users = User.withDefaultPasswordEncoder();
    InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();
    manager.createUser(users.username("user").password("user").roles("USER").build());
    manager.createUser(users.username("admin").password("admin").roles("USER", "ADMIN").build());
    return manager;
}
```

Nele, dois usuários são criados e armazenados em memória, cada um com um papel de diferente: ADMIN e USER . Esses dois papéis serão usados para definir a autorização de acesso a cada serviço ou operação.

Mais adiante, você vai configurar e usar outra funcionalidade importante desse módulo de segurança do Spring, que são as anotações para definição de autorização de acesso, de acordo com o papel do usuário.

IMPORTS DE BIBLIOTECAS

Se tiver dúvidas sobre quais imports utilizar, consulte o código-fonte do projeto, em <https://github.com/siecola/GAEBookV3Exemplo1>.

8.3 TESTANDO O SERVIÇO DE PRODUTOS COM HTTP BASIC AUTHENTICATION

Agora que a aplicação gae_exemplo1 está configurada com o mecanismo de autenticação HTTP Basic , se você tentar acessar a operação para listar todos os produtos pelo Postman, por exemplo, você deverá receber a seguinte resposta:

The screenshot shows the Postman interface with the following details:

- Body tab is selected.
- Status: 401 Unauthorized
- JSON dropdown is set to JSON.
- Response body (Pretty):

```
1 {  
2   "timestamp": 1517664054075,  
3   "status": 401,  
4   "error": "Unauthorized",  
5   "message": "Full authentication is required to access this resource",  
6   "path": "/api/products"  
7 }
```

Figura 8.1: Autorização de acesso negada

Veja que o código da mensagem de resposta é o HTTP 401 Unauthorized , indicando que o acesso a essa operação não foi autorizado pela aplicação. Veja também que há uma mensagem no corpo da resposta, detalhando o fato. Esse era o comportamento esperado, pois agora a aplicação gae_exemplo1 está configurada

para permitir que somente os usuários user e admin acessem os seus serviços.

Para fazer com que a requisição de acesso à listagem de produtos seja autorizada, é necessário adicionar as credenciais de um dos dois usuários permitidos. No Postman, é possível fazer isso acessando a aba `Authorization`, logo abaixo do campo onde o endereço do serviço foi configurado.

Nessa aba, escolha a opção `Basic Auth` e configure os campos `Username` e `Password` com as credenciais de acesso de um dos dois usuários configurados na aplicação. Veja como deve ficar:

The screenshot shows the Postman interface with the following details:

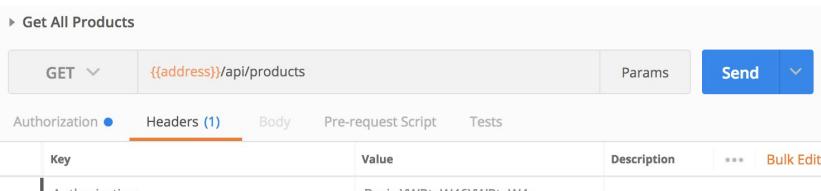
- Request Method:** GET
- URL:** {{address}}/api/products
- Authorization Tab (selected):** Shows `Basic Auth` selected under `TYPE`. A note says: "The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)".
- Username:** admin
- Password:** admin
- Show Password:** checked

Figura 8.2: Configurando autenticação HTTP Basic no Postman

Com essas informações, é possível acessar qualquer operação da aplicação `gae_exemplo1`, pois agora a requisição HTTP conterá um novo cabeçalho, chamado `Authorization`, com as credenciais de acesso do usuário `admin`, no formato `HTTP Basic`.

Para visualizar o novo cabeçalho, clique no botão `Send` para enviar a requisição, e acesse a aba `Headers`. O Postman se

encarregará de acrescentá-lo com as informações que você configurou na aba Authorization :



The screenshot shows the Postman interface with a 'Get All Products' request. The method is set to 'GET' and the URL is '{{address}}/api/products'. The 'Headers' tab is selected, showing one header named 'Authorization' with the value 'Basic YWRtaW46YWRtaW4='. Other tabs like 'Body', 'Pre-request Script', and 'Tests' are visible. A 'Send' button is at the top right.

Figura 8.3: Cabeçalho Authorization

Repare que o nome do novo cabeçalho é Authorization , com o valor Basic YWRtaW46YWRtaW4= . O valor após a palavra Basic é o usuário e a senha, configurados na aba Authorization e codificados no padrão Base64 (no formato username:password).

Essa é uma configuração que vale somente para essa requisição salva no Postman. Certifique-se de incluir as credenciais de acesso em todas as requisições que fizer aos serviços da aplicação gae_exemplo1 .

BASE DE DADOS DE USUÁRIOS

É importante ressaltar que, até o momento, a aplicação `gae_exemplo1` não possui base de dados de usuários, com o intuito de focar nas explicações e no entendimento dos conceitos da construção dos serviços REST. Por isso, as informações de usuário, senha e papel (que será acrescentado na próxima seção) estão fixas no código. No capítulo *Adicionando o serviço de usuários*, será construído um serviço para gerenciamento de usuários, e o mecanismo de autenticação será alterado para usar essa base de dados em vez de usuários fixos em memória.

8.4 ADICIONANDO ANOTAÇÕES PARA CONTROLE DE PERMISSÕES E PAPÉIS

O serviço de gerenciamento de produtos já está protegido com *HTTP Basic Authentication*, mas todas as operações possuem as mesmas regras de segurança, sem nenhuma diferenciação do usuário que acessa a operação em si.

Agora, imagine que você precisasse implementar as seguintes diretrizes:

- A listagem de todos os produtos (ou de somente um), pelo seu código, é permitida a qualquer usuário autenticado;
- O cadastramento, a alteração e a exclusão de um produto só podem ser feitos por usuários autenticados com o papel **ADMIN** .

Utilizar papéis para os usuários é uma forma de diferenciá-los, dando a eles permissões ou restrições, seja para todas ou algumas operações de um serviço REST.

Para implementar tal funcionalidade no serviço de produtos, basta adicionar anotações nos métodos que implementam essas operações. Elas funcionaram em conjunto com o mecanismo de autenticação configurado, trabalhando em conjunto com os papéis de usuários existentes.

As configurações de autorização de acesso devem ser realizadas na própria classe que implementa o serviço. Para isso, abra a classe `ProductController` e adicione a seguinte anotação nos métodos `getProduct` , `getProducts` :

```
@PreAuthorize("hasRole('USER') or hasRole('ADMIN')")
```

Isso faz com que qualquer usuário autenticado, que possua o papel `USER` ou `ADMIN` , possa acessar a operação.

Então, para instruir o mecanismo de autenticação a permitir que somente usuários com papel `ADMIN` possam criar, alterar ou excluir um produto, adicione a seguinte anotação aos métodos `saveProduct` , `deleteProduct` e `updateProduct` :

```
@PreAuthorize("hasRole('ADMIN')")
```

Dessa forma, somente o usuário `ADMIN` terá autorização para acessar tais operações. Se você tentar acessar qualquer uma dessas três operações com as credenciais de acesso de um usuário com papel `USER` , o serviço vai responder com o código `HTTP 403 Forbidden` .

Isso possui um significado diferente do `HTTP 401`

`Unauthorized`, que apareceu quando você tentou acessar sem as credenciais de acesso de nenhum usuário. O `HTTP 403 Forbidden` significa que o usuário é reconhecido, porém, ele não está autorizado a acessar o recurso que deseja.

Perceba que, com apenas algumas configurações, é possível fazer com que a aplicação fique protegida de acessos não autorizados aos seus serviços.

Você pode encontrar o código completo do projeto `gae_exemplo1` no repositório:

<https://github.com/siecola/GAEBookV3Exemplo1>.

Conclusão

Neste capítulo, você aprendeu a como proteger serviços REST com o mecanismo HTTP Basic Authentication, utilizando uma configuração do Spring e suas anotações. A partir de agora, qualquer serviço que for criado na aplicação `gae_exemplo1` poderá utilizar o mesmo mecanismo já desenvolvido, com as anotações colocadas no serviço de produtos.

No próximo capítulo, você tornará o mecanismo de autenticação ainda mais eficiente e sofisticado, criando um serviço de gerenciamento de usuários do sistema e uma tabela no Google Cloud Datastore, para poder armazenar seus dados. Com essa tabela, será possível guardar as credenciais de acesso, papéis e outras informações sobre o usuário.

CAPÍTULO 9

ADICIONANDO O SERVIÇO DE USUÁRIOS

Neste capítulo, você incrementará a aplicação `gae_exemplo1`, adicionando mais um serviço que será muito importante para tornar o mecanismo de autenticação mais completo. Ele será o serviço de gerenciamento de usuários. Com o serviço de usuários, será possível cadastrar usuários na aplicação, para usarmos um mecanismo de autenticação em vez de apenas dois usuários fixos, como feito no capítulo anterior.

Junto a esse novo serviço, criaremos um novo tipo (*Kind*) no Google Cloud Datastore, que será o usuário. Essas informações também serão úteis para a funcionalidade de enviar mensagens a dispositivos móveis, com o Firebase Cloud Messaging, o que será visto no próximo capítulo. Cada entidade do usuário terá informações para que esse serviço consiga entregar mensagens individuais a cada um deles.

Para que você se aprofunde mais nas habilidades de desenvolver aplicações com o Spring Boot e o GAE, desenvolveremos uma camada de abstração entre o *controller*, que expõe as operações do novo serviço de usuário, e a manipulação dos dados no Google Cloud Datastore.

Elá será responsável por implementar a lógica de acesso a esse serviço de armazenamento do GAE, e será chamada de *repository*. Nesse primeiro momento, essa técnica terá as seguintes vantagens:

- A classe do *controller* ficará mais simples e responsável apenas por expor as operações do serviço de usuários;
- A lógica de acesso ao Google Cloud Datastore ficará isolada em uma classe, apenas com essa responsabilidade;
- Essa classe de acesso ao Google Cloud Datastore poderá ser usada em mais de um ponto da aplicação, sem a necessidade de duplicação de código;
- No futuro, caso você deseje alterar o serviço de armazenamento de dados, poderá trocar apenas essa implementação.

9.1 CRIANDO O MODELO DE USUÁRIOS

Antes de criar o novo serviço, é necessário criar o modelo de dados que o representa. Ele também será usado para a construção do novo tipo (ou tabela) no Google Cloud Datastore, para que os usuários sejam armazenados nele.

O modelo de usuários deve ter, pelo menos, os seguintes atributos para poder ser usado no mecanismo de autenticação:

- Identificação única no Google Cloud Datastore;
- Registro no Firebase Cloud Messaging;
- E-mail do usuário, para ser utilizado como login;
- Senha de acesso do usuário;
- Data e hora do último login;
- Data e hora do último registro no Firebase Cloud

- Messaging;
- Papel do usuário.

O atributo com o valor de registro no Firebase Cloud Messaging será usado para enviar mensagens aos dispositivos móveis, registrados para receber notificações da aplicação. Os detalhes da implementação desse mecanismo serão mostrados no capítulo *Enviando mensagens com o Firebase Cloud Messaging*.

De posse dos atributos do usuário, você pode criar a classe modelo User para representá-lo no pacote br.com.siecola.gae_exemplo1.model. Veja no código a seguir como ela ficará:

```
import java.io.Serializable;
import java.util.Date;

public class User implements Serializable {
    private Long id;
    private String email;
    private String password;
    private String fcmRegId;
    private Date lastLogin;
    private Date lastFCMRegister;
    private String role;
    private boolean enabled;

    //Getters and setters
}
```

Essa classe modelo também será usada para persistir os dados dos usuários em um novo tipo no Google Cloud Datastore, como dito anteriormente.

Detalhando a autenticação com Spring Security e Google Cloud Datastore

Como explicado e desenvolvido no capítulo anterior, a aplicação gae_exemplo1 utiliza o **Spring Security** para cuidar do mecanismo de autenticação das requisições HTTP . Tais requisições são autenticadas, de acordo com as configurações implementadas na classe `SpringSecurityConfigHttpBasic` da aplicação, antes de chegarem ao *controller* (que trata as requisições através dos métodos da classe `ProductController`).

Caso uma requisição não contenha credenciais válidas, ou o usuário não possua autorização, a requisição é negada. Até o momento, o Spring Security usava dois usuários criados e armazenados em memória, como pode ser visto no método `userDetailsService` da classe `SpringSecurityConfigHttpBasic` :

```
@Bean  
public UserDetailsService userDetailsService() {  
    User.UserBuilder users = User.withDefaultPasswordEncoder();  
    InMemoryUserDetailsManager manager = new InMemoryUserDetailsM  
anager();  
    manager.createUser(users.username("user").password("user").ro  
les("USER").build());  
    manager.createUser(users.username("admin").password("admin").  
roles("USER", "ADMIN").build());  
    return manager;  
}
```

A ideia agora é fazer com que os usuários venham do Google Cloud Datastore. Dessa forma, eles poderiam ser cadastrados ou excluídos, de forma a prover um mecanismo de autenticação mais concreto, do ponto de vista da autorização de usuários.

Antes de aplicar as regras de autenticação e autorização, o

Spring Security verificaría no Google Cloud Datastore se elas estão corretas, a partir do usuário que tenta se autenticar. Veja o diagrama que mostra o caminho de uma requisição HTTP:

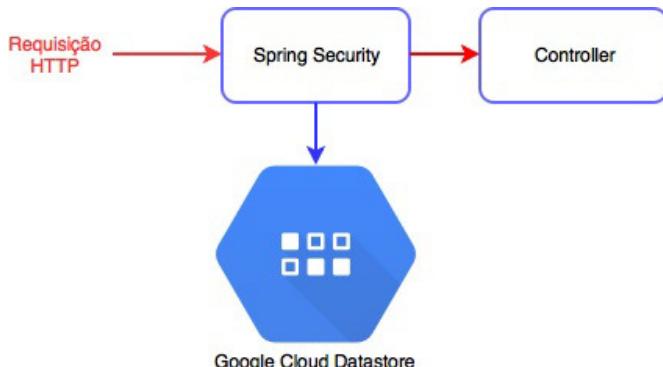


Figura 9.1: Autenticação utilizando o Google Cloud Datastore

Se o usuário existir na base e a senha for verificada, a requisição prossegue para o *controller* de destino.

9.2 PREPARANDO O MODELO DE USUÁRIOS PARA O MECANISMO DE AUTENTICAÇÃO

Para que o mecanismo de autenticação consiga acessar a base de dados de usuários e manipulá-los corretamente, é necessário preparar o modelo de usuário, com métodos específicos (detalhados mais adiante). Assim, o Spring fará suas validações e indicará a ele onde e como buscar os dados.

Para começar, o modelo de usuário deve implementar `UserDetails`, que é uma interface do Spring Security para descrever detalhes do usuário. Ela fará com que o modelo `User` contenha alguns métodos obrigatórios, que serão invocados pelo

mecanismo de autenticação:

```
public class User implements Serializable, UserDetails {  
    private Long id;  
    private String email;  
    private String password;  
    private String fcmRegId;  
    private Date lastLogin;  
    private Date lastFCMRegister;  
    private String role;  
    private boolean enabled;  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities(  
    ) {  
        return null;  
    }  
    @Override  
    public String getUsername() {  
        return null;  
    }  
    @Override  
    public boolean isAccountNonExpired() {  
        return false;  
    }  
    @Override  
    public boolean isAccountNonLocked() {  
        return false;  
    }  
    @Override  
    public boolean isCredentialsNonExpired() {  
        return false;  
    }  
  
    //Getters and setters
```

Dentro desses novos métodos adicionados para respeitar a interface `UserDetails`, você deve retornar as informações que são solicitadas em cada um deles. Um exemplo seria qual a informação que define o usuário, que seria retornada pelo método `getUsername` – nesse exemplo, vamos usar o e-mail do usuário, como você verá logo a seguir.

O mais importante é a questão sobre as autorizações que o usuário possui, em resumo, quais são os papéis que ele pode assumir. Como o modelo de usuário possui apenas um, ele será a informação de resposta:

```
@JsonIgnore  
@Override  
public Collection<? extends GrantedAuthority> getAuthorities() {  
    List<GrantedAuthority> setAuths = new ArrayList<>();  
    setAuths.add(new SimpleGrantedAuthority(this.getRole()));  
    return setAuths;  
}
```

Perceba que é possível retornar uma lista de autorizações do usuário. Em uma aplicação futura, caso você precise que um usuário possua mais de um papel ou autorização, isso é possível.

Os demais métodos vão retornar valores fixos, pois não serão gerenciados pela aplicação `gae_exemplo1`. Entretanto, eles serão chamados pelo Spring Security durante a execução do mecanismo de autenticação.

```
@JsonIgnore  
@Override  
public boolean isAccountNonExpired() {  
    return true;  
}  
  
@JsonIgnore  
@Override  
public boolean isAccountNonLocked() {  
    return true;  
}  
  
@JsonIgnore  
@Override  
public boolean isCredentialsNonExpired() {  
    return true;  
}
```

```
@JsonIgnore  
@Override  
public String getUsername() {  
    return this.email;  
}
```

Por exemplo, você poderia ter um mecanismo que faz com que as credenciais do usuário expirem, por algum motivo específico. Nesse caso, o método `isCredentialsNonExpired` deveria retornar falso. Esse é só um exemplo da utilização dos demais métodos que serão invocados pelo Spring.

A única exceção é o método `getUsername`, no qual você deve retornar o valor que será usado pelo mecanismo de autenticação para verificar as credenciais do usuário, juntamente com o método `getPassword`. Como o que define o usuário da aplicação `gae_exemplo1` é o seu endereço de e-mail, então é esse valor que deve ser retornado nesse método.

A anotação `@JsonIgnore` é utilizada aqui apenas para fazer com os valores retornados pelos métodos não apareçam na resposta do serviço de usuário, uma vez que eles não fazem parte realmente da informação armazenada no Google Cloud Datastore.

9.3 CRIANDO A CAMADA DE REPOSITÓRIO DE USUÁRIO

Como mencionado no início do capítulo, a ideia de criar uma camada de repositório para a entidade do usuário é para encapsular a lógica de acesso ao Google Cloud Datastore e permitir que ela possa ser acessada de outras partes da aplicação, e não somente do *controller* de serviços de usuário.

Essa ideia permite que o repositório de usuários possa ser injetado em qualquer parte da aplicação, utilizando as facilidades do Spring Framework. Um exemplo disso é quando você implementa a busca do usuário dentro do mecanismo de autenticação.

O código de manipulação das informações e acesso ao Google Cloud Datastore não será muito diferente do que já foi explicado no serviço de produtos, por isso a explicação será sucinta no que toca essa questão. O foco será nas diferenças da implementação, que permitirão transformar esse código reaproveitável para outras partes da explicação.

Criando a classe do repositório de usuário

Para manter a organização de pacotes da aplicação, crie um novo com o nome `br.com.siecola.gae_exemplo1.repository` e coloque a nova classe `UserRepository` dentro dele.

Ela deverá ter a anotação `@Repository` do Spring, indicando que é um repositório de dados. Essa anotação permitirá que a instância dessa classe possa ser injetada em outras partes da aplicação, com seu ciclo de vida gerenciado pelo Spring.

```
import br.com.siecola.gae_exemplo1.model.User;
import com.google.appengine.api.datastore.*;
import org.springframework.stereotype.Repository;

import javax.annotation.PostConstruct;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Optional;
import java.util.logging.Logger;

@Repository
```

```
public class UserRepository {  
    private static final Logger log = Logger.getLogger("UserRepos  
itory");  
  
    @Autowired  
    private PasswordEncoder passwordEncoder;  
}
```

Da forma como esse repositório foi criado, com a anotação `@Repository`, será possível injetá-lo em qualquer lugar em que seja necessário acessar a base de dados dos usuários.

O atributo privado criado nessa classe, do tipo `PasswordEncoder`, será utilizado para não deixar que a senha do usuário seja salva aberta no banco de dados. O mesmo tipo também será utilizado pelo mecanismo de autenticação a ser implementado a seguir.

Constantes de propriedades do usuário

Para facilitar a construção dos métodos que acessam às propriedades do tipo usuário no Google Cloud Datastore, é interessante criar constantes do tipo texto para cada uma delas – em vez de digitá-los a todo o momento:

```
private static final String USER_KIND = "Users";  
private static final String USER_KEY = "userKey";  
  
private static final String PROPERTY_ID = "UserId";  
private static final String PROPERTY_EMAIL = "email";  
private static final String PROPERTY_PASSWORD = "password";  
private static final String PROPERTY_FCM_REG_ID = "fcmRegId";  
private static final String PROPERTY_LAST_LOGIN = "lastLogin";  
private static final String PROPERTY_LAST_FCM_REGISTER = "lastFCM  
Register";  
private static final String PROPERTY_ROLE = "role";  
private static final String PROPERTY_ENABLED = "enabled";
```

Dessa forma, basta utilizar essas constantes quando formos nos referir a alguma propriedade do usuário.

Métodos auxiliares

Assim como foi feito no serviço de produtos, é interessante ter alguns métodos auxiliares para manipulação das informações do usuário em vez de repetir código a todo instante – por exemplo, quando for necessário converter um usuário em uma entidade do Google Cloud Datastore.

O primeiro método auxiliar a ser construído é o que transforma um objeto do tipo `User` (da aplicação `gae_exemplo1`) em uma entidade representativa no Google Cloud Datastore:

```
private void userToEntity (User user, Entity userEntity) {  
    userEntity.setProperty(PROPERTY_ID, user.getId());  
    userEntity.setProperty(PROPERTY_EMAIL, user.getEmail());  
    userEntity.setProperty(PROPERTY_PASSWORD, user.getPassword());  
    userEntity.setProperty(PROPERTY_FCM_REG_ID, user.getFcmRegId());  
    userEntity.setProperty(PROPERTY_LAST_LOGIN, user.getLastLogin());  
    userEntity.setProperty(PROPERTY_LAST_FCM_REGISTER,  
        user.getLastFCMRegister());  
    userEntity.setProperty(PROPERTY_ROLE, user.getRole());  
    userEntity.setProperty(PROPERTY_ENABLED, user.isEnabled());  
}
```

Veja que as constantes estão sendo utilizadas nesse método para evitar escrever os textos das propriedades a todo momento, o que é uma boa prática.

O segundo método auxiliar faz a conversão contrária, ou seja, de um objeto entidade do Google Cloud Datastore em uma

instância do tipo `User` da aplicação:

```
private User entityToUser (Entity userEntity) {  
    User user = new User();  
    user.setId(userEntity.getKey().getId());  
    user.setEmail((String) userEntity.getProperty(PROPERTY_EMAIL)  
)  
    user.setPassword((String) userEntity.getProperty(PROPERTY_PAS  
WORD));  
    user.setFcmRegId((String) userEntity.getProperty(PROPERTY_FCM  
_REG_ID));  
    user.setLastLogin((Date) userEntity.getProperty(PROPERTY_LAST  
_LOGIN));  
    user.setLastFCMRegister((Date) userEntity.  
        getProperty(PROPERTY_LAST_FCM_REGISTER));  
    user.setRole((String) userEntity.getProperty(PROPERTY_ROLE));  
    user.setEnabled((Boolean) userEntity.getProperty(PROPERTY_ENA  
BLED));  
    return user;  
}
```

Novamente, as constantes auxiliam no código, deixando-o mais limpo e fácil de ser mantido, caso você deseje alterar alguma propriedade no futuro.

O último método servirá para verificar se já existe um determinado e-mail cadastrado na base de usuário, para evitar duplicações:

```
private boolean checkIfEmailExist (User user) {  
    DatastoreService datastore = DatastoreServiceFactory  
        .getDatastoreService();  
  
    Query.Filter filter = new Query.FilterPredicate(PROPERTY_EMAIL,  
        Query.FilterOpe  
rator.EQUAL, user.getEmail());  
  
    Query query = new Query(USER_KIND).setFilter(filter);  
    Entity userEntity = datastore.prepare(query).asSingleEntity();  
  
    if (userEntity == null) {  
        return false;
```

```
    } else {
        if (user.getId() == null) {
            return true;
        } else {
            return userEntity.getKey().getId() != user.getId();
        }
    }
}
```

Essa lógica considera o próprio `User` que está sendo pesquisado. Isso evita que o método retorne com a informação de que o usuário já existe, se for ele mesmo. Repare como o código fica mais limpo quando utilizamos as constantes das propriedades do usuário.

Exceções do repositório de usuários

Trabalhar com lançamento de exceções para indicar estados inconsistentes é uma boa prática. Quando se trata de um repositório de dados, isso facilita a lógica do fluxo de decisões de quem os utiliza.

Como esse é um repositório simples, apenas duas exceções serão necessárias:

- Usuário não encontrado;
- Usuário já existe na base, com o mesmo endereço de e-mail.

Essas exceções poderão ser usadas em qualquer operação em que precisemos informar esse estado para quem estiver utilizando o repositório de dados do usuário. Você verá que o código ficará muito mais limpo com essa estratégia.

Como essas exceções não existem nos pacotes comuns do

mundo Java, você terá de criá-las. Cada exceção será representada por uma classe que herdará de `Exception`, para ser capturada quando acontecer. Para isso, crie um novo pacote em `br.com.siecola.gae_exemplo1.exception`.

Dentro desse novo pacote, crie a classe `UserAlreadyExistsException`. Assim, poderemos definir o evento quando um usuário já existir na base de dados com o mesmo e-mail cadastrado:

```
public class UserAlreadyExistsException extends Exception {  
    private String message;  
  
    public UserAlreadyExistsException(String message) {  
        super(message);  
        this.message = message;  
    }  
  
    @Override  
    public String getMessage() {  
        return message;  
    }  
}
```

O código criado no repositório do usuário utilizará essa exceção para informar quando um e-mail já existir na base de dados. Basta que a classe herde de `Exception` e receba uma string com a mensagem do erro para ser exibida no console da aplicação, ou na resposta de algum serviço.

A outra exceção ficará na classe `UserNotFoundException`, para definir o evento quando um usuário não for encontrado na base de dados por meio do seu endereço de e-mail, como mostra o trecho a seguir:

```
public class UserNotFoundException extends Exception {  
    private String message;
```

```
public UserNotFoundException(String message) {  
    super(message);  
    this.message = message;  
}  
  
public String getMessage() {  
    return message;  
}  
}
```

Repare que o código é muito parecido, porém, como ele tem outro nome, representa um outro evento – quando um usuário pesquisado na base não existir. Agora temos tudo pronto para começar a escrever o repositório de dados do usuário!

Salvar usuário

Como dito no início deste capítulo, o código para manipulação das informações no Google Cloud Datastore será muito parecido com o que foi feito no serviço de produtos. Por isso, a explicação aqui será focada nas diferenças da implementação – principalmente pelos lançamentos de exceção e tipos de retorno.

O método para salvar o usuário deverá, obviamente, recebê-lo como parâmetro. Além disso, ele será devolvido como resposta, já contendo a identificação única da entidade na base de dados. Veja como deve ficar a assinatura desse método:

```
public User saveUser (User user) throws UserAlreadyExistsException
```

Repare que ele já está preparado para lançar a exceção, criada há pouco, caso já exista um usuário com o mesmo e-mail na base. Essa verificação será feita por um dos métodos auxiliares já criados, o `checkIfEmailExist`:

```

public User saveUser (User user) throws UserAlreadyExistsException {
    DatastoreService datastore = DatastoreServiceFactory
        .getDatastoreService();

    if (!checkIfEmailExist (user)) {
        Key userKey = KeyFactory.createKey(USER_KIND, USER_KEY);
        Entity userEntity = new Entity(USER_KIND, userKey);

        user.setPassword(passwordEncoder.encode(user.getPassword()));
        userToEntity (user, userEntity);

        datastore.put(userEntity);

        user.setId(userEntity.getKey().getId());

        return user;
    } else {
        throw new UserAlreadyExistsException("Usuário " + user.getEmail()
            + " já existe");
    }
}

```

Veja que, antes de salvar o novo usuário, o método `checkIfEmailExist` é chamado para consultar na base de dados se já não existe um outro usuário com o mesmo endereço de e-mail. Se existir, a exceção `UserAlreadyExistsException` é lançada, indicando esse fato. Caso não exista nenhum outro usuário com o mesmo e-mail, ele é salvo no Google Cloud Datastore. Perceba também que a senha do usuário está sendo encriptada antes de ser salva no banco, para evitar que seja armazenada aberta.

Alterar o usuário

O método para alterar o usuário deve considerar dois tratamentos de erros:

- Validar se o usuário já existe com o mesmo endereço de e-mail na base de dados;
- Validar se o usuário a ser alterado, em questão, realmente já existe na base de dados.

Se esse método pode lançar dois erros distintos, logo, sua assinatura deverá possuir duas exceções distintas para representar tais cenários. Ele ficará assim:

```
public User updateUser (User user, String email)
    throws UserNotFoundException, UserAlreadyExistsException {

    if (!checkIfEmailExist (user)) {
        DatastoreService datastore = DatastoreServiceFactory
            .getDatastoreService();

        Query.Filter emailFilter = new Query.FilterPredicate(PROPERTY_
EMAIL,
            Query.FilterOperator.EQUAL, email);

        Query query = new Query(USER_KIND).setFilter(emailFilter);

        Entity userEntity = datastore.prepare(query).asSingleEntity()
    ;

        if (userEntity != null) {
            userToEntity (user, userEntity);

            datastore.put(userEntity);

            user.setId(userEntity.getKey().getId());

            return user;
        } else {
            throw new UserNotFoundException("Usuário " + email
                + " não encontrado");
        }
    } else {
        throw new UserAlreadyExistsException("Usuário " + user.getEma
il()
                + " já existe");
    }
}
```

```
    }  
}
```

Rpare na assinatura do método que ele lança duas exceções, cada uma para representar um dos possíveis erros que o método pode encontrar em sua execução, de acordo com os parâmetros de entrada. Veja também que o método recebe como parâmetro o endereço de e-mail, como uma String. Isso é necessário para que o usuário seja localizado por esse parâmetro, e não pelo endereço contido dentro do objeto `User`. Isso porque ele contém todas as modificações que serão feitas – que, inclusive, podem ser no próprio endereço de e-mail.

Buscar por e-mail

O método para buscar um usuário pelo seu e-mail deve, obviamente, receber esse endereço como parâmetro de entrada. A diferença aqui será a resposta que, em vez de ser um objeto do tipo `User`, será um `Optional` com o tipo `User`.

```
public Optional<User> getByEmail (String email)
```

Com a utilização do `Optional`, é possível informar dois estados no retorno do método:

- Que o usuário foi encontrado, já com suas informações no objeto do tipo `User`;
- Que o usuário não foi encontrado, sem ter de lançar uma exceção para esse caso.

Você verá mais adiante, na construção do *controller* de usuários, que essa estratégia deixa o código um pouco mais sofisticado do que usar exceções.

No trecho a seguir, note como ele deve ficar por completo. A explicação de como ele funciona está logo a seguir:

```
public Optional<User> getByEmail (String email) {  
    log.info("User: " + email);  
  
    DatastoreService datastore = DatastoreServiceFactory  
        .getDatastoreService();  
  
    Query.Filter filter = new Query.FilterPredicate(PROPERTY_EMAIL,  
                                                Query.FilterOp  
rator.EQUAL, email);  
  
    Query query = new Query(USER_KIND).setFilter(filter);  
  
    Entity userEntity = datastore.prepare(query).asSingleEntity();  
  
    if (userEntity != null) {  
        return Optional.ofNullable(entityToUser(userEntity));  
    } else {  
        return Optional.empty();  
    }  
}
```

Veja que, após o usuário ter sido pesquisado no Google Cloud Datastore, com um filtro pelo e-mail, há uma validação para saber se ele foi encontrado ou não. Caso seja encontrado, o objeto do tipo User é encapsulado dentro de um Optional. Caso contrário, esse mesmo objeto é retornado vazio, evitando assim que um null seja retornado ou uma exceção seja lançada.

Esse método também será chamado pelo mecanismo de autenticação quando precisarmos validar o usuário da requisição de entrada na aplicação.

Buscar todos usuários

O método para buscar todos os usuários é bem mais simples,

pois não existem condições de erro a serem tratadas, uma vez que a lista pode ser vazia caso não haja nenhum usuário na base. Veja como deve ficar seu código e sua explicação logo a seguir:

```
public List<User> getUsers() {  
    List<User> users = new ArrayList<>();  
    DatastoreService datastore = DatastoreServiceFactory  
        .getDatastoreService();  
  
    Query query;  
    query = new Query(USER_KIND).addSort(PROPERTY_EMAIL,  
                                         Query.SortDirection.ASCEND  
                                         ING);  
  
    List<Entity> userEntities = datastore.prepare(query).asList(  
        FetchOptions.Builder.withDefaults());  
  
    for (Entity userEntity : userEntities) {  
        User user = entityToUser(userEntity);  
  
        users.add(user);  
    }  
  
    return users;  
}
```

Note que a busca dos usuários, definida na linha 7, é ordenada de forma crescente pelo campo e-mail, apenas para demonstrar algo a mais além de uma busca simples no Google Cloud Datastore.

Apagar usuário

O método para excluir um usuário deve:

- Receber o seu endereço de e-mail como parâmetro de entrada;
- Excluir o usuário encontrado;
- Retornar o objeto do tipo `User` completo que foi excluído;

- Lançar a exceção `UserNotFoundException` caso o usuário não seja encontrado pelo e-mail.

Dito isso, esse método deve ficar da seguinte forma:

```
public User deleteUser (String email) throws UserNotFoundException {
    DatastoreService datastore = DatastoreServiceFactory
        .getDatastoreService();

    Query.Filter userFilter = new Query.FilterPredicate(PROPERTY_EMAIL,
        Query.FilterOperator.EQUAL, email);

    Query query = new Query(USER_KIND).setFilter(userFilter);

    Entity userEntity = datastore.prepare(query).asSingleEntity();

    if (userEntity != null) {
        datastore.delete(userEntity.getKey());

        return entityToUser(userEntity);
    } else {
        throw new UserNotFoundException("Usuário " + email
            + " não encontrado");
    }
}
```

Perceba que, nesse caso, lançar uma exceção se o usuário não for encontrado é mais apropriado, pois indica um problema de forma mais clara do que retornar um `Optional` vazio.

Inicialização do repositório

Como os usuários armazenados no Google Cloud Datastore serão usados pelo mecanismo de autenticação, é necessário que **sempre** haja pelo menos um usuário com papel de administrador na aplicação. Com ele, poderemos cadastrar os demais usuários.

Se a aplicação inicializar com a base de usuários vazia, nenhuma requisição poderá ser autenticada, pois a base estará vazia. Para evitar esse impasse, podemos construir um método a ser chamado na inicialização do repositório de usuários. Ele será responsável por garantir que exista pelo menos um usuário na base.

Para criar tal método, basta utilizar a anotação `@PostConstruct`, indicando que ele deverá ser chamado automaticamente após a construção do repositório de dados. Veja como o método deverá ficar e sua explicação logo a seguir:

```
@PostConstruct  
public void init(){  
    User adminUser;  
    Optional<User> optAdminUser = this.getByEmail("matilde@siecola.com.br");  
    try {  
        if (optAdminUser.isPresent()) {  
            adminUser = optAdminUser.get();  
            if (!adminUser.getRole().equals("ROLE_ADMIN")) {  
                adminUser.setRole("ROLE_ADMIN");  
                this.updateUser(adminUser, "matilde@siecola.com.br");  
            }  
        } else {  
            adminUser = new User();  
            adminUser.setRole("ROLE_ADMIN");  
            adminUser.setEnabled(true);  
            adminUser.setPassword("matilde");  
            adminUser.setEmail("matilde@siecola.com.br");  
            this.saveUser(adminUser);  
        }  
    } catch (UserAlreadyExistsException | UserNotFoundException e)  
    {  
        log.severe("Falha ao criar usuário ADMIN");  
    }  
}
```

Primeiramente, ele verifica se o usuário já existe na base,

pesquisando pelo endereço de e-mail. Caso não exista, um novo é criado com o papel de administrador. Se o usuário já existir na base, o método garante que ele possui o papel de administrador.

A existência desse método pode atrapalhar a execução dos testes da aplicação, que já vieram do *template* de criação do projeto para serem executados. Por isso, é necessário desabilitar a execução desses testes durante a inicialização da aplicação.

Acesse o menu `View -> Tool Windows -> Maven Projects` e, na tela que aparecer, clique no botão `Toggle 'Skip Tests' Mode` para desabilitar esse comportamento:

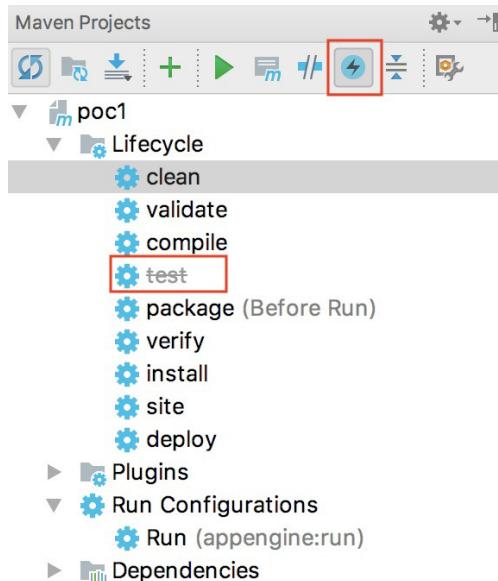


Figura 9.2: Desabilitando a execução de testes

Com isso, vamos conseguir subir normalmente, e não teremos nenhum problema causado pelo código do método `init` – que

tenta acessar a tabela de `Users` antes de o Google App Engine local estar de pé.

9.4 CONFIGURANDO O SPRING SECURITY PARA ACESSAR A BASE DE USUÁRIOS

Para que o Spring Security possa trabalhar com o repositório de usuários que você criou e desempenhar a função de autenticar as requisições, são necessárias três coisas importantes:

- Fazer com que o modelo de usuário implemente a interface `UserDetails`, que já foi feita no início deste capítulo;
- Criar um serviço a ser chamado durante o processo de autenticação, para buscar o usuário pelo seu e-mail;
- Dizer ao mecanismo de autenticação onde buscar a base de usuários.

Como a primeira implementação já foi feita, agora é necessário realizar as demais. Primeiramente, comece pelo serviço que busca o usuário pelo seu e-mail, criando um novo pacote chamado `br.com.siecola.gae_exemplo1.service` e, dentro dele, a classe `UserService`.

Esse serviço é simplesmente uma classe que implementa uma interface específica, obrigando a construção do método buscar pelo e-mail. Veja como deve ficar a sua declaração:

```
import br.com.siecola.gae_exemplo1.model.User;
import br.com.siecola.gae_exemplo1.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
```

```
oundException;
import org.springframework.stereotype.Service;

import java.util.Optional;

@Service("userDetailsService")
public class UserService implements UserDetailsService {

}
```

A anotação `@Service` indica ao Spring que, obviamente, essa classe deve ser encarada como um serviço, dentro do seu controle de injeção de dependências. Isso fará com o mecanismo de autenticação possa usar uma instância dessa classe sempre que precisar, sem a necessidade de controlar a sua criação.

A interface é o segredo para a junção do mecanismo de autenticação do Spring e a base de dados do usuário, pois ela obriga a implementação do método a seguir:

```
public UserDetails loadUserByUsername(String email) throws Userna
meNotFoundException
```

Esse método será invocado pelo Spring Security sempre que uma requisição precisar ser autenticada. Como o Spring Security não sabe como buscar o usuário no Google Cloud Datastore ou em outra base de dados de usuário, é nesse método que o trabalho será feito, de forma específica, utilizando o repositório de usuários já criado neste capítulo.

Veja que o método retorna um objeto do tipo `UserDetails`, uma interface do Spring para manipulação de informações de usuário durante o processo de autenticação. Isso não será um problema, pois a classe `User` (da aplicação `gae_exemplo1`) já foi construída implementando essa interface. Isso é necessário, pois o Spring não saberia lidar com a instância de usuário particular de

cada aplicação.

Para usarmos o repositório de dados, primeiramente é necessário injetá-lo nessa classe como um atributo privado:

```
@Autowired  
private UserRepository userRepository;
```

A anotação `@Autowired` do Spring controla o ciclo de vida da instância do objeto do tipo `UserRepository`, criado neste capítulo. Com ele, será possível acessar o Google Cloud Datastore.

Agora que tudo já está preparado, o método `loadUserByUsername` pode ser construído, como mostra o trecho a seguir:

```
@Override  
public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {  
    Optional<br.com.siecola.gae_exemplo1.model.User> optUser =  
        userRepository.getByEmail(email);  
    if (optUser.isPresent()) {  
        return optUser.get();  
    } else {  
        throw new UsernameNotFoundException("Usuário não encontrado")  
    }  
}
```

Nas linhas 3 e 4, perceba que o método `getByEmail` do repositório de usuários é usado para buscá-lo no Google Cloud Datastore, pois o Spring Security por si só não sabe como fazer isso. Com o `User` encapsulado em um `Optional`, o tratamento caso o usuário não seja encontrado fica mais elegante, simplesmente fazendo um teste para ver se há ou não um usuário presente na resposta (como é feito na linha 5).

Caso o usuário seja encontrado, ele é desempacotado do

`Optional` é devolvido como resposta. Porém, repare que o método devolve um objeto do tipo `UserDetails`, e não um `User`, sem nenhum tipo de conversão. Isso é possível graças à implementação da interface `UserDetails` pela classe `User` (da aplicação `gae_exemplo1`).

Se o usuário não for encontrado, a exceção `UsernameNotFoundException` do Spring será lançada. Com isso, ele saberá que um usuário que está tentando se autenticar não existe no Google Cloud Datastore, então, poderá negar a requisição.

O último passo a ser feito é fazer com o mecanismo do Spring Security utilize esse serviço, que você acabou de construir, dentro do seu mecanismo de autenticação. Apenas relembré que isso estava sendo feito de forma fixa no capítulo anterior, com apenas dois usuários, na configuração dentro da classe `SpringSecurityConfigHttpBasic`:

```
@Bean  
public UserDetailsService userDetailsService() {  
    User.UserBuilder users = User.withDefaultPasswordEncoder();  
    InMemoryUserDetailsManager manager = new InMemoryUserDetailsManager();  
    manager.createUser(users.username("user").password("user").roles("USER").build());  
    manager.createUser(users.username("admin").password("admin").roles("USER", "ADMIN").build());  
    return manager;  
}
```

Em vez de usarmos esse método com os usuários fixos em memória, agora será possível utilizá-los diretamente do Google Cloud Datastore, pelo serviço `UserService` criado. Veja como deve ficar o novo método no lugar do antigo:

```
@Autowired  
public void configureGlobalSecurity(AuthenticationManagerBuilder auth)  
throws Exception {  
    auth.userDetailsService(userDetailsService);  
}
```

A instrução na linha 4 diz ao mecanismo de autenticação do Spring Security que os usuários deverão ser localizados com o serviço do tipo `UserDetailsService`. Ele foi a interface implementada no serviço `UserService`, criado há pouco.

Como a senha do usuário está sendo persistida encriptada, é necessário criar a implementação do atributo `PasswordEncoder` utilizado em `UserRepository`, por isso adicione o seguinte método na classe `SpringSecurityConfigHttpBasic`:

```
@Bean  
public BCryptPasswordEncoder passwordEncoder() {  
    return new BCryptPasswordEncoder();  
}
```

Essa mesma instância será utilizada pelo mecanismo de autenticação do Spring, ou seja, a senha do usuário que foi encriptada no momento em que foi persistido poderá ser aberta para a verificação das credenciais do usuário.

Para concluir, é necessário injetá-lo na classe `SpringSecurityConfigHttpBasic`, para podermos utilizá-lo dentro dela, como no trecho a seguir:

```
@Autowired  
@Qualifier("userDetailsService")  
private UserDetailsService userDetailsService;
```

Esse princípio é semelhante ao usado na classe `UserService`, para a injeção do repositório `UserRepository`.

Tudo pronto! Agora o Spring Security já sabe buscar os usuários da aplicação `gae_exemplo1` no Google Cloud Datastore, na tabela `Users`. Isso fará com que as requisições a essa aplicação sejam autenticadas e autorizadas segundo as informações presentes nessa tabela.

9.5 CRIANDO O SERVIÇO DE USUÁRIOS

Agora que o repositório de usuários já foi construído e o mecanismo de autenticação já está utilizando-o, é hora de finalizar esse tópico com a criação do *controller* para expor as operações através de uma interface REST.

Para criar o serviço de usuário, é necessário antes planejar alguns requisitos, como qual será a URL de acesso, quais operações ele deverá ter e quais são as permissões de acesso a essas operações:

- **Requisito 1:** a URL base do serviço deverá ser `api/users` ;
- **Requisito 2:** deverá ter uma operação para listar todos os usuários, que poderá ser acessada somente por usuários com o papel `ADMIN` ;
- **Requisito 3:** a operação para cadastrar novos usuários só poderá ser feita por usuários com papel `ADMIN` . Não poderão existir usuários com e-mails repetidos;
- **Requisito 4:** a operação para alterar um usuário poderá ser feita somente por um usuário `ADMIN` ou pelo próprio usuário. Nesse último caso, ele não poderá alterar o papel (*role*). O e-mail do usuário deverá ser usado como parâmetro da operação para localizá-lo;
- **Requisito 5:** a operação para retornar todas as informações de um usuário só poderá ser feita por um usuário `ADMIN`

ou pelo próprio usuário. O e-mail do usuário deverá ser utilizado como parâmetro da operação para localizá-lo;

- **Requisito 6:** a operação para excluir um usuário só poderá ser feita por um usuário ADMIN ou pelo próprio usuário. O e-mail do usuário deverá ser usado como parâmetro da operação para localizá-lo.

Durante a implementação do serviço de usuários, novas funcionalidades relacionadas ao mecanismo de autenticação poderão ser estudadas, como:

- Identificação do usuário autenticado na requisição;
- Verificação do papel do usuário, dentro do código do método da operação do serviço.

Esses dois pontos são importantes para complementar o mecanismo de autorização, ou seja, algo que precise ser refinado e que não possa ser facilmente resolvido pelas anotações já vistas no capítulo anterior.

Criando a classe do serviço de usuários

Para começar com a implementação do serviço de usuários, primeiramente crie a classe de seu *controller* dentro do pacote `br.com.siecola.gae_exemplo1.controller`, com o nome de `UserController`:

```
import br.com.siecola.gae_exemplo1.exception.UserAlreadyExistsException;
import br.com.siecola.gae_exemplo1.exception.UserNotFoundException;
import br.com.siecola.gae_exemplo1.model.User;
import br.com.siecola.gae_exemplo1.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
```

```
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;
import java.util.logging.Logger;

@RestController
@RequestMapping(path="/api/users")
public class UserController {
    private static final Logger log = Logger.getLogger("UserController");
}
```

A primeira anotação define que essa é uma classe de *controller*, e a segunda define o caminho que esse serviço será acessado (nesse caso, /api/users). Isso cumpre o **requisito 1**.

Como a lógica de acesso à tabela de usuários do Google Cloud Datastore está toda encapsulada dentro do repositório de usuários criado, o `UserRepository`, você verá que o código do *controller* ficará muito mais simples e limpo. Porém, é necessário injetá-lo para poder ser usado:

```
@Autowired
private UserRepository userRepository;
```

Dessa forma, em todo momento que esse *controller* precisar acessar a base de dados do usuário, ele fará por meio dessa instância injetada aqui.

Criando a operação de buscar todos os usuários

Para implementação do **requisito 2**, basta criar um método que retorna todos os usuários, acessando a operação `getUsers` do

repositório de usuários:

```
@PreAuthorize("hasRole('ADMIN')")
@GetMapping
public List<User> getUsers() {
    return userRepository.getUsers();
}
```

A anotação `@PreAuthorize("hasAuthority('ADMIN')")` dá autorização somente para usuários com o papel `ADMIN` de acessarem essa operação.

Perceba que, com a lógica de acesso ao Google Cloud Datastore encapsulada no repositório de usuários, o código fica bem mais conciso. Essa operação dever ser acessada pela URI `/api/users`, com o verbo `HTTP GET`, como definido pelas anotações da classe e do método criado para a operação.

Criando a operação para salvar um usuário

Para o **requisito 3**, novamente basta utilizar o método específico do repositório de usuários para fazer o trabalho, como visto no trecho a seguir:

```
@PreAuthorize("hasRole('ADMIN')")
@PostMapping
public ResponseEntity<User> saveUser(@RequestBody User user) {
    try {
        return new ResponseEntity<User>(userRepository.saveUser(user),
                                         HttpStatus.OK);
    } catch (UserAlreadyExistsException e) {
        return new ResponseEntity<>(HttpStatus.PRECONDITION_FAILED);
    }
}
```

Parte do trabalho desse requisito, que é validar se já existe um usuário com o mesmo e-mail cadastrado, é feita no repositório de

usuários, como já foi explicado. Novamente, a primeira anotação desse método diz respeito à regra de autorização exigida na descrição do requisito.

Essa operação dever ser acessada pela URI `/api/users`, com o verbo `HTTP POST` e com as informações do usuário a ser salvo no corpo da requisição, no formato definido pelas anotações da classe e do método criado para a operação:

```
{  
    "email": "matilde@siecola.com.br",  
    "password": "matilde",  
    "fcmRegId": null,  
    "lastLogin": null,  
    "lastFCMRegister": null,  
    "role": "ROLE_ADMIN",  
    "enabled": true  
}
```

Perceba que não é necessário enviar o campo `id`, pois seu valor é preenchido pelo Google Cloud Datastore quando um novo objeto é criado.

Criando a operação para alterar um usuário

Para o **requisito 4**, será necessário conhecer informações do usuário autenticado, dentro do código do método que implementa a operação. Para isso, será o método que receberá um parâmetro a mais, injetado pelo próprio Spring Security que traz essas informações.

Veja como ele deverá ficar e sua explicação mais adiante:

```
@PreAuthorize("hasRole('USER') or hasRole('ADMIN')")  
@PutMapping(path = "/byemail")  
public ResponseEntity<User> updateUser(@RequestBody User user,  
                                         @RequestParam("email") Str
```

```
ing email,  
                                Authentication authentication  
ion)
```

O terceiro parâmetro, do tipo `Authentication`, traz informações do usuário autenticado que está fazendo a requisição. Será útil conhecê-lo para saber se ele pode ou não acessar a informação que solicita.

A anotação `PreAuthorize` agora recebe um parâmetro diferente, que é a função `hasAnyAuthority`, recebendo os papéis `USER` e `ADMIN`. Dessa forma, qualquer usuário que possua um desses papéis terá autorização.

O endereço de e-mail será recebido por um *query parameter*, passado na URI do serviço, por exemplo: `/api/users/byemail?email=doralice@siecola.com.br`. O parâmetro com o nome `email`, definido após `?` e com seu valor definido após `=`, é o chamado *query parameter*. Além disso, o parâmetro `User` traz as informações a serem alteradas no usuário.

Dentro do código desse método (e também de outros), será necessário saber qual é o papel do usuário autenticado. Para evitar código em duplicidade, crie um novo pacote chamado `br.com.siecola.gae_exemplo1.util`, com a classe `CheckRole`:

```
import org.springframework.security.core.Authentication;  
import org.springframework.security.core.GrantedAuthority;  
import org.springframework.security.core.userdetails.UserDetails;  
  
public class CheckRole {  
  
    private CheckRole(){}
  
  
    public static boolean hasRoleAdmin (Authentication authentication)
```

```

tion) {
        return hasRole(authentication, "ROLE_ADMIN");
    }

    public static boolean hasRoleUser (Authentication authentication) {
        return hasRole(authentication, "ROLE_USER");
    }

    private static boolean hasRole (Authentication authentication,
        String role) {
        UserDetails userDetails = (UserDetails) authentication.getPrincipal();
        for (GrantedAuthority grantedAuthority : userDetails.getAuthorities()) {
            if (grantedAuthority.getAuthority().equals(role)) {
                return true;
            }
        }
        return false;
    }
}

```

Os dois métodos públicos serão úteis para saber se o usuário autenticado possui o papel ADMIN ou USER .

Voltando ao método updateUser , você deverá implementar toda a lógica para fazer o que o **requisito 4** pede. A seguir, veja o código para isso e sua explicação logo após:

```

@PreAuthorize("hasRole('USER') or hasRole('ADMIN')")
@PutMapping(path = "/byemail")
public ResponseEntity<User> updateUser(@RequestBody User user,
                                         @RequestParam("email") String email,
                                         Authentication authentication)
{
    if ((user.getId() != null) && user.getId() != 0) {
        try {
            boolean hasRoleAdmin = CheckRole.hasRoleAdmin(authentication);
            UserDetails userDetails = (UserDetails) authentication.getPrincipal();

```

```

        if (hasRoleAdmin || userDetails.getUsername().equals(email))
    ) {
        if (!hasRoleAdmin) {
            user.setRole("USER");
        }
        return new ResponseEntity<User>(userRepository.updateUser
(user,
                           email), HttpStatus.OK);

    } else {
        return new ResponseEntity<>(HttpStatus.FORBIDDEN);
    }
} catch (UserNotFoundException e) {
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);
} catch (UserAlreadyExistsException e) {
    return new ResponseEntity<>(HttpStatus.PRECONDITION_FAILED)
;
}
} else {
    return new ResponseEntity<User>(HttpStatus.BAD_REQUEST);
}
}

```

A linha 6 verifica se a entidade `User` recebida possui o atributo `id` preenchido, pois isso é necessário pelo mecanismo de verificação de usuário já existente no repositório de usuários. A linha 8 verifica se o usuário tem o papel `ADMIN`. Essa informação será usada logo a seguir. A linha 9 coleta informações do usuário.

O teste na linha 11 verifica se usuário autenticado é `ADMIN`, ou se está tentando alterar o próprio usuário. Caso não seja, a operação responde com `HTTP 403 Forbidden`. Isso é importante para evitar que outro usuário, que não seja `ADMIN`, altere informações de outra pessoa.

O teste na linha 12 garante que somente usuários `ADMIN` possam alterar um usuário, colocando-o como `ADMIN`. Por último, na linha 15, o método `updateUser` do repositório de

usuários é chamado para fazer a alteração das informações do usuário em questão.

Essa operação dever ser acessada pela URI `/api/users/byemail?email={email}`, com o verbo HTTP PUT, com as informações do usuário a ser salvo no corpo da requisição, no mesmo formato apresentado na seção anterior.

Criando a operação para buscar um usuário

A operação para buscar o usuário deve usar algumas partes da lógica utilizada na operação de salvar um usuário, principalmente no que se refere às regras para permitir o usuário autenticado acesse somente as suas informações – a não ser que ele seja um administrador, como pede o **requisito 5**.

Veja como esse método deve ficar:

```
@GetMapping("/byemail")
public ResponseEntity<User> getUserByEmail(@RequestParam String email,
                                             Authentication authentication) {
    boolean hasRoleAdmin = CheckRole.hasRoleAdmin(authentication);
    UserDetails userDetails = (UserDetails) authentication.getPrincipal();

    if (hasRoleAdmin || userDetails.getUsername().equals(email)) {
        Optional<User> optUser = userRepository.getByEmail(email);
        if (optUser.isPresent()) {
            return new ResponseEntity<User>(optUser.get(), HttpStatus.OK);
        } else {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
    } else {
        return new ResponseEntity<>(HttpStatus.FORBIDDEN);
    }
}
```

}

Veja que o método recebe como parâmetro o endereço de e-mail pelo *query parameter*, além das informações do usuário autenticado, no parâmetro do tipo `Authentication`. Essas duas informações são usadas na lógica construída nas linhas 5, 6 e 8, em que é verificado se o usuário realmente tem permissão para acessar a informação do usuário que solicita por meio de seu endereço de e-mail.

O restante do método utiliza o repositório de usuários, buscando-o pelo método `getByEmail`, que retorna um `Optional`. Se o `Optional` retornado pelo método `getByEmail` estiver vazio, significa que o usuário não foi encontrado e o método retorna `HTTP 404 Not Found`. Caso contrário, o usuário encontrado é desempacotado do `Optional` e retornado pela operação.

Essa operação dever ser acessada pela URI `/api/users/byemail?email={email}`, com o verbo `HTTP GET`, como definido pelas anotações da classe e do método criado para a operação.

Criando a operação para apagar um usuário

A operação de excluir um usuário utilizará a mesma lógica para alterar ou buscar um usuário, devendo permitir que tal operação só seja feita por um usuário administrador ou pelo próprio usuário dono de suas informações. Veja como deve ficar:

```
@DeleteMapping(path = "/byemail")
public ResponseEntity<User> deleteUser(
    @RequestParam("email") String email, Authentication authentication) {
```

```

try {
    boolean hasRoleAdmin = CheckRole.hasRoleAdmin(authentication)
;
    UserDetails userDetails = (UserDetails) authentication.getPrincipal();

    if (hasRoleAdmin || userDetails.getUsername().equals(email))
{
        return new ResponseEntity<User>(userRepository.deleteUser(e
mail),
                                         HttpStatus.OK);
    } else {
        return new ResponseEntity<>(HttpStatus.FORBIDDEN);
    }
} catch (UserNotFoundException e) {
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);
}
}

```

Essa operação dever ser acessada pela URI `/api/users/byemail?email={email}` , com o verbo HTTP `DELETE` , como definido pelas anotações da classe e do método criado para a operação.

9.6 TESTANDO O SERVIÇO DE USUÁRIO COM O POSTMAN

Para testar o serviço de usuários com o Postman, proceda da mesma forma como fez com o serviço de produtos, atentando-se às seguintes modificações para se adequar às regras do serviço de usuários:

- A URI do serviço de usuários é `/api/users` ;
- Caso você não tenha modificado, as credenciais de acesso são:
 - Usuário: `matilde@siecola.com.br`
 - Senha: `matilde`

- O modelo para criação ou alteração de um usuário deve ser como no exemplo a seguir:

```
{  
  "email": "doralice@siecola.com.br",  
  "password": "doralice",  
  "fcmRegId": null,  
  "lastLogin": null,  
  "lastFCMRegister": null,  
  "role": "ROLE_USER",  
  "enabled": true  
}
```

Na verdade, todos os serviços da aplicação gae_exemplo1 terão o mesmo mecanismo de autenticação, utilizando os usuários que estiverem cadastrados na tabela `Users`, pois a fonte de dados informada ao Spring Security é a própria base de dados de usuários.

ACESSANDO O CONSOLE DE ADMINISTRAÇÃO LOCAL

Agora, para acessar o console de administração local da aplicação quando ela estiver em execução, pelo endereço http://localhost:8080/_ah/admin, é necessário digitar as credenciais de um usuário cadastrado.

Conclusão

Neste capítulo, você construiu o serviço de usuários, além de ter incrementado o mecanismo de autenticação com novas funcionalidades, como permissões baseadas em papéis. Aprendemos a fazer a ligação entre o mecanismo de segurança do

Spring Security ao Google Cloud Datastore, para que ele sirva como base de dados de usuários para a autenticação das requisições.

Muito do que foi feito aqui será aproveitado no capítulo *Protegendo serviços com OAuth 2.0*, quando o mecanismo de autenticação for trocado por um mais seguro e eficiente. O serviço de usuários ainda ganhará novas operações ao longo dos próximos capítulos.

No próximo capítulo, você aprenderá sobre uma funcionalidade muito interessante do Google App Engine: enviar mensagens para dispositivos móveis por meio de uma aplicação desenvolvida para o GAE. Isso equivale ao mecanismo conhecido como *Push Notification*, que auxilia muito os desenvolvedores do lado da aplicação móvel, como também os do lado da aplicação de back-end.

CAPÍTULO 10

ENVIANDO MENSAGENS COM O FIREBASE CLOUD MESSAGING

Se você é um usuário comum de aplicativos de smartphone (Android ou iOS), é provável que já tenha utilizado alguma função que permitia o recebimento de mensagens e notificações de algum servidor – por exemplo, de e-mail, comunicador ou uma rede social.

Você já parou para se perguntar como isso realmente funciona? Será que esse aplicativo fica a todo instante consultando o servidor para saber se há novas mensagens para ele? Será que quem desenvolveu o sistema teve de manter uma conexão aberta o tempo todo para receber essas notificações?

Com certeza algum desses mecanismos citados foi usado, mas ficar consultando o servidor a cada 5 segundos, por exemplo, traria uma quantidade de requisições muito grande para ele – uma vez que pode haver milhares de clientes conectados, querendo saber se há alguma mensagem nova.

Com o Firebase Cloud Messaging, é possível resolver todas essas questões de uma maneira bem simples, tanto para quem

desenvolve o lado servidor do sistema quanto para quem desenvolve o aplicativo móvel.

10.1 O QUE É FIREBASE CLOUD MESSAGING

O Firebase Cloud Messaging (FCM) é um **serviço gratuito** do Firebase. Ele permite que você envie e receba mensagens entre aplicações servidoras e clientes. Isso quer dizer que você pode desenvolver uma aplicação no GAE para enviar mensagens para aplicativos móveis (no Android, por exemplo), sem a preocupação de saber onde ele está localizado ou se está conectado à internet ou não.

Do lado da aplicação Android, você não precisa se preocupar em desenvolver códigos complexos para ficar consultando o servidor a todo o momento, nem se preocupar em estabelecer uma conexão com ele.

Visão geral da arquitetura do FCM

A arquitetura completa do FCM inclui a aplicação responsável por gerar as mensagens que se conectam ao backend do FCM, responsável por gerenciar todo o mecanismo de filas e de envio das mensagens para os aplicativos móveis. Além disso, ele cuida dos seus registros, para que possam ser localizados na internet. Veja a figura a seguir que ilustra tal arquitetura:

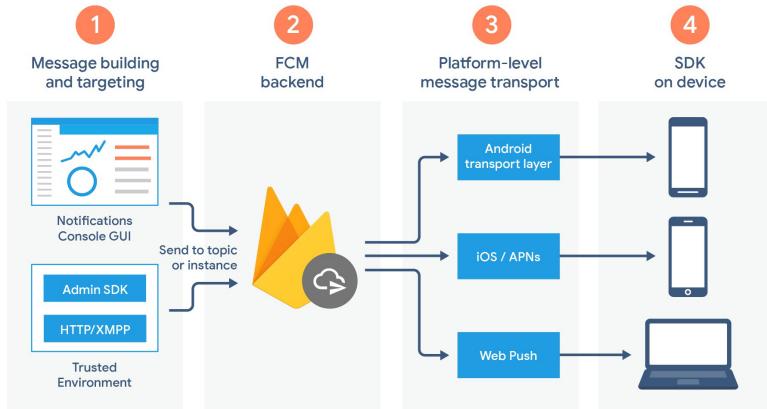


Figura 10.1: Arquitetura do FCM. Fonte: <https://firebase.google.com/docs/cloud-messaging/fcm-architecture>

É possível interagir com o backend do FCM de duas formas: conexão HTTP ou XMPP e utilizando o Admin SDK . Este livro trata exclusivamente do mecanismo de comunicação utilizando o Admin SDK , por ser mais simples e fácil de implementar.

Com esse mecanismo de conexão, é possível desenvolver uma aplicação para enviar mensagens (limitadas em até 4 KB de tamanho) para o backend do FCM em vários formatos, como o JSON. Nessa arquitetura o backend FCM é responsável por:

- Oferecer os mecanismos para que as aplicações móveis se registrem nele, por meio de uma identificação única da aplicação servidora que deseja enviar mensagens;
- Receber as requisições das aplicações para o envio de mensagens aos aplicativos móveis;
- Cuidar das filas de mensagens a serem entregues;
- Cuidar da entrega das mensagens aos aplicativos móveis;

- Garantir a tentativa de entrega das mensagens, mesmo que estes não estejam online.

Com isso, sua aplicação que envia as mensagens não precisa se preocupar com esses pontos; basta simplesmente enviar a mensagem para o destinatário desejado. Daí por diante, o backend do FCM faz todo o trabalho.

Conceitos importantes

O mecanismo do FCM possui alguns termos importantes que devem ser entendidos para conseguirmos trabalhar com ele, principalmente para os desenvolvedores das aplicações servidora e móvel:

- **Sender ID:** é a identificação da aplicação responsável por enviar as mensagens ao backend do FCM. Em uma aplicação do GAE, isso equivale ao `Project Number`, que pode ser obtido na aba `Project Settings` no menu de configurações do console do GAE, localizado no canto superior direito da tela.
- **API Key:** é a chave que garante o acesso à aplicação que deseja enviar mensagens ao backend do FCM.
- **Registration ID:** é a identificação da aplicação quando ela se registra no FCM para receber as notificações.

PLATAFORMAS CLIENTE SUPORTADAS PELO FCM

É possível desenvolver aplicações para as plataformas móveis Android e iOS, para se registrarem e receberem notificações de uma aplicação servidora. Também é possível desenvolver uma aplicação ou extensão para o Google Chrome.

10.2 ESTRATÉGIA

Antes de prosseguir com os passos necessários para incluir o FCM na aplicação `gae_exemplo1`, é interessante descrever, em linhas gerais, qual é a estratégia que será desenvolvida. A ideia é fazer com que, por meio de um novo serviço REST, possamos requisitar à aplicação `gae_exemplo1`, que envie uma informação de um produto armazenado no Google Cloud Datastore, para um dispositivo móvel (Android ou iOS), de um usuário que esteja registrado no FCM.

Veja o diagrama a seguir:

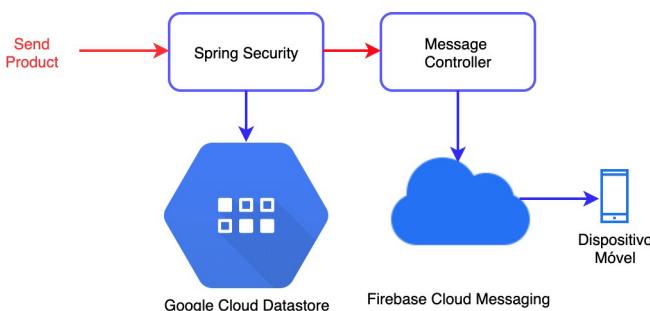


Figura 10.2: Diagrama da aplicação com FCM

O novo serviço será autenticado da mesma forma que os demais serviços existentes da aplicação, por isso a requisição para enviar a mensagem pelo FCM passará pelo Spring Security. Quando a requisição for autenticada e chegar ao *controller* de envio de mensagens, ele consultará o Google Cloud Datastore em busca do produto solicitado pela requisição, bem como o usuário que deverá receber as informações.

Dentro do usuário, existe um parâmetro chamado `fcmRegId`, que guarda um número (o registro no FCM do dispositivo móvel do usuário). Para mais informações sobre esse processo, veja a seção *O aplicativo móvel para Android* deste capítulo.

Quando as informações de usuário e produto estiverem disponíveis no método que trata a requisição para o envio de mensagens, a aplicação `gae_exemplo1` vai enviar uma requisição ao Firebase Cloud Messaging, solicitando o envio da mensagem com as informações do produto para o dispositivo móvel do usuário solicitado. Essa é uma requisição HTTP, que contém a chave de acesso da aplicação e um *payload* com a informação do produto.

Então, o backend do FCM vai se encarregar de entregar a mensagem ao dispositivo móvel – algo definido pelo valor que consta no atributo `fcmRegId` do modelo `User`, da aplicação `gae_exemplo1`. Quando o dispositivo móvel receber a mensagem, exibirá uma notificação ao usuário que, ao selecioná-la, exibirá as informações na tela do aplicativo (veja a seção *O aplicativo móvel para Android*).

Com isso, a aplicação `gae_exemplo1` será capaz de utilizar o serviço Firebase Cloud Messaging, para envio de notificações a

dispositivos móveis!

10.3 CONFIGURANDO O PROJETO NO GAE PARA UTILIZAR O FCM

Para configurar a aplicação `gae_exemplo1`, hospedada no GAE, é necessário criar uma credencial que dê permissão de acesso ao Firebase Cloud Messaging. Além disso, também é necessário vincular esse projeto ao Firebase em si.

Os passos a seguir descrevem o que deve ser feito até o fim do processo. É importante que isso seja feito antes de preparar o código-fonte do projeto.

Para começar, vá até o console do Google Cloud Platform e accesse o menu `APIs & Services`, como mostra a figura a seguir:

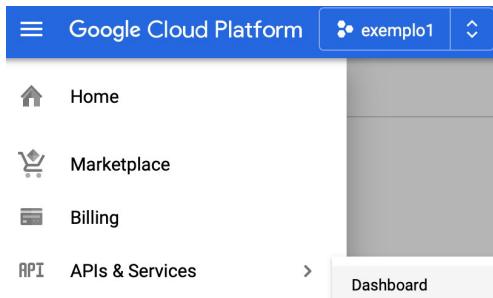


Figura 10.3: Menu de acesso a APIs e Services

Nesse menu, clique na opção `Dashboard`. Na página que abrir, clique no botão localizado no canto superior esquerdo, chamado `Enable APIs and Services`. Isso fará com que apareça uma página com todos os serviços disponíveis. Localize o serviço `Cloud Messaging`, como mostra a figura a seguir:

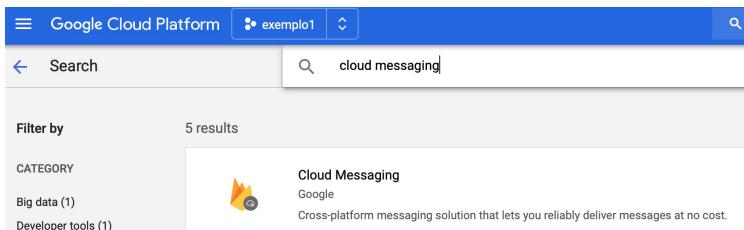


Figura 10.4: Localizando o Cloud Messaging

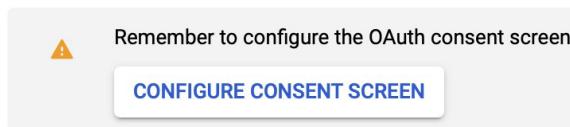
Quando encontrar a opção `Cloud Messaging`, clique nela e em seguida habilite-a através do botão `Enable`. Após esse serviço ter sido habilitado, clique no botão no canto superior direito `Create Credentials` para iniciar com o processo de criação das credenciais que darão acesso a esse serviço através da aplicação `gae_exemplo1`.

Na tela de criação das credenciais, escolha a opção `Cloud Messaging` e clique no botão `What credentials do I need?`, como mostra a figura a seguir:

A screenshot of the "Add credentials to your project" page. On the left, there's a sidebar with icons for API, Cloud Storage, Compute Engine, Functions, and Pub/Sub. The main content area has a heading "Add credentials to your project". Below it, a section titled "1 Find out what kind of credentials you need" says "We'll help you set up the correct credentials. If you wish you can skip this step and create an API key". A question "Which API are you using?" is followed by a list: "Different APIs use different auth platforms and some can only call certain APIs." Under this list, "Cloud Messaging" is selected. At the bottom, a blue button says "What credentials do I need?".

Figura 10.5: Habilitando a credencial de acesso

Para finalizar, clique em `Done`. Isso fará com que a chave de acesso seja gerada:



API Keys

<input type="checkbox"/>	Name	Creation date	Restrictions
<input type="checkbox"/>	⚠ API key 1	Apr 5, 2020	None

Figura 10.6: Credencial gerada

Essa API Key que foi gerada será utilizada pela aplicação `gae_exemplo1` para envio de mensagens pelo FCM.

É importante lembrar que não existem restrições de utilização dessa API Key, como alertado pelo aviso amarelo exibido nessa página. Em um ambiente de produção, é **IMPORTANTE CONFIGURAR RESTRIÇÕES DE ACESSO E UTILIZAÇÃO DESSA API KEY**. Se desejar, clique no botão `Configure Consent Screen` para ser redirecionado à página com tais configurações.

Agora é necessário vincular o projeto do GAE ao Firebase, para que ele finalmente possa entregar mensagens através do FCM. Para isso, entre no console do Firebase:

<https://console.firebaseio.google.com> e clique em Add project .

Na tela com a lista, selecione o projeto criado no Google App Engine, como mostra a figura a seguir:

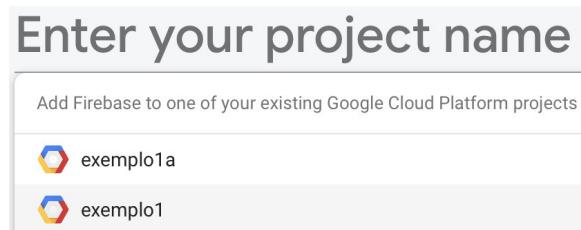


Figura 10.7: Selecionando o projeto do GAE

Depois de selecionar o projeto, clique em Continue e siga o processo para a criação do projeto no Firebase ligado ao do Google App Engine.

Depois que o projeto do Firebase for criado, associe um aplicativo Android a ele, como mostra a figura a seguir:

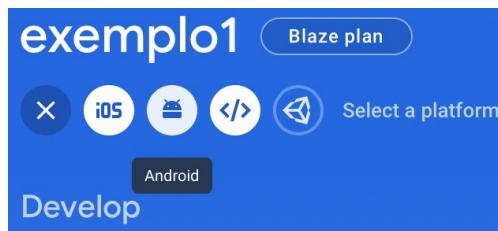


Figura 10.8: Associando um aplicativo Android

Depois de selecionar a opção Android, registre as informações do aplicativo Android a ser adicionado:

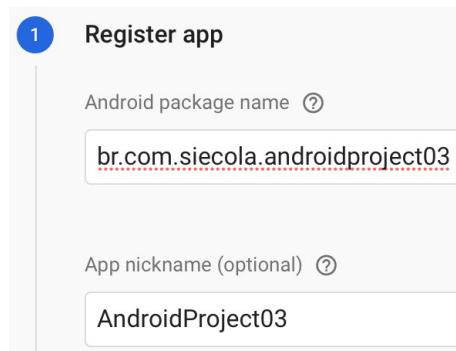


Figura 10.9: Registrando o aplicativo

É importante ressaltar que o nome do pacote da aplicação Android deve ser igual ao que será associado a esse projeto. Esse é o nome do pacote da aplicação de exemplo, em: https://github.com/siecola/android_project03.

Para continuar, clique em `Register app` e faça o download do arquivo `google-services.json`

O arquivo `google-services.json` deve ser copiado para dentro da pasta `app` do projeto Android antes de ser compilado e executado. Esse arquivo contém instruções ao aplicativo Android para se conectar no Firebase Cloud Messaging e outros serviços do Firebase, isso significa que cada projeto no Firebase possui um arquivo `google-services.json` diferente.

Esse arquivo não deve ser armazenado em um repositório de acesso público, pois contém informações sobre seu projeto do Firebase.

Para finalizar o processo, clique em `Next`, através das páginas com instruções de como preparar o projeto Android.

Para que a aplicação servidora possa se conectar ao backend do FCM e enviar mensagens a dispositivos móveis (com Android e iOS), é necessário apenas adicionar uma biblioteca no arquivo `pom.xml` do projeto `gae_exemplo1`, descrita a seguir.

```
<dependency>
    <groupId>com.google.firebaseio</groupId>
    <artifactId>firebase-admin</artifactId>
    <version>6.8.1</version>
</dependency>
```

Em termos de dependências, para que o projeto `gae_exemplo1` fique apto a trabalhar com o FCM, só precisamos fazer isso.

10.4 O APLICATIVO MÓVEL PARA ANDROID

No repositório https://github.com/siecola/android_project03 você encontra uma aplicação Android de exemplo, que se registra para receber mensagens pelo FCM. Você precisará de um dispositivo Android real para poder executar essa aplicação ou um emulador com o Google Play Services instalado. Obviamente, você também vai precisar do Android Studio

(<https://developer.android.com/studio/index.html>) para poder compilar e executar a aplicação.

Quando a aplicação é executada, e o dispositivo possui acesso à Internet, ela se registra no Firebase Cloud Messaging e ganha uma chave única que define seu registro, como pode ser observado na figura a seguir:

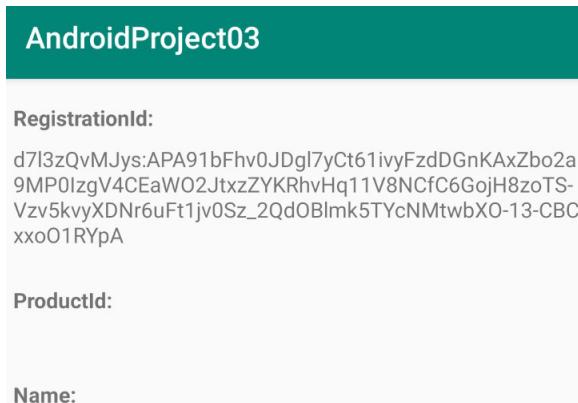


Figura 10.10: Aplicação registrada no FCM

Quando uma mensagem é recebida, uma notificação aparece na barra de notificações do dispositivo. A aplicação foi projetada para receber o modelo de produtos da aplicação `gae_exemplo1`, exibindo suas informações em seus campos corretos (como você pode ver na figura mais adiante).

Se o usuário clicar na notificação de mensagem recebida, ela será exibida na tela principal da aplicação, mostrando a informação do produto. Veja a figura a seguir da tela da aplicação Android de exemplo:

AndroidProject03

RegistrationId:

d7l3zQvMJys:APA91bFhv0JDgl7yCt61ivyFzdDGnKAxZbo2a9MP0IzgV4CEaW02JtxzZYKRhvHq11V8NCfC6GojH8zoTS-Vzv5kvyXDNr6uFt1jv0Sz_2Qd0Blmk5TYcNMtwbXO-13-CBCxxo01RYpA

ProductId:

3

Name:

Nome 3

Model:

Model 3

Code:

3

Price:

\$ 30.00

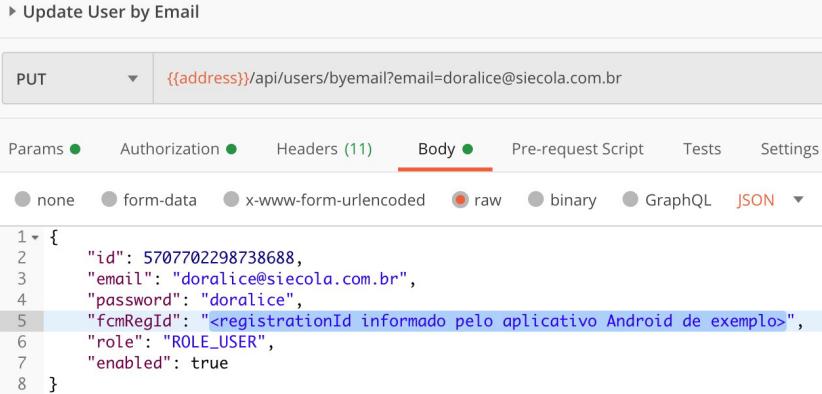
Figura 10.11: Mensagem recebida e exibida

O valor obtido no campo `RegistrationId` é o que deve ser utilizado quando a aplicação `gae_exemplo1` desejar enviar uma mensagem para esse dispositivo. Esse campo permite que seu valor seja copiado para a área de transferência do Android. Esse mesmo valor é o que deve ser salvo na propriedade `fcmRegId` do tipo `User` no Google Cloud Datastore, criado no capítulo *Adicionando o serviço de usuários*.

Em uma aplicação real no Android, esse processo de registro no FCM seria feito em *background* e, em seguida, ela deveria atualizar essa informação por meio do serviço de usuários,

atualizando o valor do `RegistrationId` obtido. Porém, isso não está sendo feito por essa aplicação de exemplo do Android.

Por isso, você deve copiar o valor do campo `RegistrationId` e atualizá-lo na operação citada do serviço de usuários, como mostra a figura a seguir:



The screenshot shows the Postman interface with the following details:

- Method: PUT
- URL: `{(address)}/api/users/byemail?email=doralice@siecola.com.br`
- Body tab selected, showing JSON content:

```
1 {  
2   "id": 5707702298738688,  
3   "email": "doralice@siecola.com.br",  
4   "password": "doralice",  
5   "fcmRegId": "<registrationId informado pelo aplicativo Android de exemplo>",  
6   "role": "ROLE_USER",  
7   "enabled": true  
8 }
```

Figura 10.12: Atualizando o RegistrationId do usuário

Isso fará com que a aplicação `gae_exemplo1` armazene o `RegistrationId` da aplicação atrelada a esse usuário, para que consiga enviar mensagens a ele através do FCM.

A seção seguinte vai detalhar o que deve ser feito, na aplicação do `gae_exemplo1` do GAE, para enviar mensagens a dispositivos móveis com o FCM.

10.5 ENVIANDO MENSAGENS A APlicativos móveis COM O FCM

Para utilizar o FCM e enviar mensagens a dispositivos móveis,

que foram previamente registrados, é necessária a seguinte informação:

- O **Registration ID** do dispositivo que deve receber a mensagem da aplicação servidora. Na aplicação `gae_exemplo1`, essa informação está atrelada a cada usuário, por meio da propriedade `fcmRegId` do tipo `User`, armazenada no Google Cloud Datastore. Quem deve fornecer esse valor é a aplicação móvel que se registrar no FCM. Cada dispositivo registrado no FCM possui um valor de registro diferente.

Para ilustrar melhor todo o mecanismo, crie um novo serviço, responsável por fazer esse trabalho na aplicação `gae_exemplo1`. Para isso, crie uma nova classe chamada `MessageController` no pacote `br.com.siecola.gae_exemplo1.controller`, onde as demais classes de `controller` estão:

```
@RestController
@RequestMapping(path="/api/message")
public class MessageController {
    private static final Logger log = Logger.getLogger("MessageController");

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private ObjectMapper objectMapper;
}
```

Esse novo serviço será acessado pela URL `api/message`. Para tornar o exemplo mais interessante, será enviado um dos produtos cadastrados no Google Cloud Datastore como mensagem pelo FCM aos aplicativos móveis, usando o formato JSON.

Dentro dessa classe, crie o método `initialize()` para ser chamado assim que a classe for instanciada, para inicializar as configurações do cliente do Firebase dentro da aplicação `gae_exemplo1`:

```
@PostConstruct  
public void initialize() {  
    try {  
        FirebaseOptions options = new FirebaseOptions.Builder()  
            .setCredentials(GoogleCredentials.getApplicationDefault())  
            .setDatabaseUrl("https://<project-id>.firebaseio.com")  
            .build();  
        FirebaseApp.initializeApp(options);  
        log.info("FirebaseApp configurado");  
    } catch (IOException e) {  
        log.info("Falha ao configurar FirebaseApp");  
    }  
}
```

Nesse trecho, substitua `<project-id>` pela identificação de seu projeto no GAE, o mesmo que compõe a URL base de acesso aos seus serviços.

Esse código fará com que o cliente do Firebase tenha acesso às credenciais criadas anteriormente, para permitir que ele possa enviar mensagens pelo FCM.

Dentro desse serviço, crie uma operação para efetivamente enviar as informações do produto pelo FCM. Essa operação deve receber dois parâmetros: o código do produto a ser enviado e o endereço do e-mail do usuário. Por meio do endereço de e-mail, será possível localizar o usuário e o valor do registro de sua aplicação móvel no FCM.

```
@PreAuthorize("hasRole('ADMIN')")  
@PostMapping(path = "/sendproduct")
```

```
public ResponseEntity<String> sendProduct(  
    @RequestParam("email") String email,  
    @RequestParam("productCode") int productCode) {  
}  
}
```

IMPORTS DE BIBLIOTECAS

Se tiver dúvidas sobre quais imports utilizar, consulte o código-fonte do projeto, no repositório <https://github.com/siecola/GAEBookV3Exemplo1>.

Essa operação deverá ser acessada pelo método HTTP POST pela URL `api/message/sendproduct?email={email}&productCode={productCode}`. Lembre-se de que essa operação também está protegida com o mecanismo de **HTTP Basic Authentication** e que somente usuários com papel ADMIN poderão acessá-la por conta da anotação `@PreAuthorize`, no início do método.

Para tornar o exemplo um pouco mais simples, o produto será localizado no Google Cloud Datastore por meio de um método privado, criado dentro da própria classe `MessageManager` (da mesma forma vista em capítulos anteriores). O usuário será localizado pelo repositório `UserRepository`, que será injetado nessa classe:

```
private Product findProduct (int code) {  
    DatastoreService datastore = DatastoreServiceFactory  
        .getDatastoreService();  
  
    Query.Filter codeFilter = new Query.FilterPredicate("Code", Q  
    uery.FilterOperator.EQUAL, code);
```

```

Query query = new Query("Products").setFilter(codeFilter);
Entity productEntity = datastore.prepare(query).asSingleEntity();
}

if (productEntity != null) {
    return ProductController.entityToProduct(productEntity);
} else {
    return null;
}
}
}

```

Para tornar a explicação mais direta e simples, transforme o método `entityToProduct` da classe `ProductController` pública e estática:

```
public static Product entityToProduct(Entity productEntity)
```

Com todas essas novas implementações e modificações, agora o método `sendProduct` pode começar a ser preparado para executar sua função, como detalhado a seguir. Veja como ele deve ficar:

```

@PreAuthorize("hasRole('ADMIN')")
@PostMapping(path = "/sendproduct")
public ResponseEntity<String> sendProduct(
    @RequestParam("email") String email,
    @RequestParam("productCode") int productCode) {

    Optional<User> optUser = userRepository.getByEmail(email);
    if (optUser.isPresent()) {
        User user = optUser.get();

        Product product = findProduct(productCode);
        if (product != null) {
            String registrationToken = user.getFcmRegId();
            //TODO - código para enviar a mensagem

        } else {
            log.severe("Produto não encontrado");
            return new ResponseEntity<String>("Produto não encontrado",

```

```

        HttpStatus.NOT_FOUND);
    }
} else {
    log.severe("Usuário não encontrado");
    return new ResponseEntity<String>("Usuário não encontrado",
        HttpStatus.NOT_FOUND);
}
}

```

Repare que existem dois tratamentos de erro:

- Caso o usuário não seja encontrado, a resposta será `HTTP 404 Not Found`, com uma mensagem informando o fato;
- Caso o produto não seja encontrado, a resposta será `HTTP 404 Not Found`, com uma mensagem informando o fato.

Outros tratamentos de erro serão adicionados ao mecanismo de envio da mensagem.

Agora é só implementar a lógica para enviar a mensagem pelo FCM, da mesma forma explicada no início desta seção, só que com os dados retirados do Google Cloud Datastore. Além disso, precisaremos colocar um pouco mais de tratamento, caso algo dê errado. Um exemplo seria o usuário não estar registrado no FCM, onde uma mensagem de erro é gerada no log, e o retorno `HTTP 404 Not Found` é devolvido como resposta:

```

//TODO - código para enviar a mensagem
try {
    Message message = Message.builder()
        .putData("product", objectMapper.writeValueAsString(p
roduct))
        .setToken(registrationToken)
        .build();

    String response = FirebaseMessaging.getInstance().send(messag
e);

```

```
log.info("Mensagem enviada ao produto " + product.getName());
log.info("Reposta do FCM: " + response);

return new ResponseEntity<String>("Mensagem enviada com o pro-
duto "
+ product.getName(), HttpStatus.OK);
} catch (FirebaseMessagingException | JsonProcessingException e)
{
    log.severe("Falha ao enviar mensagem pelo FCM: " + e.getMes-
sage());
    return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
}
```

Para ver o código completo dessa classe, visite o repositório de código da aplicação, no endereço <https://github.com/siecola/GAEBookV3Exemplo1>.

Se tudo der certo e o FCM enviar a mensagem, a operação retornará o código `HTTP 200 OK`, informando que a mensagem foi entregue ao FCM.

10.6 TESTANDO O ENVIO DE MENSAGENS PELO FCM

Para testar o funcionamento desse serviço, basta usar o Postman, passando um código de um produto armazenado no Datastore e o e-mail do usuário que você tenha cadastrado o `RegistrationId`, como mostra a figura a seguir:

The screenshot shows the Postman interface with the following details:

- Method:** POST
- URL:** {{address}}/api/message/sendproduct?email=doralice@siecola.com.br&productCode=2
- Headers (1):** Authorization (with value Basic bWF0aWxkZUBzaWVjb2xhLmNvbS5icjptYXR...)
- Body:** Mensagem enviada com o produto Name 2
- Preview:** Shows the JSON structure of the message body.

Figura 10.13: Requisição para envio de notificação de produto

Veja que o usuário que receberá a mensagem, bem como o código do produto que vai na notificação, estão sendo passados como parâmetros na URL.

O mecanismo do FCM é muito interessante e, ao mesmo tempo, fácil de ser utilizado. Como você pode observar, toda a tarefa pesada de enfileirar as mensagens e tentar entregá-las a cada dispositivo móvel fica a cargo do backend FCM. A aplicação deve simplesmente conectar-se nele e enviar a mensagem.

Conclusão

Parabéns! Você construiu um serviço REST que aciona o mecanismo do FCM para enviar mensagens a dispositivos móveis. Esse serviço pode ser usado por outras aplicações, que não estão rodando no GAE, para solicitar que tal trabalho seja feito.

No próximo capítulo, você aprenderá mais uma funcionalidade interessante que a plataforma do Google App Engine oferece:

agendamento de tarefas. Este é um recurso muito útil quando você deseja executar alguma tarefa, em um intervalo de tempo pré-determinado, utilizando a infraestrutura do GAE.

AGENDANDO TAREFAS NO GAE

Em aplicações Web, a necessidade de agendar tarefas para serem executadas de forma automática é muito comum. Como exemplo, temos: enviar e-mails durante a faixa de horário com menor tráfego de rede, executar limpeza de banco de dados ou arquivos de log etc.

11.1 COMO FUNCIONAM AS TAREFAS AGENDADAS NO GAE

Com o Google App Engine, é possível programar uma tarefa para ser executada automaticamente pela plataforma. Essa tarefa vai acessar um serviço da sua aplicação, utilizando o método `HTTP GET`. Dessa forma, basta criar um serviço e uma operação, especificamente para ser chamada pela tarefa agendada.

Dentro do método que tratará a requisição dessa operação, você pode realizar a atividade programada que for necessária. Essa operação que será acessada pela tarefa agendada é igual a qualquer outra, com as mesmas regras e com o mesmo tempo de execução e resposta.

Além disso, para o GAE considerar que a tarefa foi executada com sucesso pela sua aplicação, a operação deve retornar um código HTTP com os valores entre 200 e 299, inclusive. No GAE, as aplicações no regime gratuito da plataforma podem ter até 20 tarefas agendadas, enquanto as que estão com o mecanismo de cobrança ativado podem ter até 100.

TAREFAS AGENDADAS NO AMBIENTE DE DESENVOLVIMENTO

As tarefas agendas não funcionam no ambiente de desenvolvimento, então você deve obrigatoriamente publicar a aplicação para vê-las funcionando.

11.2 CRIANDO UM SERVIÇO AGENDADO

Para exemplificar o mecanismo de tarefas agendadas, crie um novo serviço com uma única operação, como imprimir uma mensagem no log da aplicação. Dessa forma, você poderá ver, pelo log de execução da aplicação, que a tarefa está sendo executada no tempo programado.

Para facilitar o trabalho e deixar o foco somente no agendamento da tarefa, o novo serviço não deverá passar pelo mecanismo de autenticação.

Para começar, crie um novo pacote na aplicação com o nome

de `br.com.siecola.gae_exemplo1.cron`. Nesse novo pacote, crie a classe que vai implementar o serviço e a operação que será chamada pela tarefa agendada:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.Calendar;
import java.util.logging.Logger;

@RestController
@RequestMapping(path="/api/cron")
public class CronService {
    private static final Logger log = Logger.getLogger("CronService");

    @GetMapping(path = "/testcron")
    public ResponseEntity<?> testCron() {
        log.severe("Cron message --- " + Calendar.getInstance().getTime());

        return new ResponseEntity<>(HttpStatus.OK);
    }
}
```

Como você pode ver, ele é um método comum para implementar uma operação normal, sem nada de novo. A sua URL de acesso será `/api/cron/testcron`, já que foi configurado pelas anotações na classe e no método. O que essa operação faz é apenas exibir uma mensagem no log da aplicação quando ela for chamada pela tarefa agendada.

Da forma como está, essa operação só poderia ser acessada por uma requisição autenticada, ou seja, com o cabeçalho `Authorization` presente na requisição. Porém, isso não é possível ser feito no mecanismo de tarefas agendadas do GAE.

Como dito anteriormente, para manter o foco somente na explicação de como tarefas agendadas funcionam no GAE, deixe que essa operação seja acessível a todos, sem exigir nenhuma autenticação. Para isso, volte à classe `SpringSecurityConfigHttpBasic` e configure a URL do novo serviço, `/api/cron/testcron`, para permitir requisições anônimas. Veja o trecho de código a seguir:

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.csrf().disable()  
        .authorizeRequests()  
            .antMatchers("_ah/**").permitAll()  
            .antMatchers("/api/cron/testcron/**").anonymous()  
        .anyRequest().authenticated()  
        .and().httpBasic()  
        .and().sessionManagement()  
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS);  
}
```

Perceba que, na linha 6, a URL `/api/cron/testcron` agora pode ser acessada sem a necessidade de autenticação, pois seu endereço foi adicionado às configurações de segurança, permitindo requisições não autenticadas.

11.3 CONFIGURANDO A TAREFA

Para configurar as tarefas agendadas da aplicação, é necessário criar o arquivo de nome `cron.xml`, dentro da pasta `src/main/webapp/WEB-INF/`. Nele, é possível configurar mais de uma tarefa, com agendamentos e URLs diferentes.

Veja no exemplo a seguir como deve ficar o agendamento de uma tarefa a ser executada a cada dois minutos para acessar o serviço criado na seção anterior:

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
  <cron>
    <url>/api/cron/testcron</url>
    <description>Mensagem do Cron a cada 2 minutos</description>
    <schedule>every 2 minutes</schedule>
  </cron>
</cronentries>
```

O campo `url` define o endereço da operação a ser chamada quando a tarefa for executada. No campo `description`, você pode colocar um texto descritivo sobre a tarefa. Essa informação aparecerá no console do GAE, na seção de gerenciamento das tarefas agendadas. Por fim, no campo `schedule`, você deve definir a periodicidade de sua execução.

Formato do agendamento

As tarefas podem ser agendadas para serem executadas de diversas formas:

- Com uma periodicidade de tempo (minutos, horas ou dias), por meio do parâmetro `every` ;
- Com uma periodicidade de tempo (minutos, horas ou dias) e dentro de um intervalo de horas, combinando o parâmetro `every` com o `from` e `to` ;
- A cada hora de todos os dias, ou de dias específicos;
- Dentro de intervalos de tempo, com uma periodicidade, todos os dias ou em dias específicos, por exemplo, utilizando a expressão `every monday 09:00` .

Para mais exemplos sobre o formato a ser usado no campo `schedule` do arquivo `cron.xml`, para configurar a periodicidade da execução, consulte a documentação do Google:

<https://cloud.google.com/appengine/docs/standard/java/config/cron-yaml>.

11.4 ACOMPANHANDO A EXECUÇÃO DO CONSOLE DO GAE

Para visualizar a tarefa em execução no console do GAE, é necessário que você publique sua aplicação antes. Na tela do IntelliJ IDEA, para publicar a aplicação, certifique-se de marcar a opção para atualizar os arquivos de configuração agendada:

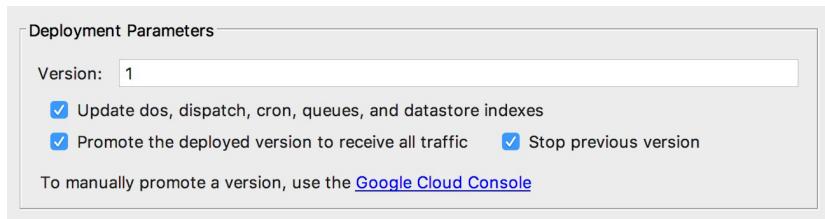


Figura 11.1: Publicando com tarefa agendada

Veja que a opção `Update dos, dispatch, cron, queues and datastore indexes` está marcada para que o arquivo `cron.xml` seja publicado.

Depois de publicar, vá ao console do GAE, dentro do menu App Engine da sua aplicação `gae_exemplo1`, e acesse a opção Cron Jobs. Nessa aba, você encontra as tarefas agendas criadas, como apresentado neste capítulo.

Nessa aba, você verá as estatísticas de execução da tarefa agendada, o endereço da operação chamada, sua descrição, a frequência de execução, a última vez que ela rodou e se foi executada com sucesso ou não.

A cron job is a scheduled task that runs at a specific time or at regular intervals.

Cron job ^	Description	Frequency	Last run	Status	Log
/api/cron/testcron	Mensagem do Cron a cada 2 minutos	every 2 minutes (GMT)	2020-03-29 (16:37:00) On time	Success	View Run now

Figura 11.2: Monitorando as tarefas agendadas

Todas as informações presentes nessa tela vieram do arquivo de configuração `cron.xml`, criado dentro da sua aplicação. Como a tarefa criada simplesmente exibe uma mensagem no log, vá até a seção que exibe as mensagens no console do GAE para se certificar de que ela realmente está escrevendo a mensagem a cada 2 minutos, como foi programada.

Showing logs from the last hour ending at 12:38 PM (PDT) Download logs View Options ▾

↓ No older entries found matching current filter in the last hour. Load older logs ↓

▼ !! 2020-03-29 12:37:00.531 PDT GET 200 284 B 28 ms AppEngine... ⋮

/api/cron/testcron

```
0.1.0.1 - - [29/Mar/2020:16:37:00 -0300] "GET /api/cron/testcron HTTP/1.1" 200 284 - "AppEngine-Google; (+http://code.google.com/appengine)" "elegant-rock-272515.appspot.com" ms=28 cpu_ms=3353 cpm_usd=3.1739e-8 loading_request=0 instance=00c61b117cf0e662e56ab64 1055b6b40d35db02987ea8cd06487ca2db1554995cf49ec1857 app_engine_release=1.9.71 trace_id=9f2604b87b5b0f812ecde44a09f6168e
```

▼ { Expand all | Collapse all

- httpRequest: {...}
- insertId: "5e80f8dc0008900c5625c439"
- labels: {...}
- logName: "projects/elegant-rock-272515/logs/appengine.googleapis.com%2Frequest_log"
- operation: {...}
- protoPayload: {...}
- receiveTimestamp: "2020-03-29T19:37:00.577142697Z"
- resource: {...}
- severity: "ERROR"
- timestamp: "2020-03-29T19:37:00.531960Z"
- trace: "projects/elegant-rock-272515/traces/9f2604b87b5b0f812ecde44a09f6168e"
- traceSampled: true

} !!

!! 2020-03-29 12:37:00.548 PDT br.com.siecola.gae_exemplo1.cron.CronService testCron: Cron message --- Sun Mar 29 19:37:00 UTC 2020 (`CronService.java:19`)

Figura 11.3: Mensagem da tarefa agendada

Nesse exemplo, note que o log selecionado exibe a mensagem criada dentro da operação `testCron`, que está sendo invocada pela tarefa agendada a cada 2 minutos!

Conclusão

Neste capítulo, você aprendeu mais uma funcionalidade da plataforma do Google App Engine: o agendamento de tarefas para serem executadas de forma automática, dentro da sua aplicação e sem grandes esforços. Basta configurar um arquivo XML.

No próximo capítulo, você utilizará outra funcionalidade interessante do GAE, o *Memory Cache*, usado para manter dados na memória, sem utilizar o Google Cloud Datastore.

CAPÍTULO 12

UTILIZANDO MEMORY CACHE

O mecanismo de autenticação de usuários da aplicação `gae_exemplo1` atualiza a data/hora na propriedade `lastLogin`, no tipo `users` do Google Cloud Datastore, toda vez que ele acessa o sistema. Essa informação pode ser usada para saber, por exemplo, qual foi a última vez que ele acessou o sistema e também quais foram os usuários que há muito tempo não utilizam a aplicação.

Como você deve perceber, essa informação é muito útil para fins estatísticos, pois é importante saber se os usuários da sua aplicação são assíduos. Porém, é bem provável que você não necessite de uma informação tão precisa, a ponto de indicar exatamente a data/hora do último acesso, com minutos e segundos de precisão. Ter a precisão em minutos ou horas já bastará.

Então, como fazemos para gerenciar todas as requisições de autenticação e, ao mesmo tempo, atualizar a data/hora do último acesso de apenas algumas requisições? E fazer tudo isso em intervalos de tempo de alguns minutos, para fins estatísticos? Isso pode ser feito com um serviço muito interessante da plataforma do Google App Engine, chamado **MemCache**.

12.1 O QUE É MEMCACHE

O MemCache é uma funcionalidade do GAE que permite que a aplicação em Java possa salvar em memória informações sob a forma de **conjuntos chave/valor**, sem a persistência desses dados – como é feito no Google Cloud Datastore.

Como esse mecanismo faz o seu trabalho em memória e sem acesso a disco, o tempo de leitura e escrita é muito mais rápido do que no Google Cloud Datastore. Por isso, ele pode ser usado para armazenar valores temporários, sem a necessidade de persisti-los.

Podemos utilizar o MemCache em situações como:

- Armazenamento de preferências de usuários;
- Evitar consultas e/ou escritas sucessivas ao Google Cloud Datastore, de informações que certamente não vão se alterar a todo instante;
- Armazenar informações temporárias, dentro do contexto de algumas requisições, em um período de tempo conhecido.

É importante ressaltar que as informações guardadas no MemCache ficarão armazenadas o maior tempo possível, caso não seja explicitamente declarado um tempo para sua expiração. Claro que isso também dependerá da utilização e do espaço disponível da plataforma, gerenciados pelo GAE.

Os dados guardados no MemCache são de comum acesso por toda a aplicação, tornando o seu uso muito simples, pois um dado pode ser salvo por uma operação de um serviço e utilizado por outro completamente diferente – em que precisamos saber apenas

a chave de seu valor. Isso vale também para as múltiplas instâncias de uma mesma aplicação, já que todas podem acessar os dados do MemCache que estão sob sua propriedade.

Além disso, na modalidade gratuita do serviço, a infraestrutura de armazenamento dos dados é compartilhada com as demais aplicações do GAE, sendo da mesma aplicação ou não. Para obter uma infraestrutura exclusiva para a sua aplicação, além de maior capacidade de armazenamento, é necessário optar pela opção do serviço pago.

Limites de utilização do MemCache

O serviço MemCache do GAE possui alguns limites de uso, tais como:

- O tamanho máximo de um valor a ser armazenado é de 1 MByte, menos o tamanho da chave e do *overhead* gerado pela implementação de acesso ao MemCache, como o JCache, que será utilizado a seguir;
- O tamanho da chave não pode ser maior do que 250 bytes;
- O valor da chave não pode ser nulo.

12.2 UTILIZANDO JCACHE

O MemCache pode ser usado pela sua API nativa, acessando diretamente o serviço do GAE. Porém, o SDK do App Engine possui uma implementação do JCache (JSR 107) para acessá-lo, o que torna o trabalho muito mais fácil e prático.

Para utilizar o MemCache com o JCache, basta usar a interface `javax.cache.Cache`. Dessa forma, uma instância do objeto do

tipo Cache pode ser obtida por meio do CacheFactory , como pode ser visto no trecho de código a seguir:

```
try {
    CacheFactory cacheFactory = CacheManager.getInstance().getCacheFactory();
    cache = cacheFactory.createCache(Collections.emptyMap());

} catch (CacheException e) {
    //Tratar exceção
}
```

Com a instância do tipo Cache , é possível inserir e localizar valores no cache, como descrito nas seções seguintes, com exemplos genéricos.

Inserindo e buscando valores no MemCache

De posse da instância de Cache , para inserir um valor no MemCache, basta executar o método put . Lembre-se de que o método put pode receber valores do tipo Object como parâmetros.

```
cache.put("chave", "valor");
```

Para ler algum valor do MemCache, basta utilizar o método get :

```
String valor = (Date)cache.get("chave");
```

Se a chave não existir no MemCache, o objeto retornado será nulo.

Verificando se uma chave existe no MemCache

Para verificar se uma chave existe no Memcache, usamos o método containsKey . Este recebe como argumento a chave de

pesquisa, retornando verdadeiro, caso encontre; caso contrário, falso. Isso é necessário para validar se algum valor existe no cache, antes de lê-lo.

Removendo uma chave do MemCache

Para remover uma chave do MemCache, basta usar o método `remove`, que recebe como parâmetro o valor da chave a ser removida:

```
cache.remove("chave");
```

A remoção de um valor do cache é útil em um caso que você tenha certeza de que ele não é mais útil.

12.3 USANDO MEMCACHE NO MECANISMO DE AUTENTICAÇÃO

Nesta seção, veremos como utilizar o serviço MemCache do GAE para criar um mecanismo que persiste a informação do último login do usuário, na propriedade `lastLogin` da entidade `Users`. Porém, faremos isso somente se ele acessou algum serviço dessa aplicação há mais de 30 segundos.

Como toda requisição passa pelo mecanismo de autenticação, então ele será o ponto de controle para acionar essa nova tarefa de salvar a informação do último acesso do usuário.

Com o MemCache, o número de acessos ao Google Cloud Datastore para realizar essa tarefa ficará bem menor do que sem ele. Do contrário, a informação seria persistida no Google Cloud Datastore toda vez que um usuário fizesse uma requisição.

A lógica proposta para utilizar o MemCache para registrar o último acesso do usuário tem os seguintes requisitos principais:

- Salvar no MemCache o endereço de e-mail do usuário autenticado, juntamente com a data/hora que isso ocorreu;
- Durante a autenticação de uma requisição, caso essa informação ainda não exista no MemCache, ela deverá ser salva nele e no Google Cloud Datastore também;
- Se a informação já existir no MemCache e a data/hora da última atualização for menor do que 30 segundos em relação à nova requisição, nem o MemCache nem o Datastore deverão ser atualizados;
- Se a informação do MemCache for mais antiga do que 30 segundos, então tanto ele quanto o Datastore deverão ter os valores do último login atualizados para o usuário em questão.

Como você pode perceber, o número de acessos ao Google Cloud Datastore para a atualização da informação de data/hora do último login de cada usuário fica reduzido, evitando operações de escrita desnecessárias. O tempo de 30 segundos escolhido para esse exemplo pode até ser maior, em uma aplicação real. Tudo dependerá da precisão desejada para a informação de último login do usuário, reduzindo ainda mais os acessos ao Google Cloud Datastore para a gravação de dados.

Para implementar os requisitos apresentados, comece criando o método `updateUserLogin` no repositório de usuários, na classe `UserRepository` :

```
public void updateUserLogin(User user) {  
}
```

Esse método será chamado dentro do mecanismo de autenticação, um pouco mais adiante. Veja como deve ficar a implementação desse método, considerando todos os requisitos apresentados no início desta seção:

```
public void updateUserLogin(User user) {  
    boolean canUseCache = true;  
    boolean saveOnCache = true;  
  
    Cache cache;  
    try {  
        CacheFactory cacheFactory = CacheManager.getInstance().getCacheFactory();  
        cache = cacheFactory.createCache(Collections.emptyMap());  
  
        if (cache.containsKey(user.getEmail())) {  
            Date lastLogin = (Date)cache.get(user.getEmail());  
            if ((Calendar.getInstance().getTime().getTime() - lastLogin.getTime()) < 30000) {  
                saveOnCache = false;  
            }  
        }  
  
        if (saveOnCache) {  
            cache.put(user.getEmail(), (Date)Calendar.getInstance().getTime());  
            canUseCache = false;  
        }  
    } catch (CacheException e) {  
        canUseCache = false;  
    }  
  
    if (!canUseCache) {  
        user.setLastLogin((Date) Calendar.getInstance().getTime());  
    }  
    try {  
        this.updateUser(user, user.getEmail());  
    } catch (UserAlreadyExistsException | UserNotFoundException e) {  
        log.severe("Falha ao atualizar último login do usuário");  
    }  
}
```

```
}
```

Repare que, depois de a instância do MemCache ser requisitada, o código verifica a existência de uma chave com o valor do e-mail do usuário autenticado, pelo método `cache.containsKey(email)`. Caso exista algum valor, a data lá armazenada é comparada com a atual do sistema. Se o intervalo for maior do que 30 segundos, a informação é atualizada no MemCache e, posteriormente, no Google Cloud Datastore.

Perceba que o código que atualiza o Datastore só é executado caso a informação não esteja no MemCache ainda, ou se já se passaram 30 segundos desde a última atualização do MemCache para esse usuário. Isso reduz os acessos de escrita ao Datastore para cada usuário.

IMPORTS DE BIBLIOTECAS

Se tiver dúvidas sobre quais imports utilizar, consulte o código-fonte do projeto no repositório <https://github.com/siecola/GAEBookV3Exemplo1>.

Finalmente, para que o mecanismo todo seja disparado, vá até a classe `UserServiceImpl`, dentro do método `loadUserByUsername`, e adicione a seguinte instrução logo após o teste:
`userRepository.updateUserLogin(optUser.get());` para verificar se o usuário foi localizado. Veja com o método deverá ficar:

```
@Override  
public UserDetails loadUserByUsername(String email)
```

```
throws UsernameNotFoundException {
    Optional<br.com.siecola.gae_exemplo1.model.User> optUser =
        userRepository.getByEmail(email);
    if (optUser.isPresent()) {
        userRepository.updateUserLogin(optUser.get());
        return optUser.get();
    } else {
        throw new UsernameNotFoundException("Usuário não encontrado")
    }
}
```

Como esse método é chamado sempre que uma requisição precisa ser autenticada, ele é o ponto ideal para disparar o mecanismo de persistência do último login do usuário.

Claramente, esse método não conhece a implementação do MemCache, que ficou encapsulada dentro do método `updateUserLogin` do repositório de dados. Ele é o responsável por verificar se a informação de último login deve ou não ser persistida no Google Cloud Datastore, dependendo de quando foi a última vez que essa operação foi feita.

O mecanismo de autenticação simplesmente aciona tal tarefa e deixa por conta do repositório de dados decidir o que fazer. Depois de realizar todas as alterações, publique seu projeto no GAE, para observar os detalhes do MemCache, explicados na próxima seção.

12.4 VISUALIZANDO O MEMCACHE DO CONSOLE DO GAE

O console do GAE possui uma seção específica para exibir as informações que estão armazenadas no MemCache. Para acessá-las, basta ir ao menu principal da aplicação, na seção App Engine , e escolher a opção MemCache :

Memcache				
	NEW KEY	EDIT	DELETE	FLUSH CACHE
Memcache service level Shared Best effort. Change	Hit ratio — 0 hit / 0 miss	Items in cache 1	Oldest item age 1 min 6 sec	Total cache size 158 B

Figura 12.1: Estatísticas do MemCache

Para que você veja o mecanismo de persistência de último login do usuário funcionando com o MemCache, acesse qualquer operação da sua aplicação utilizando o Postman. Um exemplo seria acessar a de listar todos os usuários. Isso fará com que uma chave seja criada no MemCache, correspondente ao usuário que realizar a requisição.

Depois de fazer a requisição, a tabela exibida na figura anterior será atualizada, indicando que mais informações foram adicionadas ao MemCache. Você também pode localizar qualquer informação no MemCache, clicando no botão `Find a key` e configurando os campos:

Show all keys

Namespace

Key type

Java String

Key

admin@siecola.com

Find Cancel

Search results by key

	Rank	% of traffic in shard	Namespace	Key
				admin@siecola.com

Figura 12.2: Dados do MemCache

No campo **Key**, digite o e-mail do usuário que fez a requisição, para provar que o mecanismo de persistência de último login funcionou corretamente em conjunto com o MemCache. Se você encontrou um dado no MemCache atrelado ao usuário de e-mail que pesquisou, tudo está funcionando com o esperado!

MEMCACHE NO AMBIENTE DE DESENVOLVIMENTO

O MemCache também funciona no ambiente de desenvolvimento, mas o console de administração local não possui uma interface para exibir os dados contidos nele. Isso só está disponível na interface do console do GAE do ambiente de produção.

Conclusão

Neste capítulo, você melhorou o desempenho do mecanismo de autenticação de usuários, utilizando o serviço MemCache do GAE – mais uma funcionalidade muito interessante e útil dessa plataforma. No próximo capítulo, você aprenderá a implementar um mecanismo de autenticação com o OAuth 2.0, mais seguro e eficiente do que o HTTP Basic (utilizado até o momento).

CAPÍTULO 13

PROTEGENDO SERVIÇOS COM OAUTH 2.0

A segurança em serviços REST deve ser muito bem tratada, pois como dito no início do capítulo *Protegendo serviços com HTTP Basic Authentication*, vários aspectos devem ser considerados na escolha do melhor mecanismo a ser implementado no servidor que proverá os serviços.

Um tipo de autenticação que vem se popularizando muito é o OAuth 2.0, principalmente por alguns pontos importantes:

- Facilidade de implementação, tanto pelo provedor quanto pelo cliente;
- Pouco consumo de recursos do lado do servidor;
- Um menor *overhead* de requisições para que o cliente consiga acesso autenticado aos recursos.

Esse último é muito importante, principalmente para dispositivos móveis, em que precisamos balancear o nível de segurança do serviço em questão e a quantidade de requisições que a aplicação faz, já que muitas vezes o acesso à internet tem custo.

Este capítulo apresenta o mecanismo de autenticação **OAuth 2.0 com Bearer Token**, para proteger os serviços hospedados no

GAE. Você verá que muito do que já foi aprendido no capítulo *Protegendo serviços com HTTP Basic Authentication* será reutilizado, como: as anotações nos métodos das classes dos serviços, e como fazer para descobrir qual é o usuário autenticado dentro desses métodos.

13.1 O QUE É OAUTH 2.0

Proteger serviços REST com o mecanismo OAuth 2.0 é uma escolha sensata para a maioria das aplicações, visto que traz um bom equilíbrio entre o nível de segurança oferecido por esse método, o consumo de recursos do servidor e os números de requisições do cliente.

Basicamente, o OAuth 2.0 com Bearer Token funciona com os seguintes elementos:

- **Token:** é uma chave de acesso que o cliente deve obter junto ao provedor de autenticação para ser utilizada em todas as requisições aos serviços.
- **Provedor de autenticação:** é o responsável por fornecer os tokens para os clientes que solicitarem. Também cuida da validade de cada token, para que eles existam somente durante um tempo determinado pelo provedor.
- **Mecanismo de autenticação:** em cada requisição do cliente, verifica se há um token válido e localiza o usuário que o solicitou no provedor de autenticação, para que a requisição possa prosseguir com um usuário autenticado associado a ela.

O provedor de autenticação só entrega o token a um cliente se

ele fornecer as suas credenciais de acesso corretas, cadastradas em um banco de dados ou em outro local apropriado.

Perceba que, após o cliente solicitar o token ao provedor de autenticação, ele não fornece mais as credenciais de acesso nas demais requisições aos serviços que deseja acessar. Ele utiliza somente o token para isso, até que ele expire e tenha de solicitar outro. Dessa forma, se alguém interceptar a comunicação e obter o token do cliente, ele não terá acesso às credenciais de acesso de seu usuário.

13.2 ALTERANDO A APLICAÇÃO GAE_EXEMPLO1

Para demonstrar como funciona o mecanismo de autenticação de usuários usando OAuth 2.0 com Bearer Token – sem a necessidade de criar uma outra aplicação somente para isso –, usaremos o projeto `gae_exemplo1`. Apesar de ele já possuir um mecanismo baseado em HTTP Basic Authentication, será muito simples trocá-lo pelo OAuth 2.0.

Desabilitando o HTTP Basic Authentication

Como a aplicação `gae_exemplo1` já possui um mecanismo de autenticação ativo, não é possível adicionar um outro sem antes desabilitar o que está em uso. Felizmente, isso é muito simples de se fazer em uma aplicação Spring Boot, pois ele possui uma funcionalidade baseada em anotações, que permite configurar perfis da aplicação.

Para começar, abra o arquivo `application.properties` e

adicone a linha a seguir:

```
spring.profiles.active=auth_oauth2
```

Você poderia colocar qualquer nome para definir o perfil que deseja ativar. O escolhido aqui é apenas para deixar mais claro que a aplicação está em execução utilizando o mecanismo OAuth 2.0.

Agora vá à classe que configura a autenticação HTTP Basic, a `SpringSecurityConfigHttpBasic`, e adicione a seguinte anotação no topo de sua declaração (logo acima da anotação `@Configuration`):

```
@Profile("auth_basic")
```

Isso significa que essa classe só entrará em execução quando o perfil `auth_basic` estiver selecionado. Como você ativou o perfil `auth_oauth2` no arquivo `application.properties`, essa classe não terá nenhum efeito na aplicação, ou seja, o mecanismo de autenticação HTTP Basic não mais está ativo.

Adicionando as dependências ao projeto

Agora que o antigo mecanismo de autenticação foi desativado, é necessário adicionar mais uma biblioteca à aplicação. Assim, poderemos utilizar as bibliotecas do Spring Security que implementam a autenticação OAuth 2.0.

Para isso, abra o arquivo `pom.xml` da aplicação e adicione mais uma dependência a ele:

```
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
    <version>2.2.6.RELEASE</version>
</dependency>
```

O módulo Spring Security OAuth2, adicionado nessa dependência, possui tudo o que é necessário para que uma aplicação possa ter o mecanismo OAuth2, facilitando muito o trabalho do programador. Na verdade, você só terá de criar 2 classes de configuração, de forma semelhante à qual fez com o HTTP Basic.

Criando as classes de configuração

A primeira classe de configuração a ser criada é para definir que o mecanismo de autenticação deverá interpretar as anotações presentes nos métodos dos serviços REST, como `@PreAuthorize`, presente nos `controllers` `UserController` e `ProductController`.

Para isso, crie a classe `MethodSecurityConfig` dentro pacote `config` da aplicação:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.security.access.expression.method.MethodSecurityExpressionHandler;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodSecurity;
import org.springframework.security.config.annotation.method.configuration.GlobalMethodSecurityConfiguration;
import org.springframework.security.oauth2.provider.expression.OAuth2MethodSecurityExpressionHandler;

@Profile("auth_oauth2")
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class MethodSecurityConfig extends GlobalMethodSecurityConfiguration {
    @Override
```

```
protected MethodSecurityExpressionHandler createExpressionHandler() {
    return new OAuth2MethodSecurityExpressionHandler();
}

@Bean
public AuthenticationManager authenticationManager() throws E
xception {
    return super.authenticationManager();
}
}
```

Com essa configuração, será possível manter tudo o que já foi construído em termos de autorização de requisições – por exemplo, permitir que somente um administrador possa criar outros usuários.

Repare que foi adicionada a anotação `@Profile` com o perfil `auth_oauth2` – o qual está ativo atualmente na aplicação –, por meio da configuração realizada no arquivo `application.properties`. É interessante adotar essa prática, pois assim será possível alterar entre os dois tipos de autenticação com maior facilidade.

A segunda classe de configuração a ser criada é muito semelhante à primeira. Ela indicará onde o mecanismo de autenticação deverá buscar a base de dados de usuário, bem como a estratégia de proteção das requisições feitas à aplicação.

Para isso, crie uma outra classe chamada `SpringSecurityOAuth2` no mesmo pacote `config`:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.security.authentication.Authentication
```

```

Manager;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.oauth2.config.annotation.configurers.ClientDetailsServiceConfigurer;
import org.springframework.security.oauth2.config.annotation.web.configuration.AuthorizationServerConfigurerAdapter;
import org.springframework.security.oauth2.config.annotation.web.configuration.EnableAuthorizationServer;
import org.springframework.security.oauth2.config.annotation.web.configurers.AuthorizationServerEndpointsConfigurer;

@Profile("auth_oauth2")
@Configuration
@EnableAuthorizationServer
public class SpringSecurityOAuth2 extends AuthorizationServerConfigurerAdapter {

    @Autowired
    @Qualifier("userDetailsService")
    private UserDetailsService userDetailsService;

    @Autowired
    private PasswordEncoder passwordEncoder;

    @Autowired
    private AuthenticationManager authenticationManager;

}

```

Veja que o serviço `UserDetailsService` – o mesmo utilizado pelo mecanismo HTTP Basic – foi injetado nessa classe. Ele será usado integralmente pelo mecanismo de autenticação OAuth 2.0 que está sendo construído.

O primeiro método de configuração criado nessa classe será o qual diz onde os usuários deverão ser buscados quando uma requisição precisar ser autenticada:

```
@Override
public void configure(AuthorizationServerEndpointsConfigurer configurer)
        throws Exception {
    configurer.authenticationManager(authenticationManager);
    configurer.userDetailsService(userDetailsService);
}
```

Veja que, na linha 5, o serviço `UserDetailsService` foi informado na configuração para ser aquele que deverá ser chamado quando uma requisição for autenticada e o mecanismo precisar buscar as informações do usuário.

Da mesma forma como aconteceu no HTTP Basic, o Spring Security vai chamar o método `loadUserByUsername` da classe `UserDetailsService`, para buscar as informações do usuário que está tentando realizar uma requisição. Com isso, podemos aproveitar tudo o que foi feito até o momento em relação à junção do Spring Security com a base de dados do usuário no Google Cloud Datastore.

O segundo método de configuração dessa classe define algumas informações necessárias do mecanismo OAuth 2.0, dentre elas:

- **Credenciais da aplicação cliente:** o OAuth 2.0 pode distinguir as aplicações através de credenciais atribuídas a elas. Para tornar o exemplo mais simples, essas credenciais serão definidas de forma fixa no código.
- **Escopo de acesso do usuário:** é possível definir qual é o seu escopo de acesso (leitura ou escrita) para cada cliente e para o recurso que este deseja acessar.
- **Tempo de validade do token:** define por quanto tempo um token adquirido pelo usuário ficará ativo no sistema, para

que possa ser usado novamente em outras requisições.

Veja como deve ficar esse segundo método de configuração:

```
@Override  
public void configure(ClientDetailsServiceConfigurer clients)  
    throws Exception {  
  
    clients.inMemory()  
        .withClient("siecola").secret(passwordEncoder.encode(  
"matilde"))  
        .accessTokenValiditySeconds(3600)  
        .scopes("read", "write")  
        .authorizedGrantTypes("password")  
        .resourceIds("oauth2-resource");  
}
```

Repare que o tempo de validade definido para o token ficou em 3600 segundo. Além disso, as credenciais de acesso do cliente (não do usuário) ficaram definidas como:

- Client: siecola ;
- Secret: matilde .

Essas duas informações deverão estar presentes em todo pedido de token, bem como as credenciais de acesso do usuário. Mais adiante, será mostrado o processo para obtenção do token para usarmos nas requisições.

Por fim, ainda nessa classe, adicione o método para criar o bean responsável por encriptar a senha do usuário:

```
@Bean  
public BCryptPasswordEncoder passwordEncoder() {  
    return new BCryptPasswordEncoder();  
}
```

A última configuração a ser feita é a adição de uma anotação na

```
classe GaeExemplo1Application :
```

```
@EnableResourceServer
```

Essa anotação diz ao Spring Boot para habilitar o servidor para gerenciar os tokens que forem sendo adquiridos pelos usuários para realizarem as autenticações. Esse servidor cuida da expiração de cada token, bem como a sua validação durante o processo de autenticação.

Assim, tudo já está pronto e a sua aplicação já possui o mecanismo OAuth 2.0 integrado com a base de dados de usuários armazenada no Google Cloud Datastore!

13.3 TESTANDO O MECANISMO DE AUTENTICAÇÃO OAUTH 2.0

Acessar serviços REST com autenticação OAuth 2.0 e com o Postman é bem simples, pois este possui ferramentas para conseguirmos adquirir e manipular tokens de autenticação. Nesse exemplo, será mostrado como proceder para acessar a operação para listar todos os usuários, porém esse mesmo processo também se aplica aos demais serviços da aplicação.

Dentro do Postman, clique na aba `Authorization` para abrir as configurações relacionadas a isso. Nela, escolha o tipo `OAuth 2.0` e também a opção `Request Headers` para o campo `Add authorization data to`.

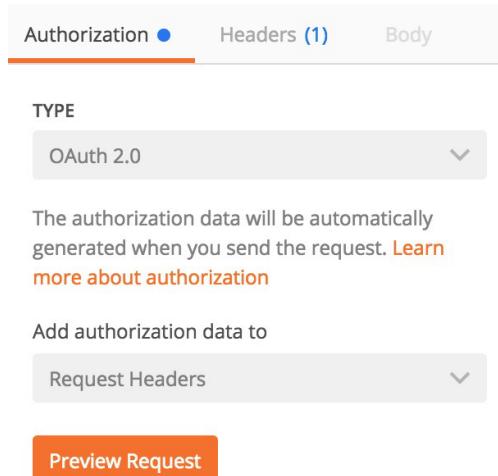


Figura 13.1: Configurando o OAuth2 no Postman

Isso faz com que o Postman prepare-se para trabalhar com o mecanismo de autenticação OAuth 2.0, enviando o token no cabeçalho de cada requisição.

Nessa mesma tela, clique no botão Get New Access Token para solicitar um novo token para a aplicação. Será aberta uma tela para você preencher alguns dados, detalhados a seguir:

GET NEW ACCESS TOKEN X

Token Name	Matilde
Grant Type	Password Credentials
Access Token URL ?	http://localhost:8080/oauth/token
Username	matilde@siecola.com.br
Password	matilde
<input checked="" type="checkbox"/> Show Password	
Client ID ?	siecola
Client Secret ?	matilde
Scope ?	e.g. read:org
Client Authentication	Send as Basic Auth header
Request Token	

Figura 13.2: Solicitando o token de acesso

Essa tela solicita os seguintes parâmetros:

- **Token Name:** um nome que você pode definir como quiser, com intuito de melhor definir o token que está requisitando. Pode ser o nome do usuário dono das credenciais de acesso, por exemplo.
- **Grant Type:** o tipo das credenciais a serem fornecidas para solicitar o token. Na aplicação que está sendo construída, ele deve ser do tipo **Password Credenciais**.
- **Access Token URL:** quando o mecanismo OAuth 2.0 foi adicionado à aplicação, uma nova URL foi implicitamente

criada para que os clientes possam solicitar o token de acesso. Essa URL é /oauth/token e sempre deverá ser usada para a obtenção do token de acesso para a autenticação.

- **Username:** e-mail do usuário solicitante do token.
- **Password:** senha do usuário solicitante do token.
- **Client ID:** definido de forma fixa na classe SpringSecurityOAuth2 , ele caracteriza o cliente que está solicitando o token. Não confunda com o e-mail do usuário.
- **Client Secret:** definido de forma fixa na classe SpringSecurityOAuth2 , ele representa a senha do cliente que está solicitando o token. Não confunda com a senha do usuário.
- **Scope:** use-o caso queira solicitar um token com escopo restrito ao recurso que deseja acessar.
- **Client Authentication:** esta é a forma como o Postman deverá enviar as credenciais de acesso do cliente. Nesse caso, deve ser enviado no cabeçalho da requisição, por isso, escolha Send as Basic Auth reader .

Com todas essas configurações realizadas, clique no botão Request Token para solicitar o token de acesso. Deverá aparecer uma tela como mostra a figura a seguir:

MANAGE ACCESS TOKENS		
ALL TOKENS	Token Name	Matilde
Matilde	Access Token	ddb79d52-98bc-47af-877f-4aacbf8ee1d7
	token_type	bearer
	refresh_token	818b0999-40fe-46dd-bbd8-ad47bd3f5c56
	expires_in	3599
	scope	read write

[Use Token](#)

Figura 13.3: Gerenciando tokens

Essa é a tela de gerenciamento de tokens, que também pode ser acessada no menu `Available Tokens`. Ela exibe todos os tokens disponíveis no Postman para essa aplicação.

Veja que todas as informações (como o tipo do token e o tempo que ele vai expirar) aparecem aqui. Sempre que você reiniciar a aplicação, ou quando um token expirar, você deve excluí-lo e solicitar um novo, para ter acesso à aplicação.

Para instruir o Postman a utilizar o token que você deseja, acesse o *combo box* `Available Tokens` e escolha o token do usuário que desejar. Isso fará que com o Postman insira-o no cabeçalho da requisição, como pode ser visto na figura a seguir:

Authorization	Headers (2)	Body	Pre-request Script	Tests
Key	Value			
Authorization	Bearer ddb79d52-98bc-47af-877f-4aacbf8ee1d7			
Content-Type	application/json			

Figura 13.4: Visualizando o token no cabeçalho

Veja que o nome do cabeçalho é igual ao que foi usado no mecanismo HTTP Basic, ou seja, `Authorization`. O mecanismo de autenticação OAuth 2.0 do Spring Security vai procurar por esse cabeçalho para validar a requisição. A diferença aqui está no valor desse cabeçalho, que agora possui um tipo diferente, chamado de `Bearer`, seguido pelo token de acesso obtido nos passos anteriores desta seção.

Com tudo isso feito, todas as requisições conterão esse token para autenticar a requisição. Vale lembrar que ele sempre possui um tempo de validade e, caso expire, será necessário solicitar um novo.

Conclusão

Agora você pode construir serviços REST, hospedados no Google App Engine, de forma mais segura com o mecanismo de autenticação OAuth 2.0 e com Bearer token – um mecanismo muito usado e conhecido.

ALGO MAIS SOBRE GOOGLE APP ENGINE

O Google App Engine é uma plataforma fantástica que sempre está inovando e lançando novas funcionalidades. Há algumas outras coisas pelas quais você pode se interessar, ou talvez precise, para um outro projeto. Veja a seguir alguns temas e uma referência de consulta na documentação do Google.

Envio e recebimento de e-mails

É possível enviar e receber e-mails de uma aplicação do Google App Engine. Para mais informações, consulte:

<https://cloud.google.com/appengine/docs/java/mail/>.

Google Cloud SQL

O Google oferece uma infraestrutura de hospedagem de banco de dados, semelhante ao MySQL. Com ela, você pode desenvolver aplicações que precisam de uma base de dados relacional, ou que um dia poderão ser portadas para outra plataforma. Para mais informações, consulte:

<https://cloud.google.com/appengine/docs/java/cloud-sql/>.

Google Cloud Endpoints

Este é um framework para desenvolvimento de APIs na parte do back-end de uma aplicação do GAE, para ser utilizado por clientes móveis (como Android e iOS). O conceito é interessante e agiliza muito o desenvolvimento de ambas as partes.

Entretanto, é necessário avaliar com cuidado sua utilização se você deseja desenvolver uma aplicação que pode ser usada por uma gama maior de clientes desenvolvidos, em outras tecnologias e linguagens. Para mais informações, consulte:

<https://cloud.google.com/appengine/docs/java/endpoints/>.

JPA e JDO

É possível usar JPA ou JDO para acessar dados no Google Cloud Datastore, abstraindo a API de baixo nível nativa do SDK.

- Para informações sobre como utilizar JDO, consulte <https://cloud.google.com/appengine/docs/java/datastore/jdo/overview>.
- Para informações sobre como utilizar JPA, consulte <https://cloud.google.com/appengine/docs/java/datastore/jpa/overview-dn2>.

Conclusão

Lembre-se de que você pode encontrar tudo o que foi desenvolvido aqui no repositório de código do livro, em:
<https://github.com/siecola/GAEBookV3Exemplo1>