

Spring Framework

Curso FJ-27



 caelum
ensino e inovação

Apostila gerada especialmente para Willian Silva Moreira - moreiraws85@gmail.com



Conheça também:



alura

alura.com.br



Casa do Código
Livros e Tecnologia

casadocodigo.com.br

Blog da Caelum



blog.caelum.com.br

Newsletter



caelum.com.br/newsletter

Facebook



facebook.com.br/caelumbr

Twitter



twitter.com/caelum

Sumário

1 Sobre o curso	1
1.1 Os exercícios	1
1.2 Os super desafios	1
1.3 O projeto	2
1.4 Tirando dúvidas	3
2 Introdução	4
2.1 Por que o Spring Framework	4
2.2 Quase zero de configuração	6
2.3 À margem da especificação	6
2.4 Comece a aventura	7
2.5 Público-alvo	8
2.6 Código fonte	8
3 Começando o projeto	9
3.1 Spring Boot	9
3.2 Dependências do projeto	10
3.3 Configurando o Data Source	13
3.4 Primeiro Controller	15
3.5 Exercícios: Rodando o projeto com Spring Boot	16
4 Dando início à nossa API	20
4.1 Web Services	20
4.2 Protocolos	22
4.3 Recursos	23
4.4 Modelo	24
4.5 Controller de tópicos	25
4.6 CORS	30
4.7 DevTools	33
4.8 Exercícios: Escrevendo nossa primeira lógica	34

5 Acessando dados com Spring Data JPA	39
5.1 Eliminando o código repetitivo	39
5.2 Limitando os comportamentos da interface	41
5.3 Definindo nossas próprias queries	42
5.4 Para saber mais: queries a partir do nome dos métodos	42
5.5 Exercícios: Exibindo lista de dúvidas	44
6 Mais além com Spring Data	47
6.1 Filtrando as dúvidas	47
6.2 Exercícios: Trabalhando com filtros opcionais de busca	51
6.3 Paginando os resultados	53
6.4 Ordenando os resultados	56
6.5 Exercícios: Paginando resultados com Spring Data	58
7 Documentando e interagindo com nossos endpoints	61
7.1 Swagger	61
7.2 Customização	66
7.3 Exercícios: Habilitando o Springfox Swagger 2	68
7.4 SUPER DESAFIO: Implemente a apresentação do dashboard de dúvidas do fórum	70
8 Protegendo a API com Spring Security	73
8.1 Autenticação com Spring Security	73
8.2 Iniciando com Spring Security	74
8.3 Definindo nossos usuários	76
8.4 Encriptando as senhas	76
8.5 Exercício - Primeiro login com Spring Security	78
8.6 Login com dados do banco	81
8.7 Exercícios: Autenticação com Spring Security	86
8.8 Autorização com Spring Security	90
8.9 Exercício: Autorização com Spring Security	92
8.10 Utilizando JWT com Spring Security	93
8.11 JWT	97
8.12 Gerando um token de acesso com jjwt	99
8.13 Exercícios: Utilizando JWT com Spring Security	108
8.14 Validando e autenticando o token	115
8.15 Cadeia de Filtros do Security	121
8.16 Enviando um token pela requisição	123
8.17 Exercícios - Validando e autenticando o token	125
9 Inserindo tópicos e validando dados de entrada	131

9.1 Inserindo um tópico	131
9.2 Construindo uma resposta mais expressiva	134
9.3 Exercícios: Inserindo novos tópicos	136
9.4 Validando um tópico	141
9.5 Melhorando o código	146
9.6 Exercícios: Validando um tópico	147
9.7 Validações de negócio	151
9.8 Refatorando o código	153
9.9 Mensagens customizadas	155
9.10 PARA SABER MAIS: Mensagens internacionalizadas	158
9.11 Exercícios: Aplicando validações de negócio e personalizando mensagens	160
10 Respondendo dúvidas e enviando email	166
10.1 SUPER DESAFIO: Implemente as respostas em dúvidas abertas	166
10.2 Enviando emails com Spring e Java Mail	167
10.3 Configurando servidor SMTP do Google	168
10.4 JavaMail	170
10.5 Enviando email com Spring Mail	171
10.6 Processamento assíncrono	174
10.7 Exercício: Enviando e-mails com Spring e JavaMail	176
10.8 Utilizando templates de email e trabalhando com múltiplos profiles	179
10.9 Thymeleaf: Breve introdução	180
10.10 Template Engine	182
10.11 Enviando o template do email	184
10.12 Trabalhando com múltiplos profiles	186
10.13 Exercícios: Utilizando templates de email e trabalhando com multiplos profiles	189
11 Agendando tarefas e renderizando views com Thymeleaf	196
11.1 Projections	197
11.2 Agendando tarefas com Java	198
11.3 Agendando tarefas com Spring	199
11.4 Exercício: Agendando tarefas com Spring	201
11.5 Renderizando relatório de tópicos abertos com Thymeleaf	204
11.6 Exercício: Renderizando relatório de tópicos abertos com Thymeleaf	207
11.7 Múltiplos perfis de segurança	209
11.8 Exercício: Trabalhando com múltiplos perfis de segurança	212
12 Testando a API com Spring Boot Test	215
12.1 Testando nossa aplicação	215

Sumário	Caelum
12.2 Testando o Repository	217
12.3 Testando nossas queries	220
12.4 Exercício: Testando a integração com a infraestrutura de persistência	225
12.5 Testando nossos controllers	228
12.6 Exercício: Testando a camada web da aplicação	234
13 Monitorando a API com Spring Boot	238
13.1 Spring Actuator	238
13.2 Outros endpoints	242
13.3 Criando um novo endpoint	244
13.4 Para saber mais: Spring Boot Admin	246
13.5 Exercícios: Habilitando o Actuator e criando novo endpoint	248
14 Apêndice: Melhorando a performance com cache	254
14.1 Começando com Spring Cache	254
14.2 CacheEvict	257
14.3 Exercício: Começando com Spring Cache	260
14.4 Redis	262
14.5 Configurando o Redis no projeto	263
15 Apêndice: Mais segurança com Spring Security ACL	269
15.1 Exercício: Marcando respostas como solução	269
15.2 Exercício: Fechando tópicos e aumentando a segurança sobre as informações	272
15.3 Para saber mais: Spring Security ACL SQL Schema	277

Versão: 23.3.24

SOBRE O CURSO

Desenvolver software em Java é trabalho realizado por milhões de desenvolvedores mundo afora e, como em toda tecnologia com alguns bons anos de mercado, uma série de soluções e ferramentas surgiram com o tempo para facilitar o desenvolvimento de aplicações. Neste curso você vai aprender conceitos avançados sobre a plataforma mais amplamente utilizada no mundo Java, o Spring Framework. Ao final do treinamento você será capaz de desenvolver uma REST API de forma muito eficiente, usando os módulos mais recentes dessa stack.

1.1 OS EXERCÍCIOS

Os exercícios são essenciais para a melhor compreensão dos tópicos apresentados, pois neles vamos aplicar as técnicas recomendadas para trabalhar com o framework e vamos analisar e solucionar, de maneira prática, os problemas que enfrentamos no dia a dia como desenvolvedores. Além dos exercícios que fazem parte da didática proposta, apontaremos no decorrer do curso alguns exercícios opcionais. Sugerimos que os mesmos sejam feitos pois neles iremos explorar algumas alternativas de solução de problemas para conhecermos melhor algumas características do Spring.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

1.2 OS SUPER DESAFIOS

Além dos exercícios também apontaremos no decorrer do curso alguns desafios para que você

coloque em prática a sua capacidade analítica e de solução dos problemas. Alguns desafios irão requerer alguma pesquisa e leitura de documentação, então fique atento às referências apresentadas na apostila e pelo instrutor durante o curso.

Em geral, os desafios serão propostos em torno do desenvolvimento de novas funcionalidades, as quais você já estará apto a criar nos pontos onde eles serão apresentados. O instrutor pode ser utilizado como seu consultor nesses desafios, mas é fundamental a aplicação de forma individualizada dos conceitos aprendidos até então na busca por uma solução. O instrutor é responsável por definir e controlar o tempo de execução do desafio.

Ao fim de cada super desafio, teremos pronto na pasta de arquivos do curso um projeto com toda a base de código construída nos exercícios anteriores junto com uma das possíveis soluções para o desafio. Prosseguiremos o curso tendo como base estes projetos. Essa estratégia é adotada pois precisaremos contar com uma implementação base que sustente os próximos exercícios passo a passo.

1.3 O PROJETO

Ao longo do curso iremos desenvolver uma aplicação baseada no fórum de discussões da plataforma online da Alura, a Alura Fórum. Criaremos uma REST API que será utilizada durante os exercícios por uma aplicação front-end já desenvolvida com React. O motivo desta escolha foram os problemas que a implementação do software apresentará, como a gestão do trabalho no acesso a dados, a implementação da camada de segurança da nossa API, e várias outras práticas que serão discutidas no decorrer do desenvolvimento.

The screenshot shows a web browser window with the URL <https://cursos.alura.com.br/forum>. The page has a header with the Alura logo, a search bar, and navigation links for CURSOS, COMUNIDADE, and a user profile for 'FULANO'. Below the header, there's a main navigation bar with the word 'FÓRUM' in bold. The page displays three forum categories in cards:

- Mobile**: Represented by an orange icon of a smartphone. It lists topics like Android, Multiplataforma, Jogos, and iOS. Statistics: 4353 topics, 40 posts in the last week, 9 unanswered posts.
- Programação**: Represented by a green icon of a computer monitor. It lists topics like Java, Computação, Integrações em Java, Lógica de programação, NET para web, Análise de dados, Linguagem C, C#, Boas práticas em C#, Boas práticas em Java, Programação Funcional, Certificação Java, Java para Web, Java e persistência, Testes em Java, Testes em C#, Ruby, PHP, Python, Ruby on Rails, Interações em .NET, Persistência com .NET, and Desenvolvimento de jogos. Statistics: 24062 topics, 199 posts in the last week, 17 unanswered posts.
- Front-end**: Represented by a blue icon of a person with a laptop. It lists topics like HTML e CSS, Frameworks MVC, JavaScript, Automação e Performance, and jQuery. Statistics: 9955 topics, 94 posts in the last week, 3 unanswered posts.

Figura 1.1: Alura Fórum, aplicação desenvolvida durante o curso com funcionalidades baseadas no Fórum da Alura.

1.4 TIRANDO DÚVIDAS

Durante o curso, tenha a certeza de tirar todas as suas dúvidas com o instrutor. Algumas características de como trabalhar com Spring podem parecer básicas a princípio, mas conhecê-las a fundo é essencial para seu desenvolvimento e compreensão de alguns tópicos mais avançados. Não tenha medo de fazer perguntas! Por mais básicas que elas possam parecer, são esses os pontos mais importantes para a melhor compreensão dos assuntos abordados.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

INTRODUÇÃO

Já existem diversos frameworks no mercado e a primeira pergunta que você deve fazer é: existe a necessidade de aprender mais um? Em geral, existem algumas tecnologias que, quando são escolhidas para serem usadas em uma aplicação, não são facilmente reversíveis. O framework web, o framework de segurança ou o de acesso a dados são algumas delas.

Elas vão fazer parte de uma boa parte da sua rotina de desenvolvimento. Quase sempre que uma nova funcionalidade for pedida para o seu projeto, além de implementar a regra de negócio específica, você terá que escrever alguma coisa relativa a essa regra na camada web, ou de acesso a dados da sua aplicação. E aí, você mais uma vez estará interagindo com os frameworks escolhidos.

Pensando nesse prisma, você deve ser capaz de analisar os pontos positivos e negativos de cada uma das suas opções e tomar a melhor decisão possível.

2.1 POR QUE O SPRING FRAMEWORK

Com a crescente oferta de bibliotecas, frameworks e ferramentas no mundo Java, algumas brigando pelo mesmo espaço, surgiu a necessidade de padronizar a forma como elas se organizavam para atingir seus objetivos na resolução de problemas conhecidos no desenvolvimento de software. Com isso, foi criada a Java Enterprise Edition (Java EE), que é um conjunto de especificações que padronizam o trabalho e definem o escopo de uma plataforma de desenvolvimento para aplicações de grande porte.

Por algum tempo desenvolver sistemas utilizando como base algum software que implementava cada uma das especificações da Java EE (um servidor de aplicação Java EE) era sinônimo de segurança e confiabilidade. Mas, com o passar do tempo, ficaram evidentes os problemas gerados pela burocracia com a qual era gerida a plataforma, como a lenta evolução das especificações de software que eram providas e a dificuldade de inovação que isso gerava. O ritmo com o qual as aplicações apresentavam novos desafios com o passar do tempo se mostrou muito maior do que o ritmo com o qual a plataforma era capaz de trazer soluções ao mercado. Era hora de abrir mão do trabalho com a Java EE e utilizar soluções providas como alternativa às plataformas que seguiam a especificação; surgiu assim o Spring.

O Spring é um projeto de longa data. O projeto principal do framework provê um container de Inversão de Controle (*Container IoC*), que desde o início foi amplamente utilizado no mercado, popularizando bastante o padrão Injeção de Dependências (*Dependency Injection - DI*). Com o sucesso do projeto, logo surgiram outros módulos que foram desenvolvidos de forma completamente integrada

ao módulo principal, como o Spring MVC, que trouxe uma implementação do padrão arquitetural MVC que abstrai muito dos detalhes das especificações do Java para a web. Com a facilidade provida pelo framework na escrita de aplicações web, logo ele se tornou o mais utilizado no mercado para este fim. Assim o projeto cresceu de forma totalmente modular, facilitando a integração. Hoje conta com diversos frameworks que podem ser combinados para prover uma stack completa a seu sistema, sendo a principal alternativa ao Java EE. Na verdade, até já superou a especificação há um tempo.

Na hora da adoção de um framework, um dos pontos mais relevantes é o que ele entrega de maneira gratuita, e o Spring Framework oferece seus módulos com uma integração muito facilitada e transparente. Esse é um dos principais motivos para a adoção do Spring. A imagem abaixo mostra apenas as principais características do framework. Para conhecer todas as opções, vale a pena acessar o link <http://spring.io/projects>.

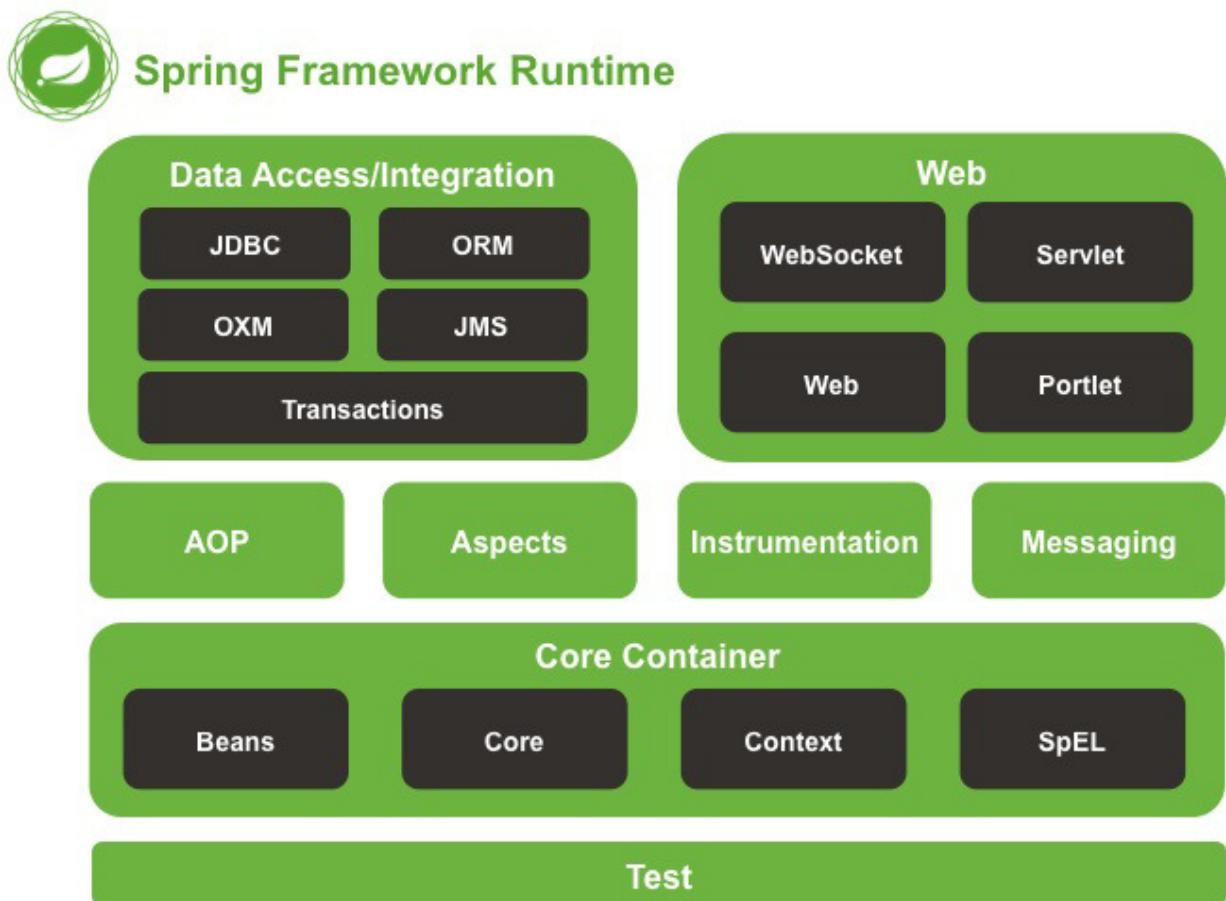


Figura 2.1: módulos do Spring

Um outro ponto altamente relevante é o quanto de ajuda você pode obter através da internet. E, nesse ponto, os projetos relacionados ao Spring também se destacam bastante. Eles têm uma página exclusiva com questões do *StackOverflow*, apenas com as perguntas marcadas com tags relativas ao Spring. Você pode dar um olhada seguindo este link: <http://spring.io/questions>. Além disso, a ZeroTurnaround,

empresa famosa por ferramentas como o JRebel, fez um levantamento dos frameworks mais usados no mercado, no qual a stack do Spring liderou com 46%. Você pode dar uma olhada neste link: <http://bit.ly/rebel-labs-java-report>. A pesquisa foi realizada em 2017 e, considerando a aceitação dos módulos mais recentes do Spring, como o Spring Boot, Spring Data e Spring Cloud, pelo mercado desde então, é bem provável que esse número tenha aumentado.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

2.2 QUASE ZERO DE CONFIGURAÇÃO

O Spring suporta que toda sua configuração seja feita de maneira programática, ou seja, você não precisa criar nenhum XML para que seus módulos comecem a funcionar. Esse é mais um ponto que facilitou ainda mais sua adoção. Em algum momento do tempo, configurações em XML eram a regra para declarar informações pertinentes ao projeto, mas o ponto é: sempre que você alterava uma dessas configurações, era necessário que uma nova instalação do projeto fosse gerada. Dada essa condição, por que escrever código em XML e deixar de escrever código em Java? Com uma boa separação de pacotes, conseguimos deixar nossas configurações bem modulares e mais fáceis de manter.

O que pode ser ainda melhor que escrever um código para a configuração da aplicação? Se você pensou em "*nem precisar escrever código para a configuração*", acertou. Com o Spring Boot como base para o software, toda a configuração necessária para a maioria dos casos de uso de um projeto web já vem pronta para você, bastando na maioria dos casos apenas escrever um arquivo *properties* com as informações inerentes ao seu projeto para que tudo já esteja funcionando.

2.3 À MARGEM DA ESPECIFICAÇÃO

As especificações produzidas pela comunidade Java são de grande valia para as empresas, isso é inegável. Ter uma empresa como a Oracle certificando as decisões que envolvem uma plataforma passa segurança para os tomadores de decisão que querem usar esta linguagem e suas ferramentas no seu projeto.

O ponto negativo, como já vimos, é que muitas vezes as especificações demoram para ser criadas e, para concluirmos alguma tarefa, somos obrigados a recorrer a projetos de terceiros, que muitas vezes não estão integrados com as especificações já existentes. É justamente aí que entra um diferencial dos projetos que usam o Spring como base. Como eles não precisam seguir as especificações, você já tem diversas opções de integrações que ainda não foram especificadas ou cujas especificações ainda estão imaturas. Alguns exemplos:

- Spring Social para integração com redes sociais;
- Quartz para integração com este agendador de tarefas;
- Spring Data para integração com bancos de dados, inclusive não relacionais;
- Spring Security para fornecer autenticação e autorização;
- Spring Cache para ter a possibilidade de cachear diversas partes do código.

Essas são algumas extensões que você pode usar nos seus projetos que tenham o Spring como base. Usaremos algumas delas no decorrer do curso.

2.4 COMECE A AVENTURA

Durante o curso vamos construir uma aplicação espelho do fórum da Alura. Desenvolveremos funcionalidades que nos levarão desde os detalhes mais básicos da configuração e arquitetura do Spring Boot e do uso do Spring MVC, até partes avançadas como:

- Customização de componentes de validação;
- Caches de resultados;
- Processamento assíncrono;
- Agendamento de tarefas com o Spring;
- Facilitação dos testes de integração.

Participe da aula junto com o instrutor, faça os exercícios e leia cada capítulo posteriormente à aula com calma. Esperamos que você tenha uma grande jornada!

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

2.5 PÚBLICO-ALVO

Para pessoas que já tenham sido apresentadas à estrutura do framework e de um projeto baseado no Design Pattern MVC. O curso vai ajudar desde pessoas que conhecem pouco dos módulos do Spring até usuários que já utilizam o framework na prática. Quase todo capítulo vai cobrir detalhes que vão além do uso normal e vai deixá-lo mais crítico em relação ao uso da tecnologia.

2.6 CÓDIGO FONTE

Todo o código fonte do projeto estará disponível no Github. Basta acessar o endereço <https://gitlab.com/aovs/projetos-cursos/fj27-alura-forum-api>. Fique à vontade para navegar pelos arquivos do projeto. Os commits foram divididos de acordo com os capítulos do curso, justamente para que você possa acessar o código compatível ao capítulo que esteja sendo estudado.

COMEÇANDO O PROJETO

Neste curso desenvolveremos a aplicação do Fórum da Alura. Criaremos funcionalidades que nos levarão desde os detalhes mais básicos do uso do Spring MVC até partes avançadas como cache, segurança e agendamento.

3.1 SPRING BOOT

Se você já usou Spring alguma vez, provavelmente utilizou a configuração via arquivo XML ou configuração programática, necessitando diversos passos para integrar diversos módulos do próprio framework como JPA, Security ou MVC. Pensando em facilitar ainda mais a vida do desenvolvedor, o time do Spring criou um projeto chamado Spring Boot, cujo objetivo é acelerar o processo de configuração da aplicação e estabelecer configurações *default* baseadas nas experiências dos desenvolvedores do próprio Spring.

Spring Boot não necessita de nenhuma ferramenta especial de integração, o que permite que você utilize qualquer IDE ou editor de texto. Podemos usar o Spring Boot como qualquer biblioteca padrão do Java. Para isso, basta incluir o .jar no classpath mas recomendamos o uso de uma ferramenta de *build* como Maven ou Gradle. Neste projeto utilizaremos o Maven junto com a IDE Eclipse. Você vai precisar do *Eclipse IDE for Enterprise Java Developers* (versão Neon ou superior).

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

3.2 DEPENDÊNCIAS DO PROJETO

Administrar dependências é sempre um aspecto crítico em um projeto, além de consumir um tempo precioso. Quanto mais tempo perdemos com configuração, menos tempo teremos com questões mais importantes do projeto (como a regra de negócio). Os *starters* do Spring Boot ajudam exatamente nesta questão. Os *starters* são um conjunto de dependências que você pode incluir em seu projeto. Geralmente seu arquivo `pom.xml` (no caso do uso do Maven) herda de `spring-boot-starter-parent` e declara as dependências de um ou mais *starters*:

```
<!-- Herda padrões do Spring Boot -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
</parent>
```

O SpringBoot utiliza o `groupId` `org.springframework.boot` que é o *starter* principal, incluindo auto-configuração, *logging* e suporte a YAML. Ele também provê mais *starters* que permitem você adicionar jars no classpath. Outros *starters* fornecem dependências que comumente necessitamos para desenvolver um tipo específico de aplicação. No projeto do Fórum da Alura precisaremos incluir as dependências do Spring MVC já que se trata de uma aplicação web:

```
<!-- Herda padrões do Spring Boot -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.4.RELEASE</version>
</parent>

<!-- Adiciona dependências de uma web application -->
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

O `spring-boot-starter-web` é o responsável pelo *build* da parte web da aplicação. Além do Spring MVC, inclui suporte na criação de APIs RESTful e usa o Tomcat como servidor padrão. O mesmo deve ser feito para acrescentar dependências necessárias de outros módulos do framework.

Para tornar esse processo menos braçal, iremos utilizar outra ferramenta para inicializar nosso projeto chamada de Spring Initializr, na qual é possível já adicionar todas as dependências que usaremos no projeto, sem necessidade de colocar cada *starter* na mão. Acesse a página <https://start.spring.io/>, clique em *More Options* e preencha como na figura abaixo:

The screenshot shows the Spring Initializr web application. At the top, there's a logo and the text "Spring Initializr" with the subtitle "Bootstrap your application". Below this, there are several tabs and dropdown menus for configuring a Maven Project:

- Project**: Maven Project (selected), Gradle Project
- Language**: Java (selected), Kotlin, Groovy
- Spring Boot**: 2.2.0 M2, 2.2.0 (SNAPSHOT), 2.1.5 (SNAPSHOT), 2.1.4 (selected), 1.5.20
- Project Metadata**:
 - Group: br.com.alura
 - Artifact: forum
 - Name: forum
 - Description: Projeto base do curso FJ-27 Spring Framework da Caelum
 - Package Name: br.com.alura.forum
 - Packaging: Jar (selected), War
 - Java Version: 12, 11 (selected), 8
- Generate Project - alt + enter**

At the bottom left, there's a copyright notice: "© 2013-2019 Pivotal Software start.spring.io is powered by Spring Initializr and Pivotal Web Services".

Figura 3.1: Spring Initializr

Para adicionar as dependências é necessário procurar por cada uma delas no campo *Search for dependencies to add* da área *Dependencies* e pressionar ENTER. Depois, basta clicar no botão *Generate Project* para baixar o .zip. Repare que iniciaremos o projeto com as seguintes dependências:

- Web
- JPA
- MySQL

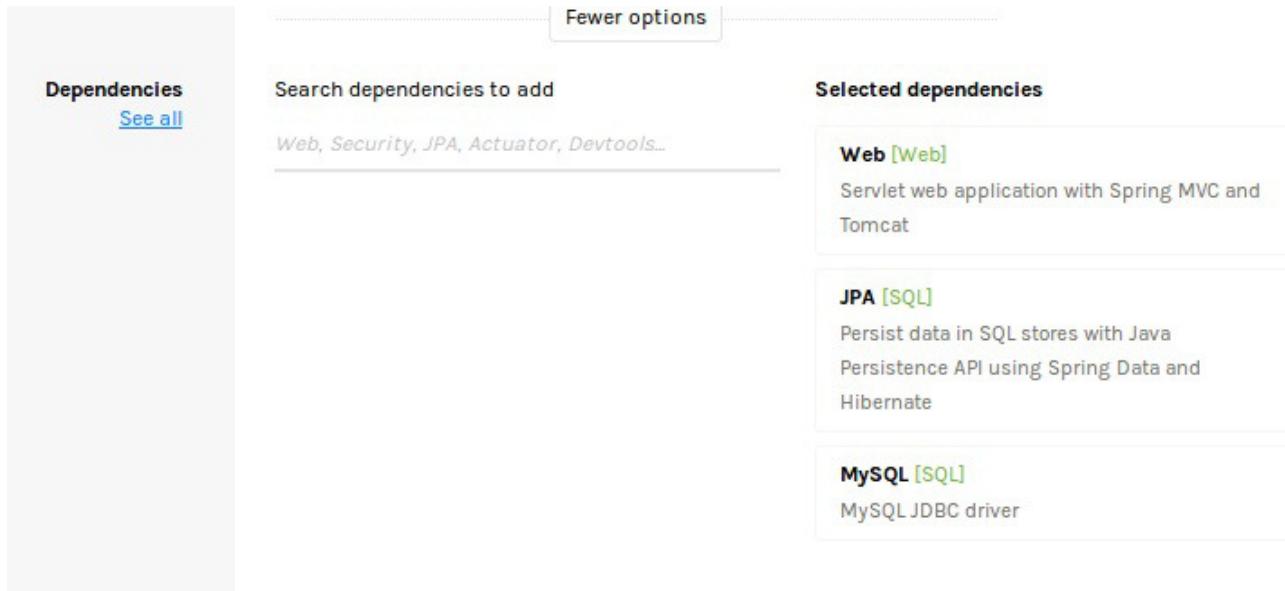


Figura 3.2: Spring Initializr Dependencies

O Spring Initializr já vai criar o projeto com toda a estrutura que especificamos. Agora basta abrir o Eclipse e importar o projeto descompactado. Vá em *File > Import*, selecione *Maven > Existing Maven Projects* e importe o conteúdo descompactado. A estrutura do projeto deve ficar como o exemplo abaixo:

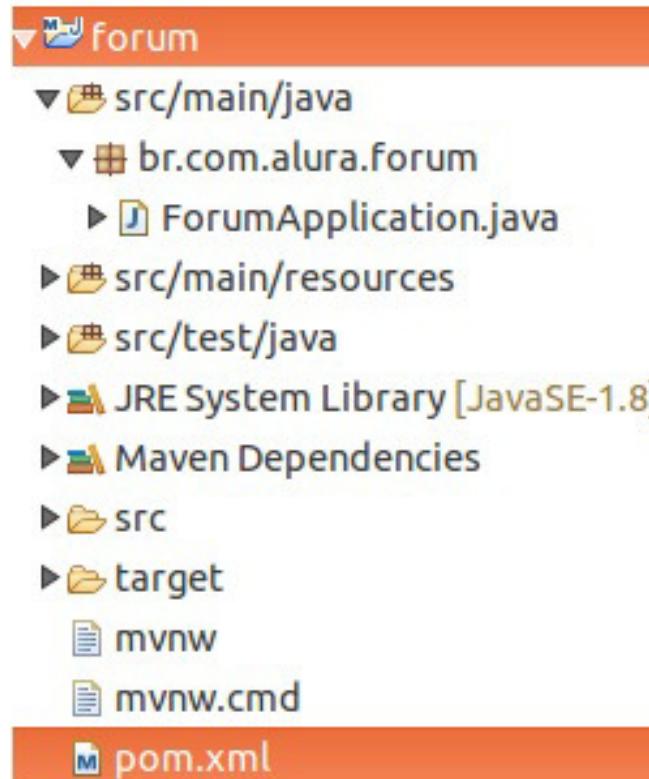


Figura 3.3: Estrutura Projeto

Por padrão, o Spring Boot cria uma classe principal contendo o método `main()`. Para iniciar a aplicação basta executar esta classe. Note que a aplicação é uma *Java Application* e o Spring Boot usa um *container* web embutido (o Tomcat é usado por padrão) para facilitar o desenvolvimento. A classe `ForumApplication` inicializa o nosso aplicativo, iniciando o Spring, que, por sua vez, inicia o servidor web Tomcat configurado automaticamente:

```
@SpringBootApplication
public class ForumApplication {

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

A anotação `@SpringBootApplication` configura algumas coisas do Spring e ela inclui automaticamente as anotações:

- `@Configuration`: permite registrar *beans* extras no contexto ou importar classes de configuração adicionais.
- `@ComponentScan`: habilita `@Component` no pacote onde o aplicativo está localizado.
- `@EnableAutoConfiguration`: ativa o mecanismo de configuração automática do Spring Boot.

Com isso já é possível rodar a aplicação. Ao fazê-lo, recebemos a seguinte exceção:

```
*****
APPLICATION FAILED TO START
*****

Description:
Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.
| Reason: Failed to determine a suitable driver class
```

Figura 3.4: Exception DataSource

O erro alerta para um problema com a configuração do `DataSource`. Incluímos a dependência do JPA, mas o Spring não conhece os dados de acesso da nossa base de dados para estabelecer conexões, então precisamos expor essas informações para que ele as leve em consideração na autoconfiguração do `DataSource`.

O Spring Boot usa o Hibernate como implementação padrão da especificação JPA. Além das informações de nossa base de dados, também podemos passar as propriedades para configuração do Hibernate.

3.3 CONFIGURANDO O DATA SOURCE

Na pasta `src/main/resources` existe um arquivo chamado `application.properties` que é

criado e usado pelo Spring Boot para configurar propriedades específicas da aplicação como as propriedades do `DataSource`. Cada propriedade possui uma chave e um valor:

```
# data source
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/fj27_spring?createDatabaseIfNotExist=true&useSSL=false
spring.datasource.username=<SEU USUARIO>
spring.datasource.password=<SUA SENHA>

# jpa properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL55Dialect
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true
```

O `application.properties` é um recurso provido pelo `spring-boot-starter-parent`. Ele também aceita em formato `yml` e você pode substituí-lo por um arquivo chamado `application.yml`. Neste [link](#) você encontra várias outras propriedades possíveis de configurar conforme as dependências utilizadas pelo projeto.

Agora, ao rodar novamente a aplicação, não recebemos nenhuma exceção. Repare que na saída do console encontramos várias informações como a que ele conseguiu inicializar o Tomcat na porta 8080:

```
Tomcat started on port(s): 8080 (http) with context path ''
```

Acessando `http://localhost:8080/` pelo navegador temos como resultado a página abaixo:

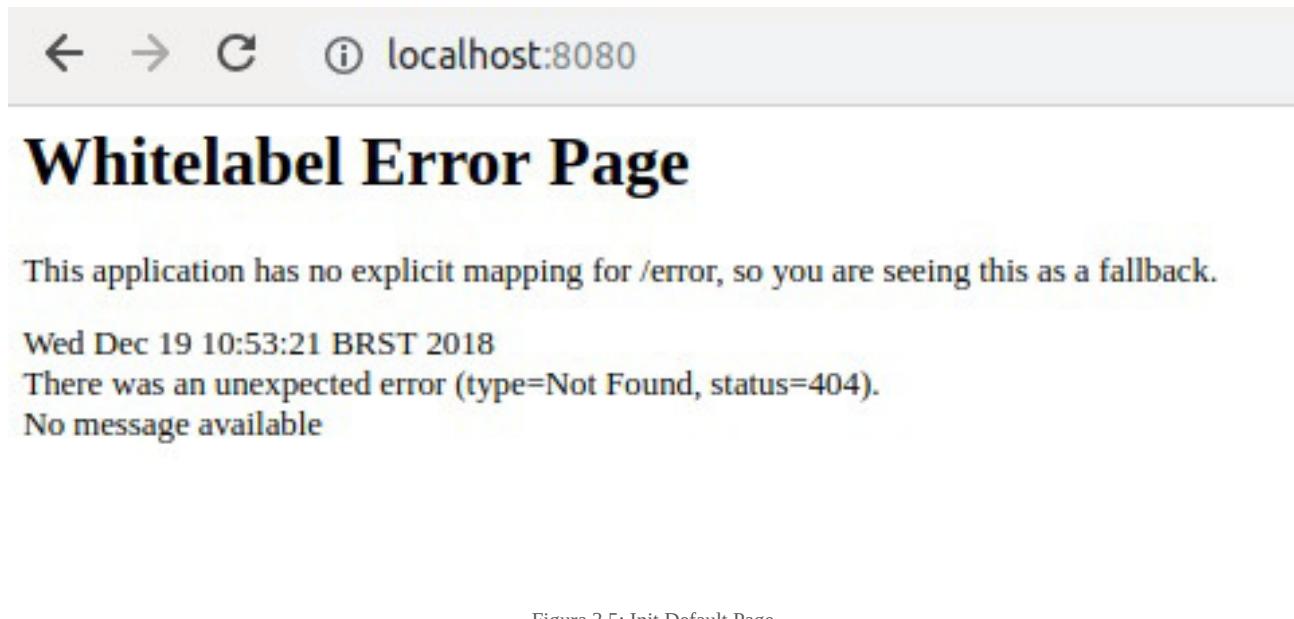


Figura 3.5: Init Default Page

Não se preocupe, essa é a página padrão que o Spring apresenta quando um endereço não pode ser atendido por ninguém da aplicação. Nossa primeira tarefa será modificar este comportamento. Quando o endereço `http://localhost:8080/` for acessado pelo navegador, a aplicação deve ser capaz de responder e mostrar uma outra mensagem customizada no corpo da página.

3.4 PRIMEIRO CONTROLLER

Em geral, com o Spring MVC, assim como com quase qualquer outro framework MVC baseado em ações, sempre que quisermos responder a uma URL, teremos que criar uma classe responsável por isso. No pacote `br.com.alura.forum.controller`, criaremos a classe `HomeController`:

```
public class HomeController {  
}
```

Por convenção, é sugerido que o nome da classe termine com o sufixo `Controller`. Para avisar ao Spring que essa classe é responsável por atender requisições feitas por um cliente (no caso, o navegador), anotamos a classe com `@Controller`:

```
@Controller  
public class HomeController {  
}
```

Vamos criar a ação que vai carregar uma nova mensagem na tela quando o cliente acessar `/`. Criamos, então, um método que retorna a mensagem e o anotamos com `@RequestMapping` para mapear o caminho específico:

```
@Controller  
public class HomeController {  
  
    @RequestMapping("/")  
    public String index() {  
        System.out.println("Log do servidor de que foi feita uma requisição para '/'.");  
        return "Bem vindo ao Forum da Alura!";  
    }  
}
```

Agora o Spring é capaz de encontrar alguém que responda o endereço `http://localhost:8080/`. Ao rodar a aplicação novamente e acessar a página, recebemos o seguinte erro:

```
Error resolving template [Bem vindo ao Forum da Alura!], template might not exist or might not be  
accessible by any of the configured Template Resolvers
```

O erro diz que o Spring não pode resolver a apresentação da página já que não encontra nenhum template na aplicação para ser renderizado. Acontece que, quando o método em uma classe `Controller` retorna uma `String`, por padrão o Spring Boot procura por uma página em `src/main/resources/templates` para ser renderizada. Podemos criar uma nova página utilizando Thymeleaf ou acrescentar a anotação `@ResponseBody` avisando o Spring para mudar este comportamento e escrever o retorno do método direto na `response`. Vamos fazer a segunda opção:

```
@Controller  
public class HomeController {  
  
    @RequestMapping("/")  
    @ResponseBody  
    public String index() {
```

```
        System.out.println("Log do servidor de que foi feita uma requisição para '/'");
        return "Bem vindo ao Forum da Alura!";
    }
}
```

Agora, ao acessar a página, conseguimos o resultado desejado:



Figura 3.6: Init Default Page

Já conhece os cursos online Alura?

alura A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

This block contains an advertisement for Alura's online courses. It features the Alura logo, a brief description of the platform's offerings, and a call-to-action button.

3.5 EXERCÍCIOS: RODANDO O PROJETO COM SPRING BOOT

1. Acesse a página do projeto Spring Initializr pelo link: <https://start.spring.io/>. Essa aplicação foi desenvolvida pela equipe do Spring para facilitar a criação dos projetos baseados no framework. Ela nos ajuda a tornar o processo de criação e configuração do projeto muito mais rápido e eficiente.
2. Preencha os campos de *Project Metadata* de acordo com a imagem abaixo. Será pedido que você

especifique a versão do Java. Neste curso utilizaremos o Java 11, mas funcionará com qualquer versão igual ou superior ao Java 8. Aproveite para acrescentar as dependências que utilizaremos neste projeto: Web, JPA e MySQL. Para adicioná-las, digite cada uma delas no campo *Search for dependencies to add* na área *Dependencies* e pressione ENTER. Depois, clique no botão *Generate Project* para baixar o .zip com todo o conteúdo necessário para já começar a escrever código no projeto.

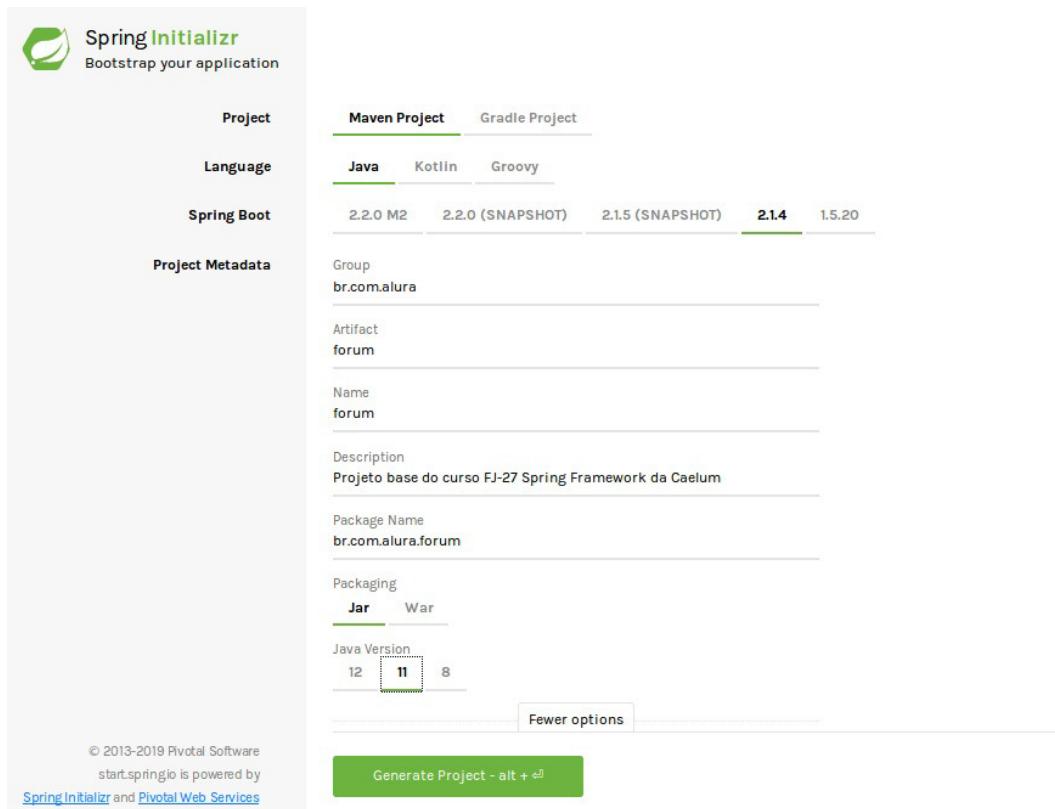


Figura 3.7: Spring Initializr

3. Agora vamos importar o projeto no Eclipse. Extraia o conteúdo do projeto para a pasta /home/springXXXX/workspace. No Eclipse, clique em *File > Import*, selecione a opção *Maven > Existing Maven Projects* e clique em *Next*. Na próxima janela, clique em *Browse*, selecione a pasta do projeto dentro do workspace e clique em *Finish*.

Aguarde o Maven fazer o *download* de todas as dependências após importar o projeto. Caso seja necessário, peça ajuda ao instrutor.

4. Com o projeto importado, abra a classe `AluraForumApplication`, do pacote `br.com.alura.forum`. Rode o projeto clicando com o botão direito do mouse sobre o código da classe e selecionando *Run As > Java Application*.
5. Repare que uma exceção foi lançada. A descrição do erro alerta para um problema com a

configuração do `DataSource`. Como o Spring não conhece os dados de acesso da nossa base de dados para estabelecer conexões, precisamos expor essas informações para que ele as leve em consideração na autoconfiguração do `DataSource`.

```

Console  Progress  Problems  Tasks  Search  JUnit
<terminated> fj27-spring-boot - AluraForumApplication [Spring Boot App] /Library/Java/JavaVirtualMachines/jdk1.8.0_131/jdk/Contents/Home/bin/java (Aug 6, 2018, 5:51:58 PM)
\ \ / _ _ _ _ _ C _ _ _ _ _ \ \ \
( ( ) _ _ | _ _ | _ _ | _ _ | _ _ | _ _ | _ _ )
\ \ _ _ _ | _ _ | _ _ | _ _ | _ _ | _ _ | _ _ |
` _ _ _ . _ _ | _ _ | _ _ | _ _ | _ _ | _ _ |
=====|_|=====|_|/_/|_|/_/|_/
:: Spring Boot ::          (v2.0.4.RELEASE)

2018-08-06 17:52:00.372 INFO 46312 --- [ restartedMain] b.com.alura.forum.AluraForumApplication : Starting AluraForumApplication on Rafaels-MacBook-Pro.local with PID 46312 (/Users
2018-08-06 17:52:00.374 INFO 46312 --- [ restartedMain] b.com.alura.forum.AluraForumApplication : No active profile set, falling back to default profiles: default
2018-08-06 17:52:00.433 INFO 46312 --- [ restartedMain] ConfigServletWebServerApplicationContext : Refreshing org.springframework.boot.web.servlet.context.AnnotationConfigServletWeb
2018-08-06 17:52:01.694 INFO 46312 --- [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Multiple Spring Data modules found, entering strict repository configuration mode!
2018-08-06 17:52:01.729 INFO 46312 --- [ restartedMain] .s.d.r.c.RepositoryConfigurationDelegate : Multiple Spring Data modules found, entering strict repository configuration mode!
2018-08-06 17:52:01.729 INFO 46312 --- [ restartedMain] trationDelegates$BeanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTransactionManagementConfigu
2018-08-06 17:52:02.893 INFO 46312 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2018-08-06 17:52:02.893 INFO 46312 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2018-08-06 17:52:02.934 INFO 46312 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.32
2018-08-06 17:52:02.934 INFO 46312 --- [ost-startStop-1] o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows optimal performance in pro
2018-08-06 17:52:03.018 INFO 46312 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2018-08-06 17:52:03.019 INFO 46312 --- [ost-startStop-1] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 2590 ms
2018-08-06 17:52:03.332 ERROR 46312 --- [ost-startStop-1] o.s.b.web.embedded.tomcat.TomcatStarter : Error starting Tomcat context. Exception: org.springframework.beans.factory.BeanCr
2018-08-06 17:52:03.362 INFO 46312 --- [ restartedMain] o.apache.catalina.core.StandardService : Stopping service [Tomcat]
2018-08-06 17:52:03.375 WARN 46312 --- [ restartedMain] ConfigServletWebServerApplicationContext : Exception encountered during context initialization - cancelling refresh attempt:
2018-08-06 17:52:03.389 INFO 46312 --- [ restartedMain] ConditionEvaluationReportLoggingListener :

Error starting ApplicationContext. To display the conditions report re-run your application with 'debug' enabled.
2018-08-06 17:52:03.398 ERROR 46312 --- [ restartedMain] o.s.b.d.LoggingFailureAnalysisReporter :

*****
APPLICATION FAILED TO START
*****

Description:
| Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class

Action:
Consider the following:
If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.
If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).


```

Figura 3.8: Exception Data Source

Para expor as informações necessárias à configuração do `DataSource`, assim como algumas propriedades importantes para o uso da JPA, abra o arquivo `application.properties`, da pasta `src/main/resources`, e escreva o seguinte conteúdo:

```

# data source
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost/fj27_spring?createDatabaseIfNotExist=true&useSSL=false
spring.datasource.username=root
spring.datasource.password=

# jpa properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL55Dialect
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true

```

6. Rode novamente o projeto e veja que tudo funciona dessa vez.

```

Console  Progress  Problems  Tasks  Search  JUnit
fj27-spring-boot - AluraForumApplication [Spring]
[ restartedMain] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTransactionManagementConfigu
2018-08-06 18:19:36.635 INFO 47229 --- [ restartedMain] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTransactionManagementConfigu
2018-08-06 18:19:37.071 INFO 47229 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2018-08-06 18:19:37.101 INFO 47229 --- [ restartedMain] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2018-08-06 18:19:37.101 INFO 47229 --- [ restartedMain] org.apache.catalina.core.StandardEngine : Starting Servlet Engine: Apache Tomcat/8.5.32
2018-08-06 18:19:37.108 INFO 47229 --- [ost-startStop-1] o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows optimal performance in prc
2018-08-06 18:19:37.108 INFO 47229 --- [ost-startStop-1] o.a.catalina.core.AprLifecycleListener : Initializing Spring embedded WebApplicationContext
2018-08-06 18:19:37.184 INFO 47229 --- [ost-startStop-1] o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in 2097 ms
2018-08-06 18:19:38.240 INFO 47229 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'characterEncodingFilter' to: [//*]
2018-08-06 18:19:38.240 INFO 47229 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'webMvcMetricsFilter' to: [//*]
2018-08-06 18:19:38.240 INFO 47229 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'hiddenHttpMethodFilter' to: [//*]
2018-08-06 18:19:38.241 INFO 47229 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'httpPutFormContentFilter' to: [//*]
2018-08-06 18:19:38.241 INFO 47229 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'requestContextFilter' to: [//*]
2018-08-06 18:19:38.241 INFO 47229 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean : Mapping filter: 'httpTraceFilter' to: [//*]
2018-08-06 18:19:38.241 INFO 47229 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean : Servlet dispatcherServlet mapped to [/]
2018-08-06 18:19:38.350 INFO 47229 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2018-08-06 18:19:38.570 INFO 47229 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2018-08-06 18:19:38.616 INFO 47229 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Building JPA container EntityManagerFactory for persistence unit 'default'
2018-08-06 18:19:38.631 INFO 47229 --- [ restartedMain] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [
    name: default
    ...
]
2018-08-06 18:19:38.742 INFO 47229 --- [ restartedMain] org.hibernate.Version : HHH000412: Hibernate Core {5.2.17.Final}
2018-08-06 18:19:38.744 INFO 47229 --- [ restartedMain] org.hibernate.cfg.Environment : HHH000206: hibernate.properties not found
2018-08-06 18:19:38.797 INFO 47229 --- [ restartedMain] o.hibernate.annotations.common.Version : HCCN000001: Hibernate Commons Annotations {5.0.1.Final}
2018-08-06 18:19:38.923 INFO 47229 --- [ restartedMain] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
2018-08-06 18:19:39.165 INFO 47229 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2018-08-06 18:19:39.255 INFO 47229 --- [ restartedMain] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.web.servlet.DispatcherServlet]
2018-08-06 18:19:39.535 INFO 47229 --- [ restartedMain] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.web.servlet.context.AnnotationConfigEmbeddedWebApplicationContext@50d35a5e: startup sequence 1
2018-08-06 18:19:39.587 WARN 47229 --- [ restartedMain] o.WebConfiguration$JpaWebMvcConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be slow. Please see http://www.spring君.technet.com/doc/Handler/68d2266e.html
2018-08-06 18:19:39.662 INFO 47229 --- [ restartedMain] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "[{/error}]" onto public org.springframework.http.ResponseEntity<java.util.Map<String, Object>> org.springframework.web.servlet.HandlerExceptionResolver.resolveError(HttpServletRequest, HttpServletResponse)
2018-08-06 18:19:39.664 INFO 47229 --- [ restartedMain] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "[{/error}]" onto public org.springframework.web.servlet.ModelAndView org.springframework.web.servlet.HandlerExceptionResolver.resolveError(HttpServletRequest, HttpServletResponse)
2018-08-06 18:19:39.703 INFO 47229 --- [ restartedMain] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.servlet.resource.ResourceHttpRequestHandler]
2018-08-06 18:19:39.703 INFO 47229 --- [ restartedMain] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.HandlerMapping]
2018-08-06 18:19:39.762 WARN 47229 --- [ restartedMain] ion$DefaultTemplateResolverConfiguration : Cannot find template location: classpath:/templates/ (please add some templates or check your Thymeleaf configuration)
2018-08-06 18:19:40.349 INFO 47229 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729
2018-08-06 18:19:40.374 INFO 47229 --- [ restartedMain] s.b.a.e.w.s.WebMvcEndpointLinksResolver : Exposing 2 endpoint(s) beneath base path '/actuator'
2018-08-06 18:19:40.375 INFO 47229 --- [ restartedMain] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "[{/actuator/health},methods=[GET],produces=[application/vnd.spring-boot.actuator+json, application/json]]" onto public org.springframework.web.servlet.ModelAndView org.springframework.web.servlet.HandlerMethod.handleRequest(HttpServletRequest, HttpServletResponse)
2018-08-06 18:19:40.376 INFO 47229 --- [ restartedMain] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "[{/actuator/info},methods=[GET],produces=[application/vnd.spring-boot.actuator+json, application/json]]" onto public org.springframework.web.servlet.ModelAndView org.springframework.web.servlet.HandlerMethod.handleRequest(HttpServletRequest, HttpServletResponse)
2018-08-06 18:19:40.433 INFO 47229 --- [ restartedMain] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
2018-08-06 18:19:40.436 INFO 47229 --- [ restartedMain] o.s.j.e.a.AnnotationMBeanExporter : Bean with name 'dataSource' has been autodetected for JMX exposure
2018-08-06 18:19:40.446 INFO 47229 --- [ restartedMain] o.s.j.e.a.AnnotationMBeanExporter : Located MBean 'dataSource': registering with JMX server as MBean [com.zaxxer.hikaricp:type=HikariCP]
2018-08-06 18:19:40.496 INFO 47229 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2018-08-06 18:19:40.502 INFO 47229 --- [ restartedMain] b.com.alura.forum.AluraForumApplication : Started AluraForumApplication in 5.8 seconds (JVM running for 6.668)

```

Figura 3.9: Up

- Crie a classe `HomeController` no pacote `br.com.alura.forum.controller`, com o seguinte conteúdo:

```

@Controller
public class HomeController {

    @RequestMapping("/")
    @ResponseBody
    public String index() {
        System.out.println("Log do servidor de que foi feita uma requisição para '/');
        return "Bem vindo ao Forum da Alura!";
    }
}

```

- Rode a aplicação novamente e acesse o endereço <http://localhost:8080/> no navegador.

DANDO INÍCIO À NOSSA API

O fórum da Alura já possui a parte das telas desenvolvida pela equipe de *front-end* da Alura. A equipe precisa que disponibilizemos a lista de tópicos do fórum e utilizou React.js (uma biblioteca JavaScript) para desenvolver as telas. Uma preocupação que nos vem à cabeça é como vamos importar as telas já desenvolvidas para nosso projeto e se nossa configuração dá suporte para as ferramentas que a equipe de *front-end* utilizou.

Isso é uma preocupação importante! Além disso, outra equipe pode desenvolver outras telas com ferramentas diferentes (como Angular.js ou Vue.js) para uma outra aplicação de uso interno da Alura que também necessite da lista de tópicos. E considere que outra equipe está desenvolvendo a aplicação para plataforma mobile e também precisará da lista de todos os tópicos do fórum.

Veja que agora o problema ganhou uma complexidade maior. Três ou mais aplicações distintas dependem de um mesmo recurso. Não queremos desenvolver uma aplicação com as mesmas funcionalidades para cada uma dessas aplicações que dependem do recurso da nossa listagem. Para evitar todo esse retrabalho, será necessário que todas essas aplicações (desenvolvidas com ferramentas e linguagens diferentes) consigam, de alguma forma, se integrar com a nossa aplicação (que será responsável por atender esse serviço da listagem).

4.1 WEB SERVICES

Precisamos, então, que nossa aplicação se comporte como uma aplicação que "serve" as demais, independente da linguagem e ferramentas que foram desenvolvidas. Neste caso, precisaremos desenvolver uma aplicação que seja um **Web Service**. Um *Web Service* é uma aplicação servidora, que disponibiliza um ou mais serviços para seus clientes. *Web Services* permitem desenvolver aplicações distribuídas que cooperem e compartilhem informações entre si. Veja o esquema abaixo:

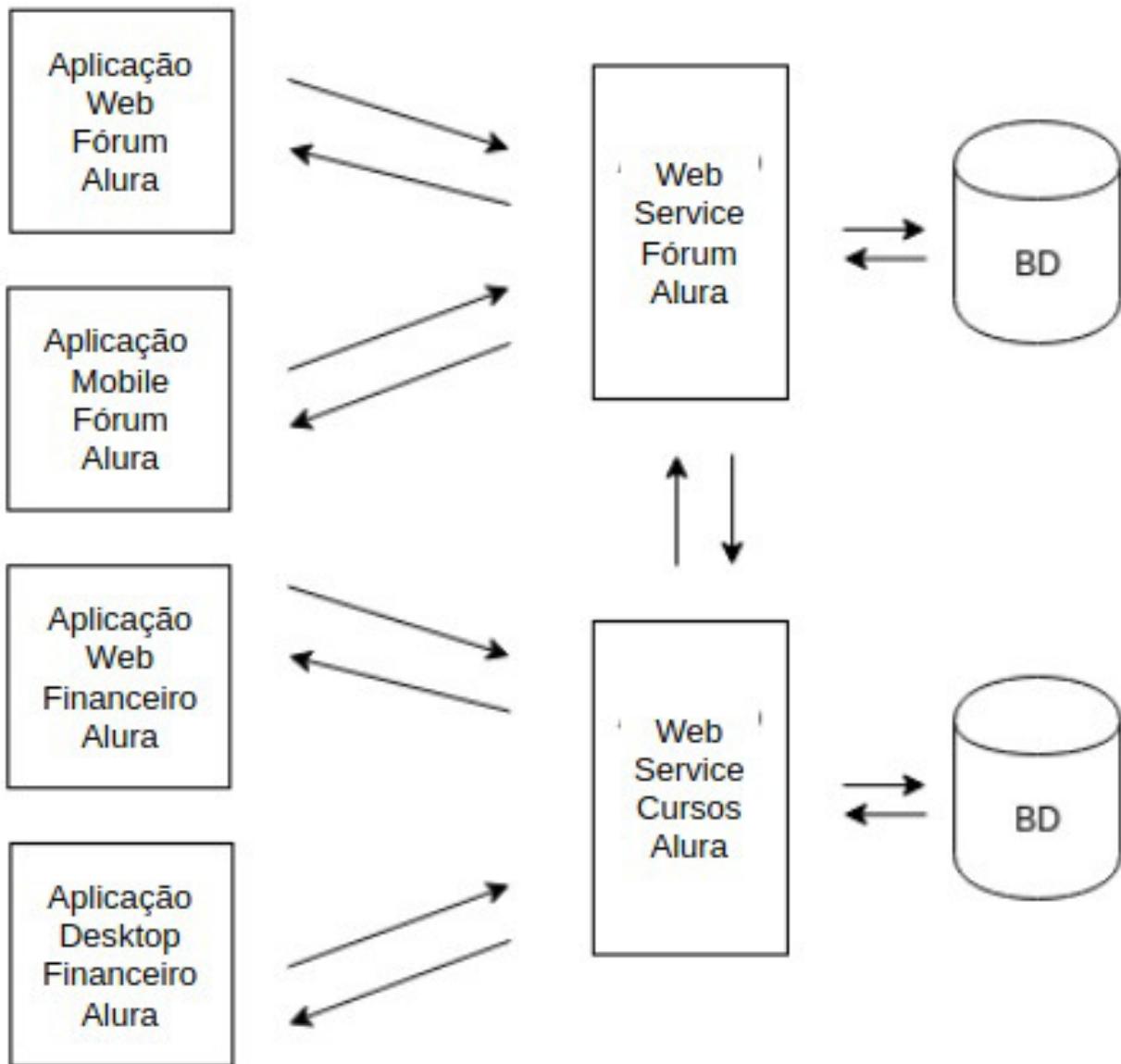


Figura 4.1: Web Services

As aplicações *Web* e *Mobile* do Fórum Alura, desenvolvidas em plataformas e linguagens diferentes, consomem um serviço prestado pela aplicação *Web Service* do Fórum da Alura, desenvolvida em outra plataforma. Já as aplicações *Web* e *Desktop* do setor financeiro da Alura consomem o serviço de cursos da Alura. E os serviços do fórum e de cursos podem ou não se comunicar entre si. As aplicações à esquerda são o que chamamos de aplicações clientes de uma aplicação servidora (fórum e cursos, por exemplo) e a aplicação servidora geralmente é a que contém toda a regra de negócio e é quem de fato acessa a base de dados.

A comunicação entre aplicação cliente e servidor é feita através de mensagens do protocolo HTTP. O cliente faz uma requisição HTTP para o servidor, que envia de volta uma mensagem que é uma resposta HTTP. Pense nessas mensagens como um conjunto de informações. Para que ela seja efetiva, ou seja,

para que ambos os lados entendam o conteúdo enviado/recebido, é necessário que exista um espécie de linguagem comum e que regras comuns sejam seguidas. Portanto, é necessário que sigam uma espécie de protocolo.

Saber inglês é muito importante em TI



cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

4.2 PROTOCOLOS

SOA (*Service Oriented Architecture*) é um modelo arquitetural que trabalha funcionalidades como serviços, ou seja, uma Arquitetura Orientada a Serviços independente das tecnologias utilizadas. Criar esses serviços significa expor uma API que encapsule a complexidade dessas funcionalidades para o cliente.

Como vimos, para distribuir uma mesma informação para diferentes clientes é necessário que exista um protocolo para essa comunicação acontecer. A informação trocada entre cliente e servidor necessita ser comprehensível por ambos os lados. O SOA utiliza o protocolo SOAP (*Simple Object Access Protocol*) e as informações são trocadas em formato XML seguindo o padrão definido pelo protocolo. O SOAP define que uma mensagem deve estar dentro de um envelope que possui um cabeçalho e o corpo da mensagem. O SOAP surgiu em 1998 é utilizado até hoje, mas a verbosidade do protocolo e a baixa performance de arquivos XML fizeram com que desenvolvedores buscassem outras alternativas. Apesar disso, o protocolo ainda é muito utilizado por aplicações financeiras por oferecer soluções mais robustas de segurança.

Em 2000, surge o REST (*Representational State Transfer*), um novo modelo para softwares distribuídos a ser utilizado na evolução da arquitetura do protocolo HTTP. Alguns desenvolvedores perceberam que poderiam utilizar o modelo REST para implementação de *Web Services* e passaram a

utilizá-lo como uma alternativa ao SOAP. O REST ganhou a preferência pelos desenvolvedores por sua simplicidade, flexibilidade e por utilizar uma abordagem que aproveita, em grande medida, o próprio protocolo HTTP.

Para saber mais

Os modelos REST e SOA são mencionados de maneira simplificada neste apostila. Caso você tenha interesse em se aprofundar mais sobre o assunto, indicamos o curso [FJ-36 - Curso SOA na Prática: Integração com Web Services e Mensageria](#) da Caelum.

4.3 RECURSOS

Em uma aplicação REST, um recurso é algo que a aplicação gerencia. Por exemplo, nossa aplicação deve gerenciar os tópicos do fórum da Alura, portanto um tópico é um recurso. E, no modelo REST, todo recurso deve possuir uma identificação única. Essa identificação deve ser feita utilizando o conceito de URI (*Uniform Resource Identifier*). Exemplo:

- <http://alura.com.br/topics>
- <http://alura.com.br/topics/3>

As URI's são a interface de utilização dos seus serviços e funcionam como um contrato que será utilizado pela aplicação cliente para acessá-los. Também são comumente chamados de *endpoints*. Além da URI, é necessário que o cliente informe o tipo de ação que ele deseja fazer, como atualizar ou deletar um recurso. O protocolo HTTP possui diversos métodos para especificar uma ação. Exemplos:

Método HTTP	URI	Descrição
GET	http://alura.com.br/topics	Obter dados de todos os tópicos.
GET	http://alura.com.br/topics/1	Obter dados de um determinado tópico.
POST	http://alura.com.br/topics	Criar um novo tópico.
PUT	http://alura.com.br/topics/1	Atualizar um tópico específico.
DELETE	http://alura.com.br/topics/2	Deletar um tópico específico.

Os métodos `GET` , `POST` , `PUT` e `DELETE` são os mais utilizados e o protocolo HTTP possui outros (como `HEAD` e `OPTIONS`) que podem ser úteis para outros tipos de serviço. Saiba mais neste [link](#)

Um recurso pode ser representado de diversas maneiras, utilizando formatos específicos, tais como `XML` , `JSON` , `HTML` , `CSV` . Em nossa aplicação utilizaremos o formato `JSON` (*JavaScript Object Notation*) por ser de fácil leitura, possuir alta performance e ser o formato mais utilizado atualmente na

Web.

A aplicação cliente já tem a tela pronta que apresenta a lista de tópicos do fórum e espera receber os dados em formato JSON . Cada tópico é representado pelo seguinte JSON :

```
{  
    "id": 33,  
    "shortDescription": "Breve título da dúvida",  
    "secondsSinceLastUpdate": 30, // segundos desde a última atualização  
    "ownerName": "Fulano da Silva", // nome do usuário que publicou o tópico  
    "courseName": "Java e JSF", // nome do curso ao qual o tópico foi registrado  
    "subcategoryName": "Java", // subcategoria do curso  
    "categoryName": "Programação", // categoria do curso  
    "numberOfResponses": 0, // numero de respostas atualmente no tópico  
    "solved": false // esta solucionada ou não  
}
```

Nossa API deverá ser capaz de responder à solicitação do cliente, que executará uma requisição HTTP GET pela URI /api/topics . Como trabalharemos localmente, a URI completa será http://localhost/api/topics . Antes de colocar a mão na massa, vamos conhecer nosso modelo.

4.4 MODELO

Importaremos as classes de modelo prontas da pasta do curso para iniciar nossa aplicação. A classe Topic , que representa um tópico do fórum, está modelada da seguinte maneira:

```
@Entity  
public class Topic {  
  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private Long id;  
  
    private String shortDescription;  
  
    @Lob  
    private String content;  
  
    private Instant creationInstant = Instant.now();  
    private Instant lastUpdate = Instant.now();  
  
    @Enumerated(EnumType.STRING)  
    private TopicStatus status = TopicStatus.NOT_ANSWERED;  
  
    @ManyToOne  
    private User owner;  
  
    @ManyToOne  
    private Course course;  
  
    @OneToOne(mappedBy = "topic")  
    @LazyCollection(LazyCollectionOption.EXTRA)  
    private List<Answer> answers = new ArrayList<>();  
  
    public Topic(String shortDescription, String content, User owner, Course course) {  
        super();  
    }
```

```

        this.shortDescription = shortDescription;
        this.content = content;
        this.owner = owner;
        this.course = course;
    }

    //getters
}

```

Navegue pelas classes do pacote `br.com.alura.forum.model` e note que as classes de modelo estão relacionadas e que os relacionamentos estão representados pelas anotações da especificação JPA (Java Persistence API), que nos serão úteis no próximo capítulo.

Um `Topic` representa uma dúvida no fórum da Alura e está relacionado a um dono, que é um usuário do fórum (representado pelo objeto `User`), a um curso (representado pelo objeto `Course`), a uma lista de respostas (cada resposta é representada pelo objeto `Answer`) e a um `status` (`NOT_ANSWERED`, `NOT_SOLVED`, `SOLVED`, `CLOSED`), que está representado pelo enum `TopicStatus`. Além disso, um tópico possui uma pequena descrição (um breve título da dúvida), um conteúdo (o conteúdo da dúvida) e também os instantes de criação e de última modificação do tópico.

No fórum é possível que um usuário abra um novo tópico (uma nova dúvida) sobre um curso, responda a um tópico aberto, busque tópicos por categoria, etc... Vamos trabalhar cada uma dessas funcionalidades durante o curso. Neste momento, vamos criar alguns tópicos e disponibilizar esta informação para que a aplicação cliente possa apresentá-los no navegador.

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

4.5 CONTROLLER DE TÓPICOS

Voltando ao nosso problema, devemos então ter a classe controladora `TopicController` com um

método que processe o recurso esperado pelo cliente, ou seja, a lista de tópicos. Usamos, como visto no capítulo anterior, as anotações `@Controller` e `@RequestMapping` do Spring MVC. Mapeamos o método com `/api/topics`, que é a requisição que a aplicação cliente irá executar para buscar pela lista de tópicos:

```
@Controller
public class TopicController {

    @RequestMapping("/api/topics")
    public void listTopics(){
        // processar lógica
    }
}
```

Vamos então criar um tópico para apresentar ao cliente:

```
@Controller
public class TopicController {

    @RequestMapping("/api/topics")
    public Topic listTopics(){

        Category subcategory = new Category("Java", new Category("Programação"));
        Course course = new Course("Java e JSF", subcategory);
        Topic topic = new Topic("Problemas com o JSF",
            "Erro ao fazer conversão da data",
            new User("Fulano", "fulano@gmail.com", "123456"), course);

        return topic;
    }
}
```

Agora, precisamos devolver o tópico em formato `JSON`. Para isso, vamos trocar a anotação `@Controller` pela anotação `@RestController` também do módulo web do Spring. Essa anotação trata a classe `TopicController` como um `Controller` e assume que todos os métodos da classe assumam a anotação `@ResponseBody` por padrão. Como vimos, a anotação `@ResponseBody` retorna um objeto de domínio ao invés de uma `view`. Isso é interessante para nós já que nossa API apenas retornará a informação em formato `JSON` e não uma página com este conteúdo.

Além disso, vamos trocar a anotação `@RequestMapping` por uma anotação mais semântica. A anotação `@GetMapping` é uma anotação do Spring que serve como atalho para `@RequestMapping(method = RequestMapping.GET)` além de possuir outros parâmetros, como o `produces`, em que especificaremos que o retorno produzido será em formato `JSON` através da classe do Spring `MediaType`:

```
@RestController
public class TopicController {

    @GetMapping(value="api/topics", produces=MediaType.APPLICATION_JSON_VALUE)
    public Topic listTopics(){

        Category subcategory = new Category("Java", new Category("Programação"));

        return topic;
    }
}
```

```

        Course course = new Course("Java e JSF", subcategory);
        Topic topic = new Topic("Problemas com o JSF",
            "Erro ao fazer conversão da data",
            new User("Fulano", "fulano@gmail.com", "123456"), course);

        return topic;
    }
}

```

Dessa maneira, o Spring já converte o retorno do método, o objeto `topic`, para formato JSON. Isso é possível já que o `starter-web` contém todas as bibliotecas necessárias para fazer o *parse* do objeto e o Spring cuida de todo esse trabalho para nós. Agora, ao rodar a aplicação e acessar `http://localhost:8080/api/topics` pelo navegador, obtemos:

```
{
    "id": null,
    "shortDescription": "Problemas com o JSF",
    "content": "Erro ao fazer conversão da data",
    "creationInstant": "2018-12-26T13:32:25.144Z",
    "lastUpdate": "2018-12-26T13:32:25.144Z",
    "status": "NOT_ANSWERED",
    "owner": {
        "id": null,
        "name": "Fulano",
        "password": "123456",
        "email": "fulano@gmail.com"
    },
    "course": {
        "name": "Java e JSF",
        "subcategory": {
            "id": null,
            "name": "Java",
            "category": {
                "id": null,
                "name": "Programação",
                "category": null,
                "subcategoryNames": []
            },
            "subcategoryNames": []
        },
        "subcategoryName": "Java",
        "categoryName": "Programação"
    },
    "answers": [],
    "ownerName": "Fulano",
    "ownerEmail": "fulano@gmail.com",
    "numberOfAnswers": 0
}
```

Veja que conseguimos retornar um `Topic` no formato JSON. Mas este não é o JSON esperado pelo cliente, além de ser apenas um tópico e não uma lista de tópicos.

Como nosso modelo não corresponde ao que o cliente espera, precisamos criar uma classe que seja um espelho do JSON esperado. Esse tipo de classe é chamada de DTO (Data Transfer Object), utilizada exatamente para transferir dados de uma camada para outra. Criaremos a classe `TopicBriefOutputDto` no pacote `br.com.alura.forum.controller.dto.output` com os atributos que o cliente espera

receber:

```
public class TopicBriefOutputDto {  
  
    private Long id;  
    private String shortDescription;  
    private long secondsSinceLastUpdate;  
    private String ownerName;  
    private String courseName;  
    private String subcategoryName;  
    private String categoryName;  
    private int numberofResponses;  
    private boolean solved;  
  
    public TopicBriefOutputDto(Topic topic) {  
        this.id = topic.getId();  
        this.shortDescription = topic.getShortDescription();  
        this.secondsSinceLastUpdate = getSecondsSince(topic.getLastUpdate());  
        this.ownerName = topic.getOwner().getName();  
        this.courseName = topic.getCourse().getName();  
        this.subcategoryName = topic.getCourse().getSubcategory().getName();  
        this.categoryName = topic.getCourse().getCategoryName();  
        this.numberofResponses = topic.getNumberOfAnswers();  
        this.solved = TopicStatus.SOLVED.equals(topic.getStatus());  
    }  
  
    private long getSecondsSince(Instant lastUpdate) {  
        return Duration.between(lastUpdate, Instant.now())  
            .get(ChronoUnit.SECONDS);  
    }  
  
    //getters  
}
```

Mude o retorno do método `listTopics()` de `TopicController` para retornar nosso DTO :

```
@RestController  
public class TopicController {  
  
    @GetMapping(value="api/topics", produces=MediaType.APPLICATION_JSON_VALUE)  
    public TopicBriefOutputDto listTopics(){  
  
        Category subcategory = new Category("Java", new Category("Programação"));  
        Course course = new Course("Java e JSF", subcategory);  
        Topic topic = new Topic("Problemas com o JSF",  
            "Erro ao fazer conversão da data",  
            new User("Fulano", "fulano@gmail.com", "123456"), course);  
  
        return new TopicBriefOutputDto(topic);  
    }  
}
```

Acesse novamente `http://localhost:8080/api/topics` e veja que agora obtemos a representação JSON do tópico como esperado pelo cliente:

```
{  
    "id": null,  
    "shortDescription": "Problemas com o JSF",  
    "secondsSinceLastUpdate": 0,  
    "ownerName": "Fulano",  
    "courseName": "Java e JSF",  
}
```

```

        "subcategoryName": "Java",
        "categoryName": "Programação",
        "numberOfResponses": 0,
        "solved": false
    }
}

```

Ótimo! Agora falta devolvermos uma lista ao invés de apenas um tópico. Vamos criar uma lista com vários tópicos e modificar novamente o retorno do método `listTopics()`. Para facilitar, criaremos uma lista contendo 3 tópicos iguais, aproveitando o tópico criado anteriormente:

```

@RestController
public class TopicController {

    @GetMapping(value="api/topics", produces=MediaType.APPLICATION_JSON_VALUE)
    public List<TopicBriefOutputDto> lista(){

        Category subcategory = new Category("Java", new Category("Programação"));
        Course course = new Course("Java e JSF", subcategory);
        Topic topic = new Topic("Problemas com o JSF",
            "Erro ao fazer conversão da data",
            new User("Fulano", "fulano@gmail.com", "123456"), course);

        List<Topic> topics = Arrays.asList(topic, topic, topic);
        return TopicBriefOutputDto.listFromTopics(topics);
    }
}

```

Não esqueça de criar o método `listFromTopics()` na classe `TopicBriefOutputDto`, que recebe uma lista de `Topic` e retorna uma lista de `TopicBriefOutputDto`:

```

public static List<TopicBriefOutputDto> listFromTopics(List<Topic> topics) {
    return topics.stream()
        .map(TopicBriefOutputDto::new)
        .collect(Collectors.toList());
}

```

Acesse novamente `http://localhost:8080/api/topics` e veja que agora a resposta gerada é uma lista de tópicos em formato `JSON`, como esperado pelo cliente:

```

[
    {
        "id": null,
        "shortDescription": "Problemas com o JSF",
        "secondsSinceLastUpdate": 0,
        "ownerName": "Fulano",
        "courseName": "Java e JSF",
        "subcategoryName": "Java",
        "categoryName": "Programação",
        "numberOfResponses": 0,
        "solved": false
    },
    {
        "id": null,
        "shortDescription": "Problemas com o JSF",
        "secondsSinceLastUpdate": 0,
        "ownerName": "Fulano",
        "courseName": "Java e JSF",
        "subcategoryName": "Java",
        "categoryName": "Programação",
        "numberOfResponses": 0,
        "solved": false
    }
]

```

```

    "numberOfResponses": 0,
    "solved": false
},
{
"id": null,
"shortDescription": "Problemas com o JSF",
"secondsSinceLastUpdate": 0,
"ownerName": "Fulano",
"courseName": "Java e JSF",
"subcategoryName": "Java",
"categoryName": "Programação",
"numberOfResponses": 0,
"solved": false
}
]

```

4.6 CORS

Bom, já alcançamos o resultado desejado. Mas o recurso não deve ser acessado diretamente do navegador. É a aplicação cliente que deve acessar esse recurso e apresentar os dados no navegador de uma forma mais agradável visualmente. Portanto, o usuário final do fórum da Alura acessa a aplicação cliente via navegador que, por sua vez, acessa nossa API para obter o recurso a ser apresentado. O fluxo é algo parecido com a figura abaixo:



Figura 4.2: Spring Initializr

A aplicação cliente já foi desenvolvida e será executada localmente na porta 3000. Portanto, o usuário final do site da Alura deverá acessar a aplicação pelo endereço `http://localhost:3000`. A aplicação cliente já é capaz de acessar nosso recurso pelo endereço `http://localhost:8080/api/topics` e apresentar os tópicos.

Na pasta do curso está disponível a aplicação cliente. Copie o arquivo `fj27-alura-forum-client.zip` da pasta `/caelum/cursos/27` para o Desktop e extraia o seu conteúdo. Clique com o botão direito do mouse na pasta `fj27-alura-forum-client/` e selecione `Open in Terminal`. No terminal, digite o comando `git checkout 01DandoInicioANossaAPI`, em seguida digite o comando `npm start` e veja que a aplicação abrirá em seu navegador. Caso você esteja fazendo em sua casa, acesse este [link](#) e clone o repositório.

Como dito anteriormente, a aplicação foi desenvolvida utilizando React. O React é uma biblioteca Javascript desenvolvida pelo Facebook e sua principal função é renderizar componentes. Essa função específica da biblioteca permite desenvolver *views* que são fáceis de estender e manter. A documentação é bastante rica e pode ser acessada pelo [site da ferramenta](#). Os desenvolvedores do React fizeram este [post](#) explicando o porquê criaram a ferramenta e comentam suas vantagens.

Para saber mais

Neste curso não entraremos em detalhes de como a aplicação cliente do Fórum Alura funciona. Se você tem interesse em saber mais sobre sua implementação e funcionamento, recomendamos os cursos da [Formação Front-end](#) e o curso [JS-46 - Curso React e Redux para construção de Web Apps](#) da Caelum.

A aplicação é carregada, mas uma janela *pop-up* apresenta a seguinte mensagem "localhost:3000 says Failed to fetch" :

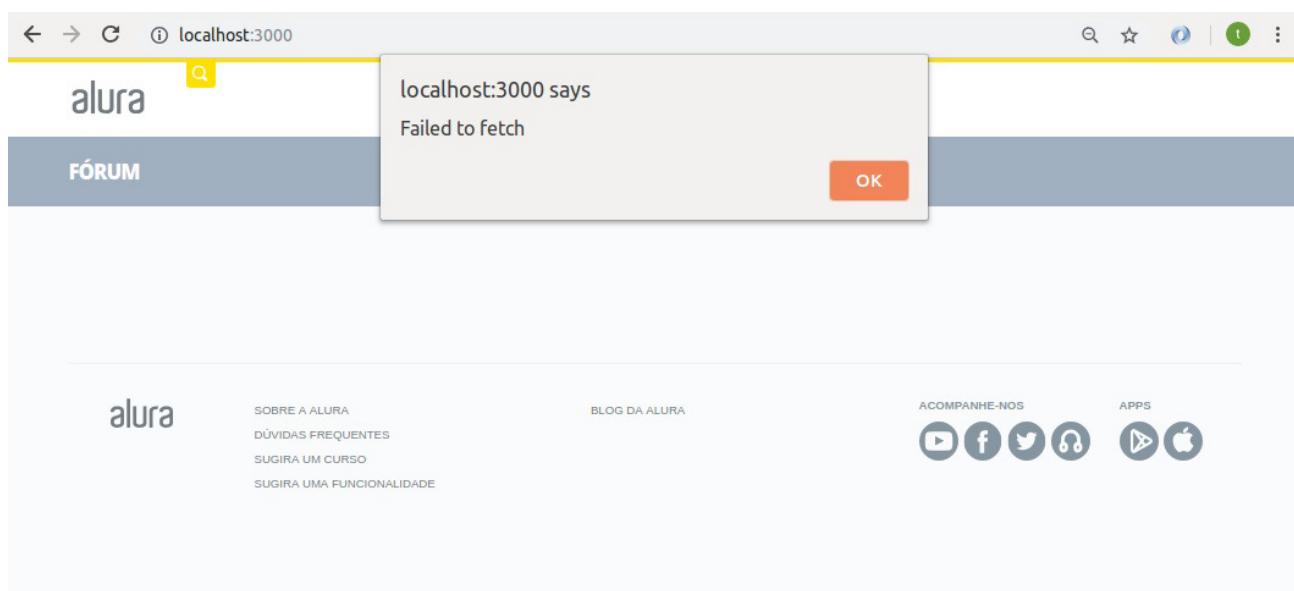


Figura 4.3: Fórum Alura Client

Clique com o botão direito na página, selecione **Inspect Element** e acesse o **Console** do navegador, que abrirá o terminal do JavaScript. Veja que uma mensagem de erro aparece:

```
✖ Access to fetch at 'http://localhost:8080/api/topics' from origin 'http://localhost:3000' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.
```

Figura 4.4: Cors Error

A mensagem diz que o recurso `http://localhost:8080/api/topics` foi bloqueado para `http://localhost:3000` e que nenhum cabeçalho com `Access-Control-Allow-Origin` foi encontrado no recurso requisitado. Um navegador compatível com a política de CORS (*Cross-Origin Resource Sharing*) tentará fazer a requisição para o serviço da nossa API enviando a origem da requisição, ou seja, `http://localhost:3000`. A API pode responder com outra informação de `Access-Control-Allow-Origin` no cabeçalho da resposta, indicando quais origens (sites) são permitidas para acessar o recurso, por exemplo:

```
Access-Control-Allow-Origin: http://localhost:3000
```

Cross-Origin Resource Sharing (CORS), ou Compartilhamento de Recursos de Origem Cruzada, é um mecanismo que permite que recursos restritos em uma página web sejam solicitados de outro domínio e define uma maneira na qual um navegador e um servidor podem interagir para determinar se é ou não seguro permitir solicitação de origem cruzada. O padrão CORS descreve novos cabeçalhos HTTP que fornecem aos navegadores e servidores uma maneira de solicitar URLs remotas somente quando eles tem permissão.

Portanto, nossa API precisa liberar o acesso para `http://localhost:3000`. Acrescentar essa informação para cada recurso que nossa aplicação vier a ter será uma tarefa muito repetitiva. Vamos pedir para o Spring incluir esta informação no cabeçalho de todas as respostas solicitadas pelo cliente. Para isso, criamos uma classe de configuração que chamaremos de `CorsConfiguration`, anotada com `@Configuration`. Essa classe deve implementar a interface `WebMvcConfigurer`, que possui uma série de *default methods* que podemos implementar para customizar uma configuração. Um deles é o `addCorsMappings()`, que recebe como parâmetro um `CorsRegistry`, que auxilia no registro da configuração global da política de CORS:

```
@Configuration
public class CorsConfiguration implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        }
}
```

Por meio do objeto `CorsRegistry`, configuramos para que todas as URLs que iniciem com `/api/` sejam liberadas para a origem `http://localhost:3000`:

```
@Configuration
public class CorsConfiguration implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**").allowedOrigins("http://localhost:3000");
    }
}
```

Além disso, precisamos avisar o Spring quais métodos do protocolo HTTP serão permitidos pelas

requisições do cliente. No caso, liberaremos para todos os suportados pelo protocolo:

```
@Configuration  
public class CorsConfiguration implements WebMvcConfigurer {  
  
    @Override  
    public void addCorsMappings(CorsRegistry registry) {  
        registry.addMapping("/api/**").allowedOrigins("http://localhost:3000").allowedMethods("GET",  
        "POST", "PUT",  
        "DELETE", "OPTIONS", "HEAD", "TRACE", "CONNECT");  
    }  
}
```

A anotação `@Configuration` do Spring é usada para anotar classes de configuração como nossa `CorsConfiguration`. A anotação indica que a classe declara um ou mais métodos que serão processados e gerenciados pelo Spring.

Salve e rode novamente a aplicação. Em seguida, acesse novamente o endereço `http://localhost:3000` pelo navegador e veja que agora o acesso é liberado e a aplicação cliente é capaz de renderizar a lista que a API está disponibilizando neste momento:

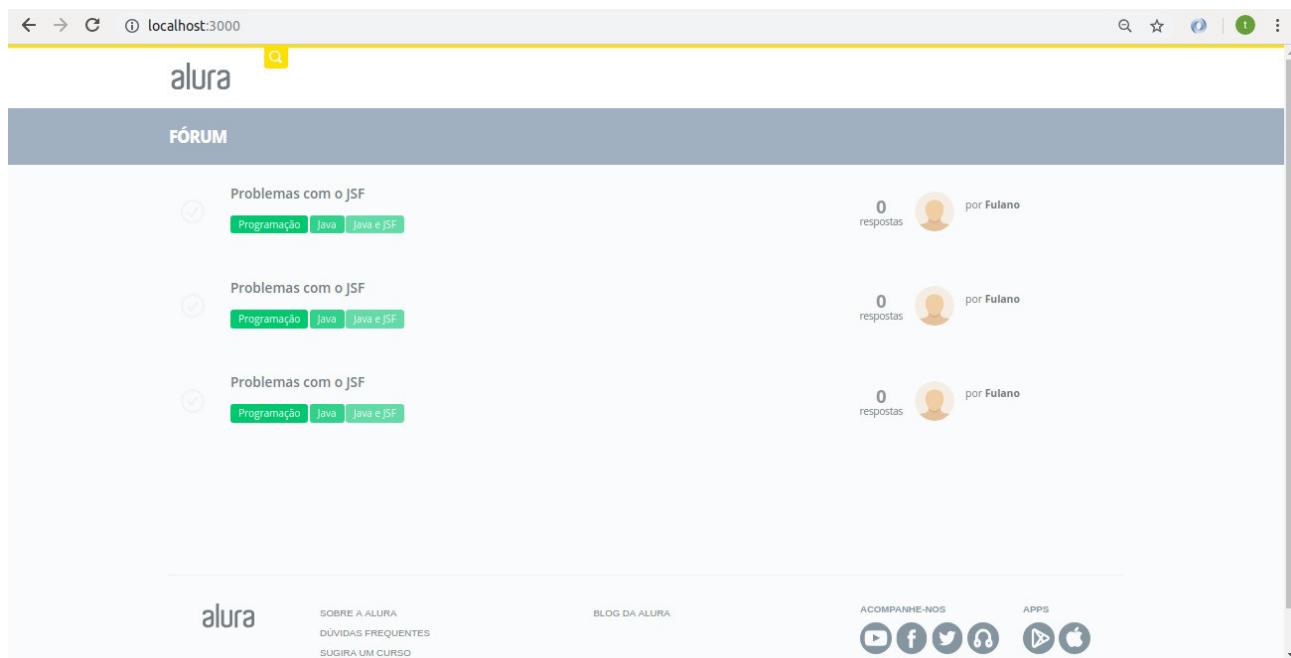


Figura 4.5: Fórum Alura Client

4.7 DEVTOOLS

O Spring Boot nos poupará grande parte do trabalho de configuração da aplicação, mas ainda temos que parar e iniciar a aplicação na mão a cada nova modificação. Para facilitar este processo, o Spring Boot incluiu um conjunto de ferramentas que torna o desenvolvimento da aplicação mais ágil e reinicia a aplicação automaticamente a cada nova mudança no código. Para incluir o *Dev Tools*, insira a seguinte dependência no arquivo `pom.xml`:

```
<dependencies>
    <!-- ... -->
    <!-- dependência do spring-boot-starter-test -->

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

Por padrão, a mudança de recursos em `/META-INF/maven`, `/META-INF/resources`, `/resources`, `/static`, `/public` ou `/templates` não aciona uma reinicialização. Caso você deseje personalizar e modificar este comportamento, use as propriedades `spring.devtools.restart.exclude` ou `spring.devtools.restart.additional-exclude` no arquivo `application.properties`.

Além disso, o *Dev Tools* desabilita as opções de *cache* por padrão, já que são contra producentes em tempo de desenvolvimento, e habilita as mensagens de *log* para web. Mais informações sobre o *Dev Tools* você encontra diretamente na [documentação](#).

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

4.8 EXERCÍCIOS: ESCREVENDO NOSSA PRIMEIRA LÓGICA

1. Para conseguirmos criar nosso primeiro *endpoint*, precisaremos contar com as classes que modelam as entidades do domínio. Mas, para que possamos focar na construção da API e nos detalhes do framework, já temos elas prontas na pasta do curso `caelum/cursos/27`.

Copie o arquivo `modelos.zip` da pasta do curso para o seu *Desktop*. Em seguida, no Eclipse, clique com o botão direito na pasta do projeto `forum`, selecione *Import > General > Archive File* e clique em *Next*. Na janela seguinte, clique em *Browse* e selecione o `zip` presente no seu *Desktop*. Agora clique em *Finish* e veja que as classes foram adicionadas juntamente com o pacote `br.com.alura.forum.model`.

2. Crie a classe `TopicController` no pacote `br.com.alura.forum.controller` e nela implemente o método para listar tópicos. Você pode criar instâncias simples de `Topic` nesse primeiro momento.

```
@Controller
public class TopicController {

    @ResponseBody
    @RequestMapping("/api/topics")
    public List<Topic> listTopics() {

        Category subcategory = new Category("Java", new Category("Programação"));
        Course course = new Course("Java e JSF", subcategory);
        Topic topic = new Topic("Problemas com o JSF",
            "Erro ao fazer conversão da data",
            new User("Fulano", "fulano@gmail.com", "123456"), course);

        List<Topic> topics = Arrays.asList(topic, topic, topic);
        return topics;
    }
}
```

Acesse o endpoint `http://localhost:8080/api/topics` no seu navegador e veja o resultado.

3. Agora teste a integração com a aplicação front-end:

- Copie e pasta `fj27-alura-forum-client.zip` da pasta `/caelum/cursos/27` para o *Desktop*, e extraia seu conteúdo.
- Clique com o botão direito do mouse na pasta `fj27-alura-forum-client/` e selecione *Open in Terminal*.
- No terminal digite o comando `git checkout 01DandoInicioANossaAPI`.
- Ainda no terminal, digite o comando `npm start` e veja que a aplicação abrirá em seu navegador.

Perceba que a lista de tópicos mais recentes ainda não está sendo apresentada.

4. Crie a classe `CorsConfiguration`, no pacote `br.com.alura.forum.configuration`:

```
@Configuration
public class CorsConfiguration implements WebMvcConfigurer {

}
```

5. Nesta classe, sobrescreva o método `addCorsMappings(CorsRegistry registry)` adicionando as configurações para liberar o acesso apenas para nossa aplicação cliente:

```
@Configuration
public class CorsConfiguration implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
            .allowedOrigins("http://localhost:3000")
            .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS",
```

```

        "HEAD", "TRACE", "CONNECT");
    }
}

```

Teste novamente recarregando a página da aplicação cliente (`http://localhost:3000/`). Perceba que ainda não foi apresentada a lista de tópicos.

Isso ocorre porque não recarregamos a API com as alterações do código. Seria necessário então encerrar o processo que está rodando no Eclipse, e rodar novamente.

6. Para facilitar o trabalho de atualização da API podemos contar com o *deploy* instantâneo provido pelo Spring Boot DevTools.

Adicione a dependência do projeto ao arquivo `pom.xml` :

```

<dependencies>
    <!-- ... -->
    <!-- dependência do spring-boot-starter-test -->

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
    </dependency>
</dependencies>

```

Rode a API e perceba que a partir de qualquer mudança no código, o Dev Tools automaticamente atualiza a aplicação rodando.

7. Teste novamente a aplicação cliente recarregando a página (`http://localhost:3000/`).

Agora, perceba que a lista apresentada contém informações incompletas. O problema é que o JSON retornado pela nossa API não está de acordo com as exigências da aplicação cliente.

O cliente espera uma representação do objeto (JSON) de tópico seguindo a especificação abaixo:

```
{
    "id": 33,
    "shortDescription": "Breve título da dúvida",
    "secondsSinceLastUpdate": 30, // segundos desde a última atualização
    "ownerName": "Fulano da Silva", // nome do usuário que publicou o tópico
    "courseName": "Java e JSF", // nome do curso ao qual o tópico foi registrado
    "subcategoryName": "Java", // subcategoria do curso
    "categoryName": "Programação", // categoria do curso
    "numberOfResponses": 0, // numero de respostas atualmente no tópico
    "solved": false // esta solucionada ou não
}
```

8. Crie a classe `TopicBriefOutputDto`, no pacote `br.com.alura.forum.controller.dto.output` com os atributos exigidos pelo cliente.

```

public class TopicBriefOutputDto {

    private Long id;
    private String shortDescription;
    private long secondsSinceLastUpdate;
}

```

```

    private String ownerName;
    private String courseName;
    private String subcategoryName;
    private String categoryName;
    private int numberOfResponses;
    private boolean solved;

}

```

9. Agora crie um construtor inicializando as variáveis a partir das informações de um `Topic`, e também os *getters* de cada atributo.

```

public class TopicBriefOutputDto {

    // atributos omitidos

    public TopicBriefOutputDto(Topic topic) {
        this.id = topic.getId();
        this.shortDescription = topic.getShortDescription();
        this.secondsSinceLastUpdate = getSecondsSince(topic.getLastUpdate());
        this.ownerName = topic.getOwner().getName();
        this.courseName = topic.getCourse().getName();
        this.subcategoryName = topic.getCourse().getSubcategory().getName();
        this.categoryName = topic.getCourse().getCategoryName();
        this.numberOfResponses = topic.getNumberOfAnswers();
        this.solved = TopicStatus.SOLVED.equals(topic.getStatus());
    }

    private long getSecondsSince(Instant lastUpdate) {
        return Duration.between(lastUpdate, Instant.now())
            .get(ChronoUnit.SECONDS);
    }

    // getters omitidos
}

```

10. Adicione também o método que encapsula os detalhes da construção de uma lista de *DTOs* na classe `TopicBriefOutputDto`.

```

public class TopicBriefOutputDto {

    // atributos, construtor e getters omitidos

    public static List<TopicBriefOutputDto> listFromTopics(List<Topic> topics) {
        return topics.stream()
            .map(TopicBriefOutputDto::new)
            .collect(Collectors.toList());
    }
}

```

11. Agora altere o código da classe `TopicController` para retornar a lista de *DTOs*.

```

@RestController
public class TopicController {

    @GetMapping(value = "/api/topics", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<TopicBriefOutputDto> listTopics() {

        Category subcategory = new Category("Java", new Category("Programação"));
        Course course = new Course("Java e JSF", subcategory);

```

```
Topic topic = new Topic("Problemas com o JSF",
    "Erro ao fazer conversão da data",
    new User("Fulano", "fulano@gmail.com", "123456"), course);

List<Topic> topics = Arrays.asList(topic, topic, topic);
return TopicBriefOutputDto.listFromTopics(topics);
}
```

Não esqueça de usar as novas *Annotations* `@RestController` e `@GetMapping` para facilitar a escrita do código.

- | 2. Teste novamente atualizando a aplicação cliente e verifique se tudo funciona como deveria.

ACESSANDO DADOS COM SPRING DATA JPA

5.1 ELIMINANDO O CÓDIGO REPETITIVO

Até o momento, estamos retornando tópicos salvos numa lista na memória da aplicação. Normalmente, os dados reais ficam salvos em algum lugar onde eles existam mesmo quando a aplicação não estiver sendo executada. O mais comum é usar um sistema de gerenciamento de banco de dados relacional, como o MySQL ou o PostgreSQL.

Quando se trata de acessar um banco de dados relacional no Java, normalmente acabamos utilizando alguma ferramenta que faz mapeamento do mundo relacional para o mundo da orientação a objetos. O Java EE possui uma especificação que trata do assunto, chamada JPA. A implementação da JPA mais usada no mercado é o Hibernate. O Spring se integra muito bem com a JPA, já que é uma especificação consolidada e muito utilizada.

Se você já teve que configurar sua aplicação para usar JPA, deve saber que normalmente é uma tarefa repetitiva e um tanto verbose. Normalmente envolve [configurar o arquivo persistence.xml](#). Caso esteja usando um servidor de aplicação, como o JBoss ou o WebSphere, é necessário configurar o `datasource`. Já vimos como o Spring nos ajuda neste momento da configuração: basta adicionar os valores no `application.properties` e a autoconfiguração se encarrega de ler os dados.

Mas, após configurar a conexão com o banco, precisamos fazer as operações, como salvar e buscar um registro pelo `id`. Isto é feito por meio do `EntityManager`. Você provavelmente já criou uma classe como esta:

```
@Stateless
public class TopicDao {

    @PersistenceContext
    private EntityManager em;

    public void adiciona(Topic topic) { em.persist(topic); }

    public Topic findById(Long id) {
        return em.find(Topic.class, id);
    }

    public List<Topic> findAll() {
        return em.createQuery("select t from Topic t", Topic.class).getResultList();
    }
}
```

```

    }

    public void delete (Topic topic) { em.remove(topic); }
}

```

O problema é que, para as outras entidades do sistema, o código seguiria bem parecido, mudando apenas o tipo do objeto com o qual você está lidando (Answer, User, Course). Algumas estratégias são utilizadas para evitar a repetição do código trivial nas operações do banco de dados, como o [DAO genérico](#).

O Spring possui um projeto cujo um dos objetivos é nos ajudar a reduzir a quantidade desse código repetitivo e quase igual ([boilerplate](#)) que escrevemos na parte de acesso aos dados. É o [Spring Data](#). Existe uma série de subprojetos para lidar com os mais diversos tipos de tecnologias de acesso a dados, como bancos relacionais e não-relacionais. Um deles se chama justamente [Spring Data JPA](#).

Para usar o Spring Data JPA, basta adicionar a seguinte dependência dentro da tag `dependencies` do `pom.xml`:

```

<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
</dependency>

```

Dentro do Spring Data JPA existe uma interface com alguns métodos comuns a todas as operações de CRUD:

```

public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    <S extends T> S save(S entity);

    Optional<T> findById(ID primaryKey);

    Iterable<T> findAll();

    long count();

    void delete(T entity);

    void deleteAll();

    // ... mais algumas operações aqui
}

```

Perceba que `CrudRepository` é uma interface genérica. Então vamos ter que definir a nossa com o tipo específico. O primeiro parâmetro do `generics` é o tipo da nossa entidade. O segundo é o tipo da chave primária que foi definida:

```

public interface TopicRepository extends CrudRepository<Topic, Long> {
}

```

Como fizemos uma herança, agora o `TopicRepository` tem todos os métodos definidos na

interface `CrudRepository`. Feito isso, podemos pedir para o Spring injetar alguma implementação da nossa interface:

```
public class TopicController {  
  
    @Autowired  
    private TopicRepository topicRepository;  
  
    // ...  
}
```

Mas qual implementação de `TopicRepository` o Spring irá injetar? Se tivermos que criar uma classe, seremos obrigados a definir todos os métodos das interfaces, e cairemos no problema de ficar escrevendo o código muito parecido.

O ponto aqui é que não precisamos fornecer a implementação. O Spring fará isso em tempo de execução. Assim, a única coisa que temos que fazer é definir nossa interface, como já fizemos.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

5.2 LIMITANDO OS COMPORTAMENTOS DA INTERFACE

Perceba que `CrudRepository` possui métodos como `delete()` e `deleteAll()`. Esses métodos ficam disponíveis para serem chamados em qualquer ponto da nossa aplicação, mesmo se não quisermos deletar valores depois de inseridos. Para resolver este problema, o Spring tem uma interface central da qual todas as outras herdam, chamada `Repository`. Podemos herdar diretamente dela:

```
public interface TopicRepository extends Repository<Topic, Long> { }
```

Porém, `Repository` é uma interface de marcação. Ela não possui nenhuma definição interna:

```
public interface Repository<T, ID> { }
```

Isso significa que agora não temos mais comportamentos às vezes indesejados, como o `delete()`, mas também não temos mais nenhuma operação do CRUD. E se tentarmos definir um método igual ao

que herdamos de `CrudRepository`, como o `save()`?

```
public interface TopicRepository extends Repository<Topic, Long> {  
    void save(Topic topic);  
}
```

É isso, basicamente! O Spring verá que só temos o método `save()` e irá gerar uma implementação apenas com este método. Assim definimos exatamente o que queremos.

5.3 DEFININDO NOSSAS PRÓPRIAS QUERIES

Existem ainda os casos onde queremos fazer uma *query* customizada. Em breve você verá mais sobre isso. Mas vamos pensar no caso mais básico: trazer todos os dados do banco em uma lista de objetos. Precisamos fazer a seguinte JPQL:

```
"select t from Topic t"
```

O Spring suporta a execução de JPQL por meio da annotation `@Query`. Ele vai se virar pra pegar o resultado e montar uma lista pra gente. Muito interessante, não?

```
public interface TopicRepository extends Repository<Topic, Long> {  
    @Query("select t from Topic t")  
    List<Topic> list();  
}
```

Como buscar todos os dados é uma operação padrão de um CRUD básico, a `CrudRepository` já tem a definição de um método `findAll()`. Neste caso, podemos também ter um método com esse nome, fazendo com que a anotação `@Query` seja desnecessária e o Spring consiga buscar todos os objetos apenas olhando para a assinatura do nosso método:

```
public interface TopicRepository extends Repository<Topic, Long> {  
    List<Topic> findAll();  
}
```

5.4 PARA SABER MAIS: QUERIES A PARTIR DO NOME DOS MÉTODOS

Imagine que, além de trazer todos os tópicos, queremos filtrar os tópicos de um determinado curso. Como essa é uma operação que foge das operações mais básicas do CRUD, podemos escrever uma JPQL por meio da anotação `@Query` do Spring Data.

Como é necessário o nome do curso, podemos simplesmente receber uma referência para um curso como parâmetro do nosso método. A anotação `@Param` indica o nome que queremos usar para acessar o parâmetro na nossa *query*:

```
public interface TopicRepository extends Repository<Topic, Long> {
```

```

// restante do código

@Query("select t from Topic t where t.course = :course")
List<Topic> findByCourse(@Param("course") Course course);
}

```

Mas já vimos que, no caso de buscar todos os tópicos, o Spring consegue se virar já que o método se chama `findAll()`. Podemos levar isso um pouco além: `findByCourse()` já é um nome de método que indica exatamente qual nosso critério de busca. Se tentarmos tirar a anotação `@Query`, temos o mesmo resultado!

```

public interface TopicRepository extends Repository<Topic, Long> {

    // restante do código

    List<Topic> findByCourse(Course course);
}

```

O Spring consegue gerar algumas *queries* a partir do nome dos métodos definidos na nossa interface. Perceba que precisamos seguir algum padrão, como por exemplo, começar com *find*. Podemos ir mais além e buscar pelo nome do curso, por exemplo:

```

public interface TopicRepository extends Repository<Topic, Long> {

    // restante do código

    List<Topic> findByCourseName(String courseName);
}

```

Perceba que conseguimos navegar dentro dos atributos das nossas classes: a classe `Topic` tem um `course`, que, por sua vez, tem um `name`. O Spring irá levar tudo isso em consideração na hora de montar a *query*.

Após o critério de busca, podemos definir um critério de ordenação. Por exemplo, trazer os tópicos ordenados pela data de criação, do mais recente para o mais antigo:

```

public interface TopicRepository extends Repository<Topic, Long> {

    // restante do código

    List<Topic> findByCourseNameOrderByCreationInstantDesc(String courseName);
}

```

Você pode ver um pouco mais sobre o que é possível fazer com os nomes dos métodos na [documentação do Spring Data JPA](#). Perceba que, para *queries* mais complexas, o nome do método vai ficar bem grande, então pode ser mais sensato nomear como achar conveniente e escrever a *query JPQL* usando a anotação `@Query`.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

5.5 EXERCÍCIOS: EXIBINDO LISTA DE DÚVIDAS

1. O método `listTopics()` da classe `TopicController` retorna uma lista "irreal" da aplicação - uma lista que criamos apenas para testar o funcionamento do nosso *controller*. O que realmente queremos é mostrar a lista de tópicos que estão no banco de dados da aplicação. Vamos começar criando nossa interface `TopicRepository` , no pacote `br.com.alura.forum.repository` , herdando de `Repository` e passando os tipos do modelo e da *primary key* que queremos persistir no banco:

```
public interface TopicRepository extends Repository<Topic, Long> {  
}
```

2. Crie o método `list()` , que retornará a lista de tópicos presentes no banco de dados, e anote-o com `@Query` informando a query JPQL :

```
public interface TopicRepository extends Repository<Topic, Long>{  
  
    @Query("select t from Topic t")  
    List<Topic> list();  
}
```

O *import* correto da classe `List` é do pacote `java.util` . Já a classe `Repository` e a anotação `@Query` são do pacote do Spring Data `org.springframework.data` .

3. Modifique a classe `TopicController` para retornar a lista de tópicos persistida no banco de dados. Crie um atributo do tipo `TopicRepository` anotado com `@Autowired` para que o Spring injete o repositório. Dentro do método `listTopics()` chame o método `list()` do repositório para trazer os dados do banco:

```
@RestController  
public class TopicController {
```

```

    @Autowired
    private TopicRepository topicRepository;

    @GetMapping(value = "/api/topics", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<TopicBriefOutputDto> listTopics() {

        List<Topic> topics = topicRepository.list();
        return TopicBriefOutputDto.listFromTopics(topics);
    }
}

```

4. Agora, precisaremos de alguns dados na base para que possamos testar o funcionamento. Vá até a pasta do curso e copie o arquivo `forum.sql` para a pasta `Home` (`/home/springXXXX`). Este arquivo contém a estrutura de tabelas já com alguns dados sobre tópicos, respostas e usuários.

- Abra o terminal, digite o seguinte comando para importar a base de dados:

```
$ mysql -u root fj27_spring < forum.sql
```

- Acesse o banco e a *database* `fj27_spring`:

```
$ mysql -u root
mysql> use fj27_spring;
```

- Faça um `select` na tabela `topics` para testar se a importação funcionou:

```
mysql> select * from topic;
```

5. Rode a aplicação e teste recarregando a página da aplicação cliente (`http://localhost:3000/`). Veja que agora a lista de tópicos mostra os dados do banco de dados.

6. Para usar as facilidades do `Spring Data`, acrescente o método `findAll()`, que retorna uma lista de tópicos, na interface `TopicRepository`:

```

public interface TopicRepository extends Repository<Topic, Long>{

    @Query("select t from Topic t")
    List<Topic> list();

    List<Topic> findAll();
}

```

7. Na classe `TopicController`, altere o método de listagem para chamar `findAll()` ao invés do `list()` do `TopicRepository`:

```

@RestController
public class TopicController {

    @Autowired
    private TopicRepository topicRepository;

    @GetMapping(value = "/api/topics", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<TopicBriefOutputDto> listTopics() {

        //invocando findAll() ao invés de list()
    }
}

```

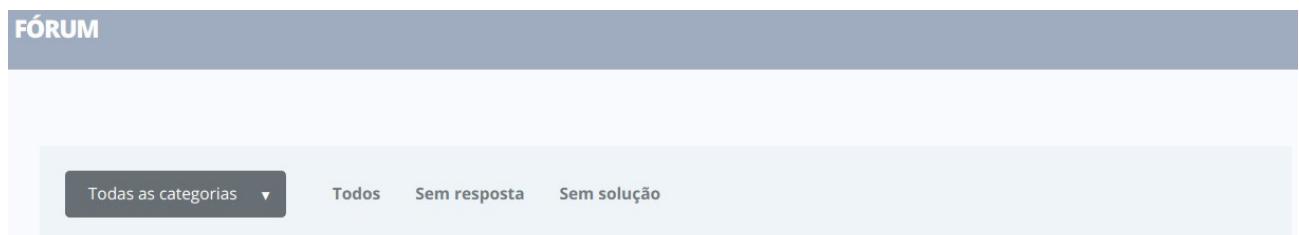
```
        List<Topic> topics = topicRepository.findAll();
        return TopicBriefOutputDto.listFromTopics(topics);
    }
}
```

8. Rode novamente a aplicação. Teste recarregando a página da aplicação cliente (<http://localhost:3000/>) e veja que tudo continua funcionando como antes.

MAIS ALÉM COM SPRING DATA

6.1 FILTRANDO AS DÚVIDAS

Pensando em facilitar a busca de quem usa o fórum, precisamos filtrar os tópicos baseados em dois critérios: a categoria e o status.

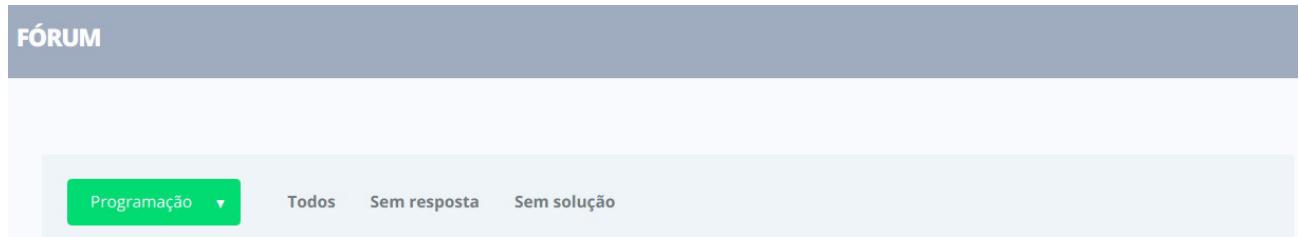


Primeiro vamos nos preparar para receber os dados selecionados. Para isso, será criada a classe `TopicSearchInputDto` no pacote `br.com.alura.forum.controller.dto.input`.

```
public class TopicSearchInputDto {  
  
    private TopicStatus status;  
    private String categoryName;  
  
    // getters e setters  
}
```

Perceba que, no caso de o usuário não selecionar nenhum dos critérios, precisamos buscar todos os tópicos. Isso também fará com que os dois atributos do nosso `TopicSearchInputDto` fiquem nulos.

Existe também a chance de que a busca seja feita baseada em apenas um dos critérios, fazendo com que um ou outro dos valores do DTO sejam nulos.



FÓRUM

Todas as categorias ▾ Todos Sem resposta Sem solução

Por fim, o usuário pode selecionar os dois critérios:

FÓRUM

Programação ▾ Todos Sem resposta Sem solução

Quantas possibilidades! É possível criar, dentro da interface `TopicRepository`, um método para cada um dos cenários. Algo como:

```
public interface TopicRepository extends Repository<Topic, Long> {  
  
    List<Topic> findAll();  
  
    List<Topic> findByStatus(TopicStatus topicStatus);  
  
    @Query("select t from Topic t " +  
           "where t.course.subcategory.category.name = :categoryName")  
    List<Topic> findByCategoryName(@Param("categoryName") String categoryName);  
  
    @Query("select t from Topic t where t.status = :status " +  
           "and t.course.subcategory.category.name = :categoryName")  
    List<Topic> findByStatusAndCategoryName(@Param("status") TopicStatus topicStatus,  
                                            @Param("categoryName") String categoryName);  
}
```

O próximo passo é descobrir quando chamar cada um dos métodos. Em algum ponto da aplicação, teríamos alguns `ifs` checando se os valores recebidos são nulos ou não e decidindo qual dos métodos do repositório devemos chamar.

Se adicionarmos mais critérios na busca, o nosso repositório terá que ganhar mais métodos para suportá-los, e aumentarão nossas verificações para descobrir qual método chamar.

Perceba que, o que muda de um método pro outro é apenas nosso critério de busca. Precisamos encontrar uma forma de construir nossa `query` dinamicamente. Algo comum de ser utilizado no JPA nestes casos é a [API de criteria](#).

Pensando na Criteria, temos dois `Predicate`s aqui: o que verifica o status e o que verifica a categoria. O código ficaria assim:

```

CriteriaBuilder builder = manager.getCriteriaBuilder();
CriteriaQuery<Topic> query = builder.createQuery(Topic.class);
Root<Topic> root = query.from(Topic.class);

List<Predicate> predicates = new ArrayList<>();

if (status != null) {
    Predicate equalStatus = builder.equal(root.get("status"), topicStatus);
    predicates.add(equalStatus);
}

if (categoryName != null) {
    Predicate equalCategoryName =
        builder.equal(root.get("course").get("subcategory").get("category").get("name"),
categoryName);
    predicates.add(equalCategoryName);
}

Predicate statusAndCategoryName = builder.and(predicates.toArray(new Predicate[]{}));

return manager.createQuery(query.where(statusAndCategoryName))
    .getResultList();

```

As três primeiras linhas são apenas *boilerplate* referentes à JPA para acessarmos os objetos necessários para construir nossa *query*. A última linha também vai ser sempre igual, onde a *query* é executada e recuperamos a lista de *Predicates*. O que realmente importa e muda neste cenário são os predicados que foram construídos dentro de cada um dos *ifs*. Na penúltima linha, montamos o predíco final.

O Spring Data fornece uma forma de pular toda essa burocracia e focarmos na construção do *Predicate*. Isso é feito por meio da interface *Specification*. Esta interface possui um método abstrato com a seguinte assinatura:

```
Predicate toPredicate(Root<T> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder);
```

Já temos os objetos necessários para que a *criteria query* seja montada: ganhamos o *Root*, a *CriteriaQuery* e o *CriteriaBuilder*! Precisamos apenas criar nossa implementação da *Specification* que irá retornar o *Predicate* baseado nos dados que chegam na requisição. Como esses dados estão no *TopicSearchInputDto*, vamos criar o método nesta classe:

```

public class TopicSearchInputDto {
    private TopicStatus status;
    private String categoryName;

    // getters e setters

    public Specification<Topic> build() {
        // retornar algo aqui...
    }
}

```

Como a *Specification* é uma interface que possui apenas um método abstrato, o mais comum nestes cenários é criarmos uma classe anônima. A partir do Java 8, podemos também criar uma

expressão lambda. Como já temos os objetos do tipo `Root`, `CriteriaQuery` e `CriteriaBuilder`, podemos construir nosso `Predicate`, similar ao que fizemos anteriormente:

```
public class TopicSearchInputDto {  
  
    // atributos  
  
    public Specification<Topic> build() {  
        return (root, query, criteriaBuilder) -> {  
  
            ArrayList<Predicate> predicates = new ArrayList<>();  
  
            if (status != null) {  
                predicates.add(criteriaBuilder.equal(root.get("status"), status));  
            }  
  
            if (categoryName != null) {  
                Path<String> categoryNamePath = root.get("course")  
                    .get("subcategory").get("category").get("name");  
                predicates.add(criteriaBuilder.equal(categoryNamePath, categoryName));  
            }  
  
            return criteriaBuilder.and(predicates.toArray(new Predicate[0]));  
        };  
    }  
  
    // getters e setters  
}
```

Utilizando classe anônima, acabamos com o seguinte código:

```
public Specification<Topic> build() {  
    return new Specification<Topic>() {  
  
        @Override  
        public Predicate toPredicate(Root root, CriteriaQuery criteriaQuery,  
            CriteriaBuilder criteriaBuilder) {  
  
            ArrayList<Predicate> predicates = new ArrayList<>();  
  
            if(status != null) {  
                predicates.add(criteriaBuilder.equal(root.get("status"), status));  
            }  
  
            if(categoryName != null) {  
                Path<String> categoryNamePath = root.get("course")  
                    .get("subcategory").get("category").get("name");  
                predicates.add(criteriaBuilder.equal(categoryNamePath, categoryName));  
            }  
  
            return criteriaBuilder.and(predicates.toArray(new Predicate[0]));  
        }  
    };  
}
```

Agora é necessário alguém que receba uma `Specification` e que internamente chamará o método

`toPredicate()`. O Spring Data já cuidou disso pra gente! Para isso, o `TopicRepository` precisará herdar da interface `JpaSpecificationExecutor`.

```
public interface TopicRepository extends Repository<Topic, Long>, JpaSpecificationExecutor<Topic> {  
    @Query("select t from Topic t")  
    List<Topic> list();  
}
```

Com isso ganhamos o método `findAll(Specification<T> spec)`. Ou seja: um método que, basta passarmos um `Specification` qualquer, irá retornar os dados baseados no nosso `Predicate`. Podemos reaproveitar o método da forma que for mais interessante. Basta chamar o método no `TopicController`, recebendo também um `TopicSearchInputDto`:

```
@GetMapping("/api/topics")  
public List<TopicBriefOutputDto> listTopics(TopicSearchInputDto topicSearch) {  
  
    Specification<Topic> specification = topicSearch.build();  
    List<Topic> topics = topicRepository.findAll(specification);  
    return TopicBriefOutputDto.listFromTopics(topics);  
}
```

Agora sim! Conseguimos trazer todos os tópicos com as restrições que definimos.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

6.2 EXERCÍCIOS: TRABALHANDO COM FILTROS OPCIONAIS DE BUSCA

- Para facilitar a criação da query dinâmica em função dos filtros opcionais de busca, também vamos contar com o apoio do Spring Data JPA. Na interface `TopicRepository`, adicione como *super interface* a interface `JpaSpecificationExecutor`.

```
public interface TopicRepository extends  
    Repository<Topic, Long>, JpaSpecificationExecutor<Topic> {  
}
```

Dessa forma, quando gerada a implementação do nosso repository, o Spring Data disponibilizará implementações para métodos que recebem uma Specification com os detalhes de busca. Os imports de Specification e JpaSpecificationExecutor são do pacote org.springframework.data .

2. Agora, faça com que seu *Controller* invoque o método findAll que recebe uma Specification :

```
@RestController
public class TopicController {

    @Autowired
    private TopicRepository topicRepository;

    @GetMapping(value = "/api/topics", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<TopicBriefOutputDto> listTopics() {

        //invocando findAll(Specification spec)
        List<Topic> topics = topicRepository.findAll(topicSearchSpecification);
        return TopicBriefOutputDto.listFromTopics(topics);
    }
}
```

Neste momento o código não está compilando! Ainda não temos uma instância de Specification criada para que possamos repassar ao *Repository*. Fique tranquilo(a), já vamos cuidar disso...

3. Para isolar a construção da Specification em função dos filtros de busca utilizaremos também a estratégia do DTO. No pacote br.com.alura.forum.controller.dto.input , crie a classe TopicSearchInputDto , com os seguintes atributos e seus getters e setters:

```
public class TopicSearchInputDto {

    private TopicStatus status;
    private String categoryName;

    // getters & setters omitidos
}
```

4. Ainda nessa classe, crie o método que constrói a Specification de acordo com a presença dos filtros de busca. As interfaces Predicate e Path são do pacote javax.persistence.criteria .

```
public class TopicSearchInputDto {

    // atributos e seus getters & setters omitidos

    public Specification<Topic> build() {
        return (root, criteriaQuery, criteriaBuilder) -> {

            ArrayList<Predicate> predicates = new ArrayList<>();

            if(status != null) {
                predicates.add(criteriaBuilder.equal(root.get("status"), status));
            }

            if(categoryName != null) {
                Path<String> categoryNamePath = root.get("course")

```

```

        .get("subcategory").get("category").get("name");
        predicates.add(criteriaBuilder.equal(categoryNamePath, categoryName));
    }

    return criteriaBuilder.and(predicates.toArray(new Predicate[0]));
};

}
}

```

Fique à vontade para implementar de forma diferente caso a sintaxe das *Lambda Expressions* do Java 8 ainda não seja muito familiar. Você pode sem dúvida criar uma classe para definir a implementação de `Specification`, ou mesmo usar uma classe anônima para isso.

5. Agora que já temos nosso DTO de input devidamente implementado, nosso *Controller* pode utilizá-lo para facilitar o trabalho de transformação dos dados que vem da Web para prosseguir com sua utilização pelas classes de domínio. Atualize o código da classe `TopicController` para usar o DTO e nossa `Specification`.

```

@RestController
public class TopicController {

    @Autowired
    private TopicRepository topicRepository;

    @GetMapping(value = "/api/topics", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<TopicBriefOutputDto> listTopics(TopicSearchInputDto topicSearch) {

        Specification<Topic> topicSearchSpecification = topicSearch.build();
        List<Topic> topics = topicRepository.findAll(topicSearchSpecification);

        return TopicBriefOutputDto.listFromTopics(topics);
    }
}

```

6. Para que seja possível testar a aplicação cliente com esse novo recurso, no terminal, acesse a pasta do projeto da aplicação cliente `Desktop/fj27-alura-forum-client/`, e execute o comando `git checkout 06MaisAlemComSpringData1`, para atualizar seu estado.

Com o *client* e a *API* rodando com os novos recursos, teste novamente o funcionamento da app cliente recarregando a página (`http://localhost:3000/`) e veja que ao selecionar uma categoria ou status as buscas são devidamente filtradas.

6.3 PAGINANDO OS RESULTADOS

O fórum da Alura possui vários tópicos criados. Imagine o caos de carregar milhares de tópicos na página para o usuário. Para isso não ocorrer, é muito comum que as aplicações usem paginação, limitando o número de tópicos carregados por vez. No nosso fórum não será diferente, teremos uma paginação logo depois dos tópicos:

Pensando em uma JPQL dentro de uma classe DAO clássica, para realizar a paginação podemos receber o número da página e a quantidade de tópicos por página:

```
public List<Topic> findAll(int pageSize, int page) {  
    TypedQuery<Topic> query = manager.createQuery("select t from Topic t", Topic.class);  
    query.setFirstResult(page * pageSize);  
    query.setMaxResults(pageSize);  
    return query.getResultList();  
}
```

Mas, se mandarmos a lista dessa forma pro cliente React, como ele vai saber quantas páginas existem no total, por exemplo? Como fazer para exibir a quantidade de páginas para o usuário conseguir clicar e realizar a navegação? Além de enviar a lista, precisamos também preparar alguns *metadados*, que contém, juntamente com os tópicos, informações como a quantidade de páginas e a página em que estamos naquele momento.

Perceba que, na *query* exibida acima, o método do DAO precisa receber os dados da página que queremos buscar. Resumindo, temos uma série de coisas pra implementar sempre que precisarmos fazer nossa paginação. Como é algo comum dentro das aplicações, o Spring fornece duas abstrações para realizarmos a paginação. Uma delas é a [Pageable](#), para representar os dados.

Como `Pageable` é uma interface, a classe `PageRequest` é uma implementação desta interface. O método `of()` é responsável pela criação do nosso objeto, recebendo como parâmetros o número da página e a quantidade de elementos que esta página deve conter.

```
Pageable pageable = PageRequest.of(1, 10);
```

Perceba que apesar de ser útil para os casos em que desejarmos obter uma página específica, normalmente esses dados são fornecidos na requisição, quando o usuário clica em um botão com o número da página, por exemplo. Por este motivo, é possível simplesmente receber um objeto do tipo `Pageable` injetado pelo Spring no método `TopicController`, assim receberemos os dados que o *client* definir.

```
@GetMapping("/api/topics")  
public List<TopicBriefOutputDto> listTopics(TopicSearchInputDto topicSearch, Pageable pageable) {  
    // restante do código  
}
```

Para a injeção ser possível é necessário usar a anotação `@EnableSpringDataWebSupport` na classe com método `main`.

```
@EnableSpringDataWebSupport  
@SpringBootApplication  
public class ForumApplication {
```

```

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

```

Depois que os dados da `Pageable` forem preenchidos com os dados que o usuário enviar na requisição, podemos simplesmente passar o `Pageable` para uma sobrecarga do método `findAll()`.

```
topicRepository.findAll(specification, pageable);
```

Mas o que esse `findAll()` retorna? Uma lista com apenas alguns tópicos? Lembre-se que, além da sublista, precisamos de informações sobre a página onde estamos. O Spring contém uma segunda abstração, que representa a lista com os dados da página juntos: a `Page`. Este é nosso retorno então:

```
Page<Topic> topics = topicRepository.findAll(specification, pageable);
```

Lembre-se que não retornamos `Topic`s para o nosso cliente. O retorno é baseado no `TopicBriefOutputDto`. Portanto, é necessário converter o `Page<Topic>` para um `Page<TopicBriefOutputDto>`, o mesmo que foi feito com a nossa lista de tópicos. Podemos então criar uma sobrecarga do método `listFromTopics()` no `TopicBriefOutputDto`, que recebe um `Page`.

```

public class TopicBriefOutputDto {

    // atributos e métodos

    public static List<TopicBriefOutputDto> listFromTopics(List<Topic> topics) {
        // implementação
    }

    public static Page<TopicBriefOutputDto> listFromTopics(Page<Topic> topics) {
        return topics.map(TopicBriefOutputDto::new);
    }
}

```

A `Page` possui um método `map`, similar ao `map` do `Stream` presente no Java 8, que podemos utilizar para fazer uma transformação, neste caso de um objeto para outro. Para cada objeto do tipo `Topic` dentro da nossa `Page`, queremos criar um `TopicBriefOutputDto` passando este `Topic` como argumento no seu construtor. O lambda correspondente a esse *method reference* seria:

```
return topics.map(topic -> new TopicBriefOutputDto(topic));
```

Para acessar os dados paginados por meio da URL, é só passar os parâmetros `page`, para indicar a página, e `size`, para definir a quantidade:

<http://localhost:8080/api/topics?page=2&size=10>

Além dos tópicos em quantidade reduzida (que foram ocultados na imagem a seguir), temos os *metadados*. É possível saber, por exemplo, que existem 5 páginas no total. A partir disso, a aplicação cliente pode exibir a quantidade de páginas para o usuário selecionar qual gostaria de ver.

```

{
  + content: [...],
  - pageable: {
    - sort: {
      sorted: false,
      unsorted: true,
      empty: true
    },
    pageSize: 10,
    pageNumber: 2,
    offset: 20,
    paged: true,
    unpaged: false
  },
  totalPages: 5,
  totalElements: 48,
  last: false,
  first: false,
  size: 10,
  number: 2,
  - sort: {
    sorted: false,
    unsorted: true,
    empty: true
  },
  numberOfElements: 10,
  empty: false
}

```

6.4 ORDENANDO OS RESULTADOS

O filtro e a paginação estão funcionando lindamente! Tem só mais um detalhe: quando o usuário abre o fórum da Alura, ele deve ver as dúvidas mais recentes primeiro. No momento, as dúvidas estão vindo na ordem em que foram criadas, ou seja: as mais antigas primeiro.

Poderíamos resolver isto direto no repository. Mas, além do caso padrão, o usuário também poderia escolher outros critérios, como ordenar baseado na última atualização do tópico. Podemos querer ver primeiro os tópicos que foram atualizados recentemente.

Além da paginação, o Spring Data nos dá suporte a definirmos os critérios de ordenação de forma dinâmica. A própria `Pageable` já carrega este suporte. Assim não vamos precisar cravar na `query` uma ordenação específica.

Na URL, podemos passar, utilizando o parâmetro `sort`, qual propriedade queremos que seja utilizada na ordenação, junto com os dados de paginação. Todos os parâmetros serão preenchidos pelo Spring dentro do `Pageable`:

<http://localhost:8080/api/topics?page=2&size=10&sort=lastUpdate,desc>

Perceba que, separando por vírgula, também é possível informar se os dados devem ser ordenados de

forma ascendente (asc) ou descendente (desc).

Mas queremos ter um caso padrão. Se o cliente não passar nenhum critério na requisição, é necessário ordenar baseado no `creationInstant`, do mais recente para o mais antigo. Neste caso, o Spring possui a annotation `@PageableDefault`, onde podemos passar, junto do `Pageable`, informações padrão sobre a página e sobre a ordenação. Por exemplo:

```
@GetMapping(value = "/api/topics", produces = MediaType.APPLICATION_JSON_VALUE)
public Page<TopicBriefOutputDto> listTopics(TopicSearchInputDto topicSearch,
    @PageableDefault(sort="creationInstant", direction=Sort.Direction.DESC) Pageable pageRequest) {
    // código interno omitido
}
```

O `sort` serve pra indicarmos qual atributo do `Topic` queremos usar na ordenação e a `direction` indica se será `asc` ou `desc`.

E é isso! Agora sempre teremos nossos resultados ordenados por padrão da forma que desejamos. Veja a query gerada depois que usamos a `@PageableDefault`:

```
Hibernate:
select
    topic0_.id as id1_3_,
    topic0_.content as content2_3_,
    topic0_.course_id as course_i7_3_,
    topic0_.creation_instant as creation3_3_,
    topic0_.last_update as last_upd4_3_,
    topic0_.owner_id as owner_id8_3_,
    topic0_.short_description as short_de5_3_,
    topic0_.status as status6_3_
from
    topic topic0_
where
    l=1
order by
    topic0_.creation_instant desc limit ?, ?
```

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

6.5 EXERCÍCIOS: PAGINANDO RESULTADOS COM SPRING DATA

1. Para que possamos devolver as páginas para nossa aplicação cliente, adicione como parâmetro do método `listTopics()` um `Pageable` do pacote `org.springframework.data`.

```
@RestController
public class TopicController {

    // atributos omitidos

    @GetMapping(value = "/api/topics", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<TopicBriefOutputDto> listTopics(TopicSearchInputDto topicSearch,
                                                Pageable pageRequest) {

        // código interno omitido
    }
}
```

2. Adicione a annotation `@EnableSpringDataWebSupport` sobre a classe `ForumApplication`, para que o Spring monte o *page request* de acordo com os valores dos parâmetros de requisição `page` e `size`.

```
@EnableSpringDataWebSupport
@SpringBootApplication
public class ForumApplication {

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

3. Com o *page request* sendo injetado com os valores corretos, altere a implementação do método

`listTopics()` para que ele invoque o método `findAll()` do *Repository* que recebe, além da `Specification`, o `Pageable`.

```
@RestController
public class TopicController {

    // atributos omitido

    @GetMapping(value = "/api/topics", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<TopicBriefOutputDto> listTopics(TopicSearchInputDto topicSearch,
        Pageable pageRequest) {

        Specification<Topic> topicSearchSpecification = topicSearch.build();
        Page<Topic> topics = this.topicRepository.findAll(topicSearchSpecification,
            pageRequest);

        return TopicBriefOutputDto.listFromTopics(topics);
    }
}
```

Perceba que o método já habilita o retorno de um `Page` também do pacote `org.springframework.data`, um *wrapper* do Spring Data que representa uma sublista dos registros totais, trazendo além do conteúdo parcial de tópicos, informações úteis sobre a posição da página em relação à totalidade dos valores.

4. Com a alteração efetuada, temos no momento um erro de compilação. O método `listFromTopics()` encapsula o mapeamento de uma `List<Topic>` para uma `List<TopicBriefOutputDto>`. Crie então uma implementação de `listFromTopics()` que seja capaz de mapear uma `Page<Topic>` para uma `Page<TopicBriefOutputDto>`.

```
public class TopicBriefOutputDto {

    // código anterior omitido

    public static Page<TopicBriefOutputDto> listFromTopics(Page<Topic> topicPage) {
        return topicPage.map(TopicBriefOutputDto::new);
    }
}
```

5. Agora que temos no DTO o método capaz de gerenciar os retornos de conteúdo paginado, altere o retorno do método `listTopics()` do *Controller* para devolver a `Page` de `TopicBriefOutputDto`.

```
@RestController
public class TopicController {

    @Autowired
    private TopicRepository topicRepository;

    @GetMapping(value = "/api/topics", produces = MediaType.APPLICATION_JSON_VALUE)
    public Page<TopicBriefOutputDto> listTopics(TopicSearchInputDto topicSearch,
        Pageable pageRequest) {

        Specification<Topic> topicSearchSpecification = topicSearch.build();
        Page<Topic> topics = this.topicRepository.findAll(topicSearchSpecification,
```

```

        pageRequest);

    return TopicBriefOutputDto.listFromTopics(topics);
}
}

```

Aproveite para anotar o parâmetro `pageRequest` com `@PageableDefault` do pacote `org.springframework.data` para que o retorno seja ordenado pela data mais recente.

```

@GetMapping(value = "/api/topics", produces = MediaType.APPLICATION_JSON_VALUE)
public Page<TopicBriefOutputDto> listTopics(TopicSearchInputDto topicSearch,
                                              @PageableDefault(sort="creationInstant", direction=Sort.Direction.DESC) Pageable pageRequest)
{
    // código interno omitido
}

```

A classe `Sort` também deve ser importada do pacote `org.springframework.data`.

6. Para que seja possível testar a aplicação cliente com esse novo recurso, no terminal, acesse a pasta do projeto da aplicação cliente `Desktop/fj27-alura-forum-client/`, e execute o comando `git checkout 06MaisAlemComSpringData2`, para atualizar seu estado.

Com o *client* e a *API* rodando com os novos recursos, teste novamente o funcionamento da app cliente recarregando a página (`http://localhost:3000/`) e veja que o conteúdo apresentado já conta com a paginação.

DOCUMENTANDO E INTERAGINDO COM NOSSOS ENDPOINTS

É importante que o cliente, ao consumir nossa API, conheça as funcionalidades disponíveis e seus detalhes de uso. Isso facilita muito a vida dos desenvolvedores de APIs, já que evita manter uma comunicação constante com os desenvolvedores da aplicação cliente para detalhar qualquer nova modificação realizada na aplicação. Além disso, agiliza a integração. Para que isso seja possível, é necessário que nossa API esteja bem documentada para os seus usuários.

O problema é que a produção de uma boa documentação dá muito trabalho, gastando horas que poderiam ser melhor aproveitadas para o desenvolvimento da própria API. A boa notícia é que atualmente existem várias ferramentas que automatizam este processo e a mais conhecida e utilizada delas é o Swagger.

7.1 SWAGGER

O Swagger é um *framework* que provê ferramentas que ajudam na modelagem, especificação e documentação de APIs. Segue a iniciativa [Open API](#) que busca e prega a padronização de APIs REST. Na verdade, o Swagger é uma especificação e possui implementações para diversas linguagens e *frameworks*. Em nossa API vamos utilizar a implementação específica para integração do Swagger com o Spring, o *Springfox*.

Podemos gerar uma documentação em tempo de execução de forma automática usando o *Springfox*. Primeiro vamos inserir as dependências do *Springfox*:

```
<dependencies>
    <!-- ... -->
    <!-- dependência do spring-boot-devtools -->

    <dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger2</artifactId>
        <version>2.9.2</version>
    </dependency>

    <dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger-ui</artifactId>
        <version>2.9.2</version>
    </dependency>
```

```
<dependencies>
```

Agora precisamos criar a classe de configuração do Swagger, a `SwaggerConfiguration`. Usamos novamente a anotação `@Configuration` para que seus métodos sejam processados e gerenciados pelo Spring e adicionamos conjuntamente a anotação `@EnableSwagger2` do Springfox:

```
@EnableSwagger2  
@Configuration  
public class SwaggerConfiguration {  
  
}
```

A anotação `@EnableSwagger2` indica que o suporte ao Swagger deve ser habilitado. Segundo a documentação, deve ser aplicado a uma classe de configuração do Spring com a anotação `@Configuration`, exatamente como nossa `SwaggerConfiguration` acima.

Basicamente, a configuração é centrada em uma classe do `SpringFox` chamada `Docket`, que fornece métodos de configuração do Swagger. Vamos criar um *bean method* que retorne um `Docket` para que o Spring gerencie esse objeto:

```
@EnableSwagger2  
@Configuration  
public class SwaggerConfiguration {  
  
    @Bean  
    public Docket api() {  
  
    }  
}
```

Dessa maneira, já é possível gerar a documentação, mas não queremos, por exemplo, expor nossa `HomeController` na documentação. Essa classe foi criada apenas para testar a configuração do Spring Boot no início do curso e não tem relação com o Fórum da Alura. Vamos então customizar a configuração padrão do Swagger.

A classe `Docket` possui vários métodos que seguem o padrão *builder method*, ou seja, retornam a própria classe e podem ser chamados de forma encadeada. Primeiro, vamos retornar um `Docket` chamando seu construtor que espera receber o tipo de documentação a ser gerada. Escolheremos `SWAGGER_2`, já que estamos utilizando a versão 2 do Swagger:

```
@EnableSwagger2  
@Configuration  
public class SwaggerConfiguration {  
  
    @Bean  
    public Docket api() {  
        return new Docket(DocumentationType.SWAGGER_2);  
    }  
}
```

Após instanciar um `Docket`, chamamos o método `select()`, que retorna uma instância de

`ApiSelectorBuilder` e que fornece uma maneira de controlar os terminais expostos pelo Swagger. Vamos passar o pacote base de nossa aplicação para o Swagger gerar a documentação por meio do método `apis()` de `ApiSelectorBuilder` e o método `build()` para construir um `Docket`:

```
@EnableSwagger2
@Configuration
public class SwaggerConfiguration {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.basePackage("br.com.alura.forum"))
            .build();
    }
}
```

Como não queremos expor nossa `HomeController`, vamos pedir que o Swagger gere a documentação apenas para os *endpoints* que iniciem com `/api/`. Para isso, basta chamar o método `paths()` de `ApiSelectorBuilder` e passar o padrão do caminho desejado por meio de um `PathSelectors` que fará o tratamento necessário:

```
@EnableSwagger2
@Configuration
public class SwaggerConfiguration {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2);
            .select()
            .apis(RequestHandlerSelectors.basePackage("br.com.alura.forum"))
            .paths(PathSelectors.ant("/api/**"))
            .build();
    }
}
```

Com o Swagger configurado, vamos acessar a documentação gerada. Acesse `http://localhost:8080/v2/api-docs` pelo navegador e veja que a documentação é gerada em formato JSON:

```
{
  "swagger": "2.0",
  "info": {
    "description": "Api Documentation",
    "version": "1.0",
    "title": "Api Documentation",
    "termsOfService": "urn:tos",
    "contact": {},
    "license": {
      "name": "Apache 2.0",
      "url": "http://www.apache.org/licenses/LICENSE-2.0"
    }
  },
  "host": "localhost:8080",
  "basePath": "/",
  "tags": [
    {
      "name": "topic-controller",
      "description": "Topic Controller"
    }
  ],
  "paths": {
    "/api/topics": {
      "get": {
        "tags": [
          "topic-controller"
        ],
        "summary": "listTopics",
        "operationId": "listTopicsUsingGET",
        "produces": [
          "application/json"
        ],
        "parameters": [
          {
            "name": "categoryName",
            "in": "query",
            "required": false,
            "type": "string"
          },
          {
            "name": "offset",
            "in": "query",
            "type": "integer"
          }
        ]
      }
    }
  }
}
```

Figura 7.1: Swagger Json Docs

Além dos *endpoints*, veja que o Swagger também gerou a definição de nossos modelos.

Mas essa não é a maneira ideal de apresentar a documentação. Não é de fácil leitura para humanos. O Swagger oferece uma ferramenta que renderiza a documentação em uma interface interativa chamada Swagger UI e podemos acessá-la pelo endereço <http://localhost:8080/swagger-ui.html>:

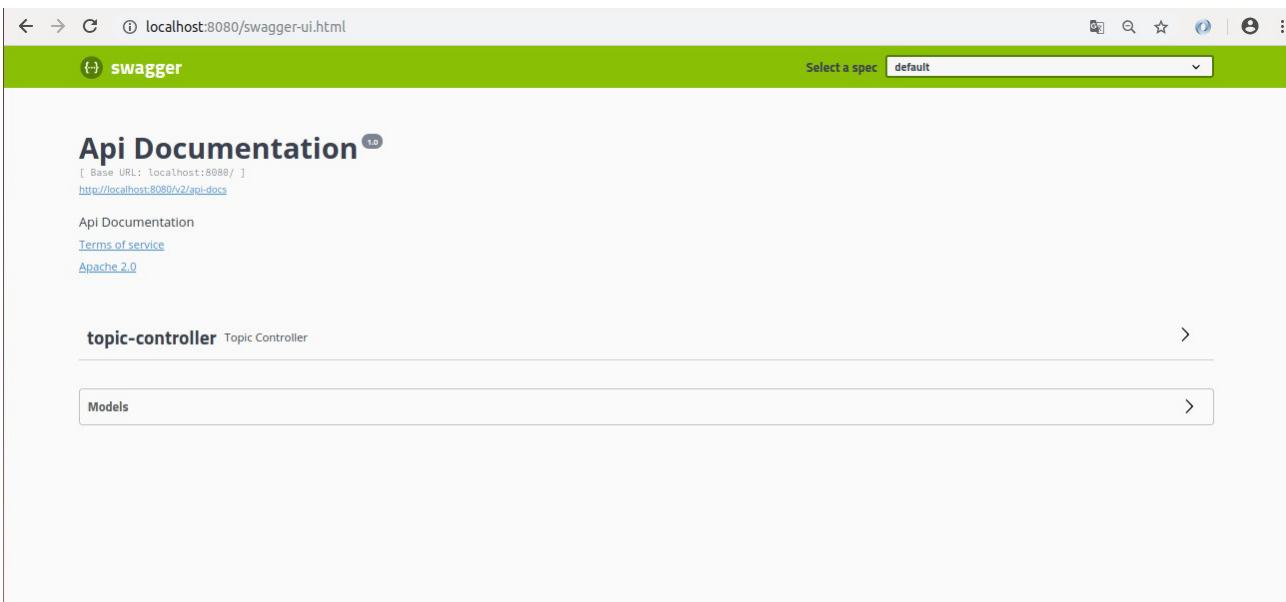


Figura 7.2: Swagger UI

Clique em `topic-controller` e no botão `GET` e veja que a documentação para o *endpoint* `/api/topics` de nossa `TopicController` foi gerada. E mais, podemos clicar no botão `Try it out` à direita e fazer uma requisição, aproveitando a interatividade da interface:

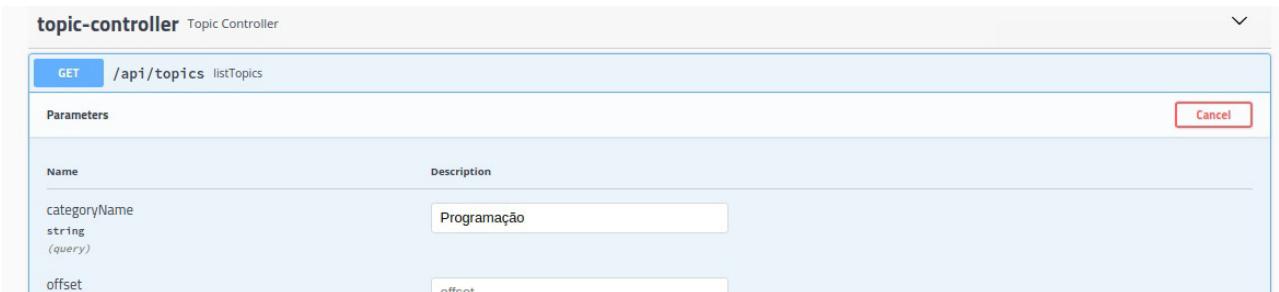


Figura 7.3: Swagger UI

Em seguida, clique em `Execute` e veja que a resposta é devolvida em formato `JSON`:

```

{
  "content": [
    {
      "id": 28,
      "shortDescription": "Como ler arquivo csv com java.io?",
      "secondsSinceLastUpdate": 11464106,
      "ownerName": "Alberto Souza",
      "courseName": "Java e Orientação a Objetos",
      "subcategoryName": "Java",
      "categoryName": "Programação",
      "numberOfResponses": 0,
      "solved": false
    },
    {
      "id": 22,
      "shortDescription": "Quando usar classes abstratas?",
      "secondsSinceLastUpdate": 11986196,
      "ownerName": "Alberto Souza",
      "courseName": "Java e Orientação a Objetos",
      "subcategoryName": "Java",
      "categoryName": "Programação",
      "numberOfResponses": 1,
      "solved": false
    },
    {
      "id": 16,
      ...
    }
  ]
}
  
```

Figura 7.4: Swagger Response

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

7.2 CUSTOMIZAÇÃO

Com poucas linhas de código já pudemos sentir o poder do Swagger. Vamos customizar a apresentação acrescentando informações sobre nossa API para o cliente. Basta adicionarmos a chamada do método `apiInfo()`, que recebe um objeto `ApiInfo` contendo as informações da API.

Vamos instanciar um `ApiInfoBuilder`, da biblioteca do Springfox, responsável por configurar essas informações e retornar um `ApiInfo` seguindo o padrão *builder method* novamente:

```
@EnableSwagger2
@Configuration
public class SwaggerConfiguration {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.basePackage("br.com.alura.forum"))
            .paths(PathSelectors.ant("/api/**"))
            .build()

            .apiInfo(
                new ApiInfoBuilder()
                    .title("Alura Forum API Documentation")
                    .description("Esta é a documentação interativa da Rest API
                        do Fórum da Alura. Tente enviar algum request ;)")
                    .version("1.0")
                    .build()
            );
    }
}
```

Note que acrescentamos um título, uma descrição e a versão que se encontra nossa API. Salve as

mudanças e acesse novamente `http://localhost:8080/swagger-ui.html`:

A screenshot of a web browser displaying the Swagger UI at `http://localhost:8080/swagger-ui.html`. The title bar shows the URL. The main header is "Alura Forum API Documentation". Below it, there's a note: "[Base URL: localhost:8080/] [http://localhost:8080/v2/api-docs]". A message says, "Esta é a documentação interativa da Rest API do Fórum da Alura. Tente enviar algum request ;)". There are two main sections visible: "topic-controller Topic Controller" and "Models". Each section has a right-pointing arrow indicating more content.

Figura 7.5: Swagger UI

É interessante que a API exponha dados para contato com a equipe de desenvolvimento, para qualquer problema que possa surgir. O Springfox possui uma classe `Contact` para essa tarefa. A classe `Contact` possui um construtor que recebe três `Strings` representando o nome, url e email, respectivamente. Em seguida, chamamos o método `contact()` de `ApiInfoBuilder` dentro de `apiInfo()`:

```
@EnableSwagger2
@Configuration
public class SwaggerConfiguration {

    @Bean
    public Docket api() {
        Contact contato = new Contact("Alura",
            "https://cursos.alura.com.br/", "contato@alura.com.br");

        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlersSelectors.basePackage("br.com.alura.forum"))
            .paths(PathSelectors.ant("/api/**"))

            .build()
            .apiInfo(
                new ApiInfoBuilder()
                    .title("Alura Forum API Documentation")
                    .description("Esta é a documentação interativa da Rest API
                        do Fórum da Alura. Tente enviar algum request ;)")
                    .version("1.0")
                    .contact(contato)
                    .build()
            );
    }
}
```

Acesse novamente `http://localhost:8080/swagger-ui.html` e veja o resultado:

A screenshot of a web browser displaying the Swagger UI. The title bar shows the URL `localhost:8080/swagger-ui.html`. The main header is "swagger". A dropdown menu says "Select a spec" with "default" selected. Below the header, the title "Alura Forum API Documentation" is displayed with a "1.0" badge. It includes a note "[Base URL: localhost:8080 /]" and a link "<http://localhost:8080/v2/api-docs>". A message says "Esta é a documentação interativa da Rest API do Fórum da Alura. Tente enviar algum request ;)" and links to "Alura - Website" and "Send email to Alura". A sidebar on the left lists "topic-controller" (Topic Controller) and "Models".

Figura 7.6: Swagger UI

Repare que o Swagger criou dois links: `Alura - Website`, que redireciona para a página da Alura como especificamos, e `Send Email to Alura`, que abre a aplicação padrão de email do sistema operacional do usuário para enviar um email para `contato@alura.com.br`.

Este capítulo foi uma pequena introdução ao Swagger e Springfox. Para conhecer outras possibilidades com Swagger, acesse o [site](#) da ferramenta.

7.3 EXERCÍCIOS: HABILITANDO O SPRINGFOX SWAGGER 2

- Para que seja possível configurar os detalhes sobre como documentar nossos endpoints com Swagger, primeiro precisamos adicionar as dependências da implementação Springfox.

Adicione as seguintes dependências no arquivo `pom.xml`:

```
<dependencies>
    <!-- ... -->
    <!-- dependência do spring-boot-dev-tools -->

    <dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger2</artifactId>
        <version>2.9.2</version>
    </dependency>

    <dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger-ui</artifactId>
        <version>2.9.2</version>
    </dependency>
</dependencies>
```

2. Com os jars já disponíveis na aplicação precisamos apenas configurar alguns detalhes mínimos sobre nossa API. No pacote `br.com.alura.forum.configuration`, crie a classe `SwaggerConfiguration`:

```
@Configuration  
@EnableSwagger2  
public class SwaggerConfiguration {  
  
}
```

3. Nessa classe, adicione o método de configuração que disponibiliza um objeto da classe `Docket` do `springfox` ao Spring:

```
@Configuration  
@EnableSwagger2  
public class SwaggerConfiguration {  
  
    @Bean  
    public Docket api() {  
        return new Docket(DocumentationType.SWAGGER_2)  
            .select()  
            .apis(RequestHandlerSelectors.basePackage("br.com.alura.forum"))  
            .paths(PathSelectors.ant("/api/**"))  
            .build();  
    }  
}
```

4. Adicione também as informações personalizadas sobre a nossa API que devem ser apresentadas na documentação gerada pelo Swagger:

```
@Configuration  
@EnableSwagger2  
public class SwaggerConfiguration {  
  
    @Bean  
    public Docket api() {  
        return new Docket(DocumentationType.SWAGGER_2)  
            .select()  
            .apis(RequestHandlerSelectors.basePackage("br.com.alura.forum"))  
            .paths(PathSelectors.ant("/api/**"))  
            .build()  
            .apiInfo(apiInfo());  
    }  
  
    private ApiInfo apiInfo() {  
        Contact contato = new Contact("Alura",  
            "https://cursos.alura.com.br/", "contato@alura.com.br");  
  
        return new ApiInfoBuilder()  
            .title("Alura Forum API Documentation")  
            .description("Esta é a documentação interativa da Rest API do Fórum da Alura. Tente enviar algum request ;)")  
            .version("1.0")  
            .contact(contato)  
            .build();  
    }  
}
```

Os *imports* necessários são do pacote `springfox.documentation`.

5. Rode a aplicação e acesse em seu navegador o endereço `http://localhost:8080/swagger-ui.html`.

Veja que é apresentada uma página com a documentação interativa da API, de onde é possível inclusive testar os *endpoints*, efetuando requisições HTTP por meio da ferramenta.

6. (Opcional) Adicionalmente, você poderia também personalizar as mensagens padrão que serão apresentadas ao testar os *endpoints* na documentação interativa:

```
@Configuration
@EnableSwagger2
public class SwaggerConfiguration {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.basePackage("br.com.alura.forum"))
            .paths(PathSelectors.ant("/api/**"))
            .build()
            .apiInfo(apiInfo())
            .globalResponseMessage(RequestMethod.GET,
                Arrays.asList(
                    new ResponseMessageBuilder()
                        .code(500)
                        .message("Xix! Deu erro interno no servidor.")
                        .build(),
                    new ResponseMessageBuilder()
                        .code(403)
                        .message("Forbidden! Você não pode acessar esse recurso.")
                        .build(),
                    new ResponseMessageBuilder()
                        .code(404)
                        .message("O recurso que você buscou não foi encontrado.")
                        .build()));
    }

    // método apiInfo() omitido
}
```

7.4 SUPER DESAFIO: IMPLEMENTE A APRESENTAÇÃO DO DASHBOARD DE DÚVIDAS DO FÓRUM

Além de apresentar os tópicos com filtragem e paginação, nossa aplicação deve também apresentar um dashboard com informações estatísticas sobre o fórum.

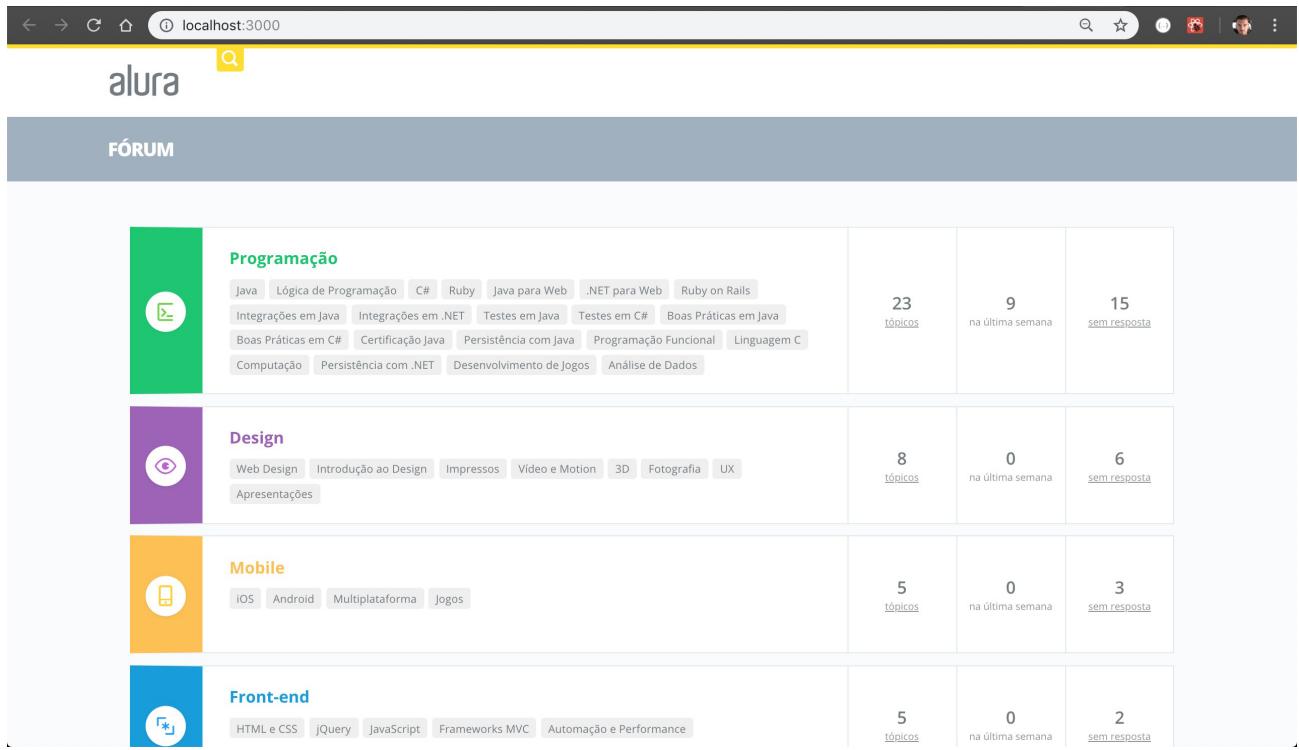


Figura 7.7: dashboard do forum

Agora é com você! Implemente a funcionalidade abaixo de acordo com as especificações requeridas.

- O cliente fará um requisição `GET` para o endpoint `/api/topics/dashboard` , e espera receber como resposta o seguinte `JSON`.

```
[
  {
    "categoryName": "Programação",
    "subcategories": [
      "Java",
      "Lógica de Programação",
      "C#",
      "Ruby",
      "Java para Web",
      // ... demais subcategorias
    ],
    "allTopics": 23,
    "lastWeekTopics": 9,
    "unansweredTopics": 15
  },
  {
    "categoryName": "Design",
    "subcategories": [
      "Web Design",
      "Introdução ao Design",
      "Impressos",
      "Vídeo e Motion",
      "3D",
      // ... demais subcategorias
    ],
    "allTopics": 8,
    "lastWeekTopics": 0,
  }
]
```

```
        "unansweredTopics": 6
    },
    // demais objetos das categorias principais
]
```

Utilize o apoio do(a) instrutor(a) como seu consultor nessa tarefa.

2. Utilize a documentação do *Swagger* para ir testando a resposta da API enquanto desenvolve.
3. (Opcional) Caso queira testar a aplicação cliente com o dashboard, no terminal, acesse a pasta do projeto da aplicação cliente `Desktop/fj27-alura-forum-client/`, e execute o comando `git checkout 07DocumentandoEInteragindoComOsEndpoints`, para atualizar seu estado.

Com o *client* e a *API* rodando com os novos recursos, teste novamente o funcionamento da app cliente recarregando a página (`http://localhost:3000/`) e veja que o conteúdo apresentado já conta com a paginação.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

PROTEGENDO A API COM SPRING SECURITY

8.1 AUTENTICAÇÃO COM SPRING SECURITY

Quando pensamos nas perguntas e respostas no fórum da Alura, é importante saber quem abriu o tópico e quem participou da discussão. Precisamos por exemplo, enviar e-mail para notificar os usuários da nova resposta. Existe também o fato de que para abrir ou participar de uma discussão, é necessário ser um usuário da Alura.

The screenshot shows a forum thread titled "Spring Boot é bom mesmo?". The thread has been updated 6 months ago. It features a response from user "Rafael Rollo" (1.8k xp, 5 posts) and one reply from "Thais André" (2412.4k xp, 4824 posts). The reply from Thais André says "Sim, muito bom!" and includes a "mark as solution" button. There is also a placeholder for a comment section labeled "O que você acha disso?".

Figura 8.1: Exemplo de Thread no fórum

Algo muito comum dentro dos sistemas é justamente só exibir determinadas informações ou permitir certas ações depois que o usuário de alguma forma se identificou previamente. Pense na sua rede social favorita: não é possível curtir es compartilhar posts sobre o assunto do momento, sem antes realizar o login. Normalmente existe um formulário para digitarmos usuário e senha, e então são exibidas as informações relacionadas à sua conta.

Isso faz parte da segurança de uma aplicação. O processo onde uma pessoa realiza seu login, se identificando dentro do sistema, é conhecido como *autenticação*. Poderíamos realizar todos esses

processos na mão. Mas imagine a quantidade de coisas que teríamos que nos preocupar: todos os ataques de segurança possíveis, criar filtros/*interceptors* para lidar com as requisições que chegam, etc.

A autenticação é apenas a primeira parte do problema: uma vez que o usuário está logado na aplicação, ainda precisamos ver se ele tem acesso a um determinado recurso dentro do sistema. Para ajudar com todas essas questões relacionadas com a segurança do sistema, o Spring possui um módulo chamado [Spring Security](#).

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

8.2 INICIANDO COM SPRING SECURITY

Toda a configuração inicial, junto com as dependências para que tudo funcione em harmonia, já está inclusa no `start` do Spring Security. É possível simplesmente adicionar a seguinte dependência no `pom.xml` do projeto:

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

Ao reiniciar a aplicação e tentar acessar qualquer url, vemos o seguinte resultado:

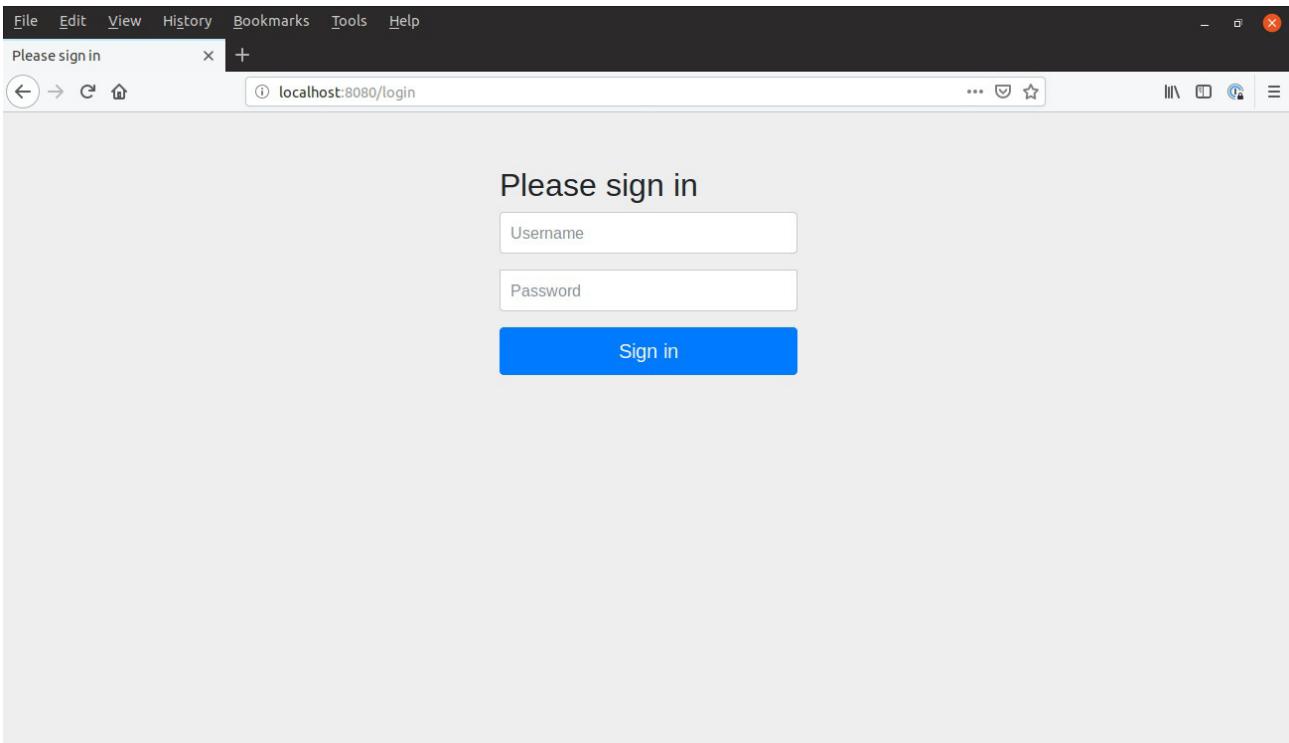


Figura 8.2: Formulário de login

Veja o que é possível *autoconfiguration* do Spring Boot: com a simples adição de uma dependência, temos todas as *urls* da aplicação protegidas. E ainda temos um formulário padrão de login. Mas qual o usuário e a senha? Por padrão, podemos logar com o usuário `user` a senha podemos encontrar nos logs da aplicação:

```
Using generated security password: 360a7add-28c0-40ff-89d1-c20e0036902e
```

Cada vez que o servidor for reiniciado uma nova senha é gerada. Apesar de ser muito interessante o que já ganhamos por padrão, normalmente queremos ir mais além: precisamos definir vários usuários, *roles*, e quais URLs da aplicação podem ser acessadas publicamente ou não. Para definir estas customizações, vamos criar uma classe de configuração:

```
@Configuration  
@EnableWebSecurity  
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {  
}
```

A primeira novidade aqui é a anotação `@EnableWebSecurity`. Ela indica que estamos criando um bean que sobrescreve as configurações pré-definidas do Spring security na parte da Web. Além disso, temos a classe `WebSecurityConfigurerAdapter` que ajuda a customizar as configurações.

Isto é feito por meio da sobreescrita de métodos. Para configurarmos alguns usuários, por exemplo, podemos sobreescriver um dos métodos `configure()` disponíveis:

```
@Override
```

```
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
}
```

8.3 DEFININDO NOSSOS USUÁRIOS

Existe uma interface no Spring chamada `AuthenticationManager`. Esta interface é a parte principal da autenticação. Voltaremos a falar sobre ela em breve. O `AuthenticationManagerBuilder` nos ajudará a construir um `AuthenticationManager`. A autenticação em memória é um dos tipos suportados:

```
@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
    auth.inMemoryAuthentication()  
        .withUser("alberto")  
        .password("123456")  
        .authorities("ROLE_ADMIN")  
    .and()  
        .withUser("rafael")  
        .password("123456")  
        .authorities("ROLE_USER");  
}
```

Agora temos dois usuários configurados na memória com as senhas que definimos.

O método `inMemoryAuthentication()` temos uma espécie de *builder* onde podemos configurar os valores desejados. É possível definir o usuário com o método `withUser()` e a senha com o método `password()`. Para finalizar, chamando o método `authorities()` para definir as permissões que ele terá dentro do sistema e trabalhar em cima disso depois. Este método retorna um `InMemoryUserDetailsManagerConfigurer` para permitir customização em memória.

8.4 ENCRYPTANDO AS SENHAS

Ao abrir o formulário de login e tentar realizar a autenticação, somos redirecionados para o formulário mesmo que os dados digitados pareçam corretos à primeira vista. Ao observar o console, uma exception ocorreu:

```
java.lang.IllegalArgumentException: There is no PasswordEncoder mapped for the id "null"
```

Um erro relacionado com `PasswordEncoder`. O Spring de alguma forma precisa saber que algoritmo de *encode* usamos para armazenar nossas senhas. Como nossas senhas estão em texto plano, vamos utilizar a classe `NoOpPasswordEncoder`:

```
@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
    auth.inMemoryAuthentication()  
        .withUser("alberto")  
        .password("123456")  
        .authorities("ROLE_ADMIN")  
    .and()
```

```

        .withUser("rafael")
        .password("123456")
        .authorities("ROLE_USER")
    .and()
        .passwordEncoder(NoOpPasswordEncoder.getInstance());
}

```

A classe `NoOpPasswordEncoder` é um implementação da interface `PasswordEncoder` que não aplica qualquer segurança na senha. Agora podemos tentar realizar o login.

Mas imagine senhas de usuários sendo armazenadas em texto plano em uma aplicação real. Se essa senha vazia porque alguém conseguiu *hackear* o banco de dados por exemplo? Por este motivo, o comum dentro de aplicações é que feito o *encode* da senha - uma transformação do dado para evitar de guardar a senha real no banco. Mais seguro ainda é aplicar uma função de *hash* que mapeia um dado arbitrário para um dado de tamanho fixo e que seja praticamente inviável fazer a inversão. Dessa forma as senhas salvas mesmo que acessadas não farão muito sentido para quem ler.

A classe `NoOpPasswordEncoder` é inclusive marcada com `@Deprecated`, não porque existe intenção de que ela seja removida do framework, mas para alertar os desenvolvedores que ela não deve ser utilizada em produção.

O método `passwordEncoder()` consegue receber qualquer implementação da interface `PasswordEncoder`. Temos algumas opções:

- `NoOpPasswordEncoder` Não aplica qualquer algoritmo. Recomendado apenas para testes
- `Pbkdf2PasswordEncoder` Aplica a encriptação Pbkdf2
- `BCryptPasswordEncoder` Aplica a encriptação BCrypt

Aqui vamos usar o [algoritmo bcrypt](#), que no momento é mais seguro do que a maioria dos outros métodos disponíveis e aplica uma função de *hash*. É necessário então trocar a implementação no código:

Aqui, vamos usar o [algoritmo bcrypt](#), que atualmente é o mais utilizado e o mais indicado contra ataques de força bruta. Será necessário trocar a implementação no código:

```

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        // código da adição dos usuário omitido
        .and()
        .passwordEncoder(new BCryptPasswordEncoder());
}

```

Ao tentar realizar o login, vemos o seguinte no console:

```
o.s.s.c.BCryptPasswordEncoder      : Encoded password does not look like BCrypt
```

Quando a senha digitada chega para o Spring no servidor, ela é encriptada e usada para fazer a comparação com a senha atual que está definida para o usuário. O problema é que quando o usuário é definido, deixamos a senha em texto plano. Precisamos então realizar a *encode* das senhas definidas:

```

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("alberto")
        .password(new BCryptPasswordEncoder().encode("123456")) // chamando encode()
        .authorities("ROLE_ADMIN")
    .and()
        .withUser("rafael")
        .password(new BCryptPasswordEncoder().encode("123456")) // chamando encode()
        .authorities("ROLE_USER")
    .and()
        .passwordEncoder(new BCryptPasswordEncoder());
}

```

Todas as classes que implementam a interface `PasswordEncoder` possuem o método `encode()` implementado. Ele trata de receber uma senha em texto plano e retornar a senha codificada (*encoded*). **Este é um caminho de mão única:** a senha salva no cadastro de um usuário, nunca é decodificada (*decoded*). O que terá que ser feito sempre, é codificar a senha digitada pelo usuário e comparar as duas senhas codificadas. E isto é feito pelo Spring por meio da chamada a um outro método da interface `PasswordEncoder` :

```
boolean matches(CharSequence rawPassword, String encodedPassword)
```

O primeiro parâmetro do método `matches()`, recebe a senha em texto plano (*raw password*) vindo do usuário. Esta senha será codificada e comparada com a senha já codificada - representada pelo segundo parâmetro, o `encodedPassword` - que definimos e foi codificada no momento da criação do usuário.



Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

8.5 EXERCÍCIO - PRIMEIRO LOGIN COM SPRING SECURITY

Para darmos prosseguimento ao projeto, precisaremos contar com as classes criadas no exercício anterior. Como explicado pelo instrutor e por esta apostila, existem n possíveis soluções para o exercício proposto, mas para que possamos prosseguir precisamos de uma implementação base conhecida que

sustente os próximos exercícios passo a passo. Por isso, na pasta de arquivos do curso existe uma das possíveis soluções para o exercício já pronta em um projeto, a qual vamos nos basear.

1. Caso você **não** tenha feito o *super desafio* do capítulo anterior, acesse a pasta do curso em `/caelum/cursos/27` e utilize o projeto `/forum-v2` para dar prosseguimento às atividades.
2. Para adicionar segurança à nossa aplicação vamos utilizar o *framework* do Spring dedicado a essa tarefa, o Spring Security.

Antes de dar início ao código, precisamos adicionar no arquivo `pom.xml` o *starter* do Spring Security para contar com a base de implementação que ele já traz pronta:

```
<dependencies>
    <!-- ... -->

    <!-- dependências do swagger -->

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

3. Com as dependências adicionadas, rode a aplicação novamente e tente acessar no navegador o endereço `http://localhost:8080/api/topics`. Perceba que automaticamente o Spring Security solicita a autenticação do usuário para acessar os *endpoints*:

Somos redirecionados para uma tela de login.

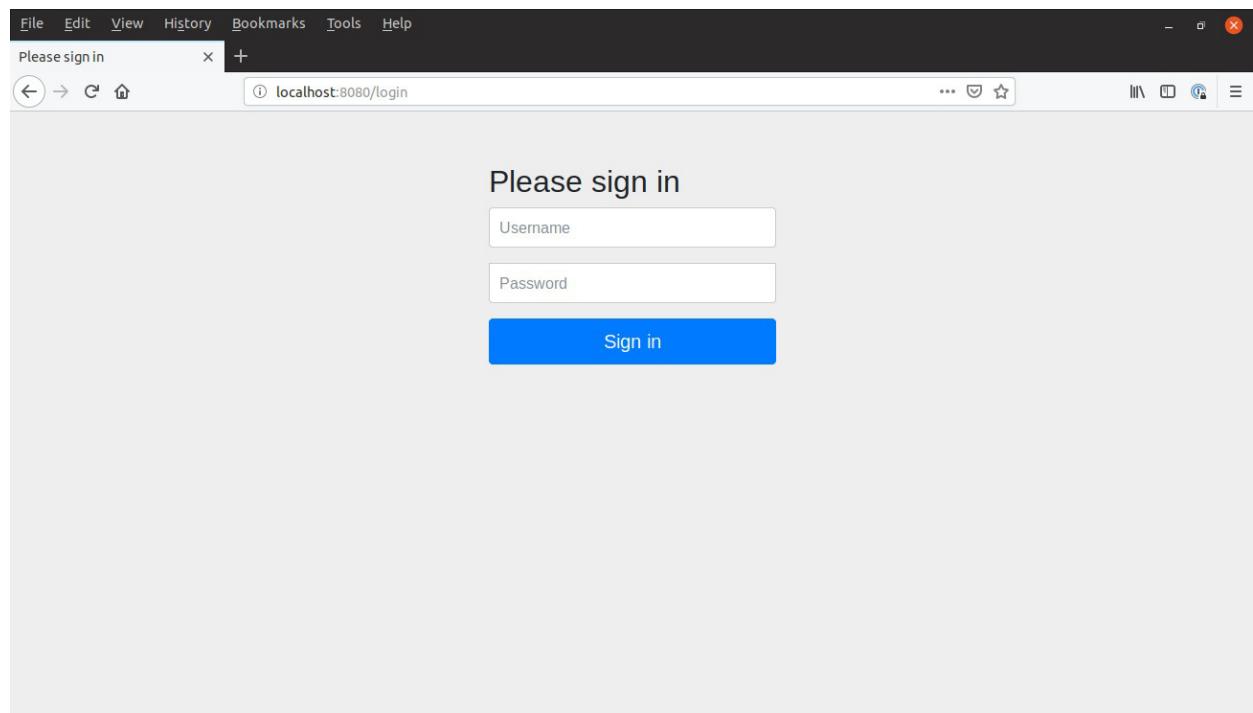


Figura 8.3: Tela de login do Spring Security

4. Faça o login com o *username* "user" e utilize o *password* gerado pelo Spring Security e que aparece no *log* da aplicação como no exemplo abaixo:

```
Using generated security password: 360a7add-28c0-40ff-89d1-c20e0036902e
```

Obs.: cada aplicação irá gerar um *password* diferente. Portanto o *password* acima não funcionará em sua máquina.

5. No pacote `br.com.alura.forum.security.configuration`, crie a classe `SecurityConfiguration`, estendendo a classe `WebSecurityConfigurerAdapter`. Esta já fornece toda a infraestrutura pronta para começarmos a fazer nossas configurações de segurança.

```
@Configuration  
@EnableWebSecurity  
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {  
  
}
```

6. Sobrescreva o método `configure()` que recebe um `AuthenticationManagerBuilder` e crie alguns usuário *in memory* através do método `inMemoryAuthentication()`:

```
@Configuration  
@EnableWebSecurity  
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {  
  
    @Override  
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
        auth.inMemoryAuthentication()  
            .withUser("alberto")  
            .password("123456")  
            .authorities("ROLE_ADMIN")  
        .and()  
            .withUser("rafael")  
            .password("123456")  
            .authorities("ROLE_USER")  
        .and()  
            .passwordEncoder(new BCryptPasswordEncoder());  
    }  
}
```

7. Rode a aplicação e tente acessar o endereço `http://localhost:8080/api/topics`. Faça o login com um dos usuários que foram configurados em memória no exercício anterior. O que acontece?

8. Para que o login com usuários configurados funcione, precisamos fazer o *encode* das senhas de acordo com o algoritmo escolhido:

```
@Override  
protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
    auth.inMemoryAuthentication()  
        .withUser("alberto")  
        .password(new BCryptPasswordEncoder().encode("123456"))  
        .authorities("ROLE_ADMIN")  
    .and()  
        .withUser("rafael")  
        .password(new BCryptPasswordEncoder().encode("123456"))  
        .authorities("ROLE_USER")
```

```
.and()
    .passwordEncoder(new BCryptPasswordEncoder());
}
```

9. Rode a aplicação e tente acessar novamente o endereço `http://localhost:8080/api/topics`. Faça o login com um dos usuários que foram configurados em memória no exercício anterior. Veja que agora o login funciona com os usuários que customizamos.

8.6 LOGIN COM DADOS DO BANCO

Pense numa aplicação do dia a dia: normalmente temos novos usuários se cadastrando com alguma frequência. Isso tornaria inviável manter todos os usuários em memória, cadastrando um a um no código-fonte. Pense em todo o trabalho relacionado a isso. Teríamos até que fazer um novo *deploy* da aplicação.

Mais um detalhe: os dados de usuário e senha continuarão no *bytecode* uma vez que o código-fonte é compilado. Alguém poderia ver os dados dos usuários ao visualizar este *bytecode*.

Devido estas questões, normalmente salvamos e recuperamos os usuários de algum lugar. Uma das opções mais comuns é salvar no banco de dados. E é isto que vamos fazer aqui.

Até este momento, o `AuthenticationManager` sabe apenas autenticar os usuários em memória que configuramos anteriormente. Para que ele faça a autenticação baseada nos usuários que estão no banco de dados, devemos "ensiná-lo" a fazer isso.

`AuthenticationManager` é apenas uma *interface* e sua implementação padrão é a `ProviderManager` - este *provider* vai delegar para uma lista de `AuthenticationProviders` que indica uma classe que pode processar a autenticação. Quando essa classe é encontrada, a autenticação é realizada. O *provider* padrão para recuperação de dados através de uma implementação do padrão `DAO` é o `DaoAuthenticationProvider` que recupera os detalhes do usuário a partir de um `UserDetailsService`.

O Spring Security possui uma interface chamada `UserDetailsService` que é responsável por carregar os dados de um usuário específico. Mudaremos, então, o método `configure()` que não vai mais utilizar a autenticação em memória mas utilizar um `UserDetailsService`:

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(usersService)
        .passwordEncoder(new BCryptPasswordEncoder());
}
```

Precisamos prover uma implementação de `UserDetailsService` para que o `AuthenticationManager` saiba recuperar os usuário para fazer a autenticação. Essa *interface* possui apenas um método chamado `loadByUsername()` que irá carregar o usuário pelo seu *username*. Vamos,

então, criar uma classe que implemente `UserDetailsService` :

```
@Service
public class UsersService implements UserDetailsService {

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        // carregar e retornar o usuário
    }
}
```

Para carregar um usuário pelo seu `username`, precisamos de alguém que sabe acessar o banco de dados. Como vimos nos capítulos anteriores, podemos criar um `repository` de usuários para isso, com um método que carrega o usuário pelo seu `username`. Nossa `User` não possui um atributo `username` mas podemos utilizar o `email` como `username` - que é o que os usuários do Fórum da Alura utilizam para fazer login. Então, criaremos a classe `UserRepository` com o método `findByEmail()` :

```
public interface UserRepository extends Repository<User, Long> {

    User findByEmail(String email);
}
```

E provemos uma implementação do método `loadUserByUsername()` em nossa classe `UsersService` :

```
@Service
public class UsersService implements UserDetailsService{

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {

        User foundUser = userRepository.findByEmail(email);
        return foundUser;
    }
}
```

Acontece que este código não compila já que o método `loadUserByUsername()` da `interface UserDetailsService` espera o retorno de um `UserDetails`. `UserDetails` é outra `interface` do Spring Security que provê as informações principais de um usuário - suas implementações são armazenadas para que futuramente sejam encapsuladas por um objeto `Authentication` usado pelo `AuthenticationManager` para realizar a autenticação.

O Spring Security possui uma implementação de `UserDetails` de referência chamada `User`. Como já temos uma classe representando um usuário em nosso sistema, vamos fazer com que nossa classe `User` implemente a `interface UserDetails` .

```
@Entity
public class User implements UserDetails{
```

```
// código omitido  
}
```

Ao fazer isso, precisamos prover implementações para os métodos da *interface*. Os principais serão `getUsername()`, `getPassword()` e `getAuthorities()`.

```
@Entity  
public class User implements UserDetails{  
  
    @Id  
    @GeneratedValue(strategy=GenerationType.IDENTITY)  
    private Long id;  
  
    private String name;  
  
    private String password;  
  
    @Column(nullable = false, unique = true)  
    private String email;  
  
    @ManyToMany(fetch=FetchType.EAGER)  
    private List<Role> authorities = new ArrayList<>();  
  
    /**  
     * @deprecated  
     */  
    public User() {    }  
  
    public User(String name, String email, String password) {  
        this.name = name;  
        this.email = email;  
        this.password = password;  
    }  
  
    public Long getId() {  
        return this.id;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    @Override  
    public String getPassword() {  
        return this.password;  
    }  
  
    public String getEmail() {  
        return this.email;  
    }  
  
    @Override  
    public Collection<? extends GrantedAuthority> getAuthorities() {  
        return this.authorities;  
    }  
  
    @Override  
    public String getUsername() {  
        return this.email;  
    }
```

```

@Override
public boolean isAccountNonExpired() {
    return true;
}

@Override
public boolean isAccountNonLocked() {
    return true;
}

@Override
public boolean isCredentialsNonExpired() {
    return true;
}

@Override
public boolean isEnabled() {
    return true;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    User user = (User) o;
    return Objects.equals(email, user.email);
}

@Override
public int hashCode() {
    return Objects.hash(email);
}
}

```

Para os demais métodos retornaremos apenas `true` para garantir o acesso às URLs protegidas. Estes métodos são para um controle mais fino que não faremos uso nesta aplicação - caso seja necessário controlar renovação de senha ou bloqueio por inatividade, por exemplo, estes métodos serão úteis.

O método `getUsername()` deve retornar o `email`, já que definimos que o email será o `username` de nosso usuário - ou seja, ele vai utilizar o email para fazer a autenticação. O método `getPassword()` já possui uma implementação que retorna o `password`. E `getAuthorities()` precisa retornar uma lista de `GrantedAuthority` - cada `GrantedAuthority` representa um privilégio individual, como ser um usuário administrador, por exemplo. Precisamos definir esses privilégios, também chamados de `roles` ("papéis"). Essa informação também precisa ser persistida no banco para o Spring Security recuperá-la. Vamos criar a classe `Role` (repare que nossa classe `User` acima já possui uma lista de `Role` que será retornada pelo método `getAuthorities()`) para representar os privilégios:

```

@Entity
public class Role implements GrantedAuthority{

    public static final Role ROLE_ADMIN = new Role("ROLE_ADMIN");
    public static final Role ROLE_ALUNO = new Role("ROLE_ALUNO");

    @Id

```

```

private String authority;

/**
 * @deprecated
 */
public Role() {
}

public Role(String authority) {
    this.authority = authority;
}

@Override
public String getAuthority() {
    return this.authority;
}
}

```

Já criamos dois *roles* que serão usados pela aplicação que são o `ROLE_ALUNO` e `ROLE_ADMIN`, representando os usuários que são os alunos e os administradores do fórum, respectivamente.

Ótimo, agora garantimos as implementações necessárias para nosso `UsersService` funcionar. Vamos apenas modificar o retorno de `findByEmail()` para retornar um `Optional<User>` para refinar o tratamento no caso do usuário não existir:

```

public interface UserRepository extends Repository<User, Long> {

    Optional<User> findByEmail(String email);
}

```

No final, nossa `UsersService` deve ficar assim:

```

public class UsersService implements UserDetailsService{

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {

        Optional<User> foundUser = userRepository.findByEmail(email);

        return foundUser.orElseThrow(() ->
            new UsernameNotFoundException("Não foi encontrado o usuário com email " + email));
    }
}

```

Fizemos bastante coisa até aqui. Apesar de todas as facilidades do Spring Security, sua configuração não é trivial. O esquema abaixo é uma tentativa de sintetizar o fluxo de autenticação que será feito pelo Security após as novas configurações e facilitar o entendimento de cada passo da implementação que acabamos de realizar:

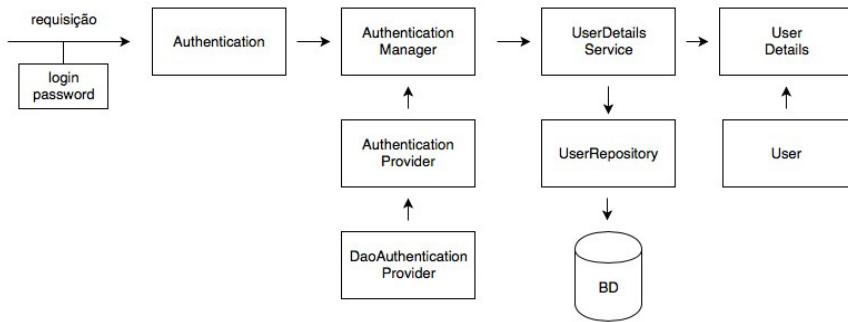


Figura 8.4: Security Cycle

Quando os dados de um usuário chega na aplicação através da tela de login padrão do Security, um objeto `Authentication` é criado com estes dados e passado para um `AuthenticationManager` autenticá-lo.

Como agora os usuários estão no banco de dados, nossa classe `User` implementa `UserDetails` que representa os dados de um usuário que será carregado pelo `UserDetailsService` através de um `UserRepository`. Registrarmos nosso `UserDetailsService` no `AuthenticationManager`. A classe `ProviderManager` é uma implementação de `AuthenticationManager` que agora vai encontrar na cadeia de providers a classe `DaoAuthenticationProvider` que sabe autenticar um usuário através de uma implementação de `UserDetailsService`.

Agora já é possível testar a aplicação com os usuários que criaremos no banco de dados durante os exercícios.

8.7 EXERCÍCIOS: AUTENTICAÇÃO COM SPRING SECURITY

- Na classe `SecurityConfiguration`, sobrescreva também o método `configure()` que recebe como parâmetro um `AuthenticationManagerBuilder`, informando qual o serviço responsável por prover os dados dos usuários durante o processo de autenticação, além da implementação padrão de criptografia que será usado para armazenar a senha do usuário.

```

@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Autowired
    private UsersService usersService;

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {

        auth.userDetailsService(usersService)
            .passwordEncoder(new BCryptPasswordEncoder());
    }
}

```

```
}
```

2. Neste momento, ainda não é possível se autenticar. Precisamos prover a implementação de `UserDetailsService` que será usada pelo `AuthenticationManager`.

No pacote `br.com.alura.forum.security.service`, crie a classe `UsersService`, que implementa a interface `UserDetailsService`, e sobrescreva o método `loadUserByUsername()`.

```
@Service
public class UsersService implements UserDetailsService {

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {

        return null;
    }
}
```

3. No pacote `br.com.alura.forum.repository`, crie também a interface `UserRepository` estendendo a interface `Repository` do Spring Data.

Declare o método `findByEmail()` que recebe uma `String` como parâmetro e devolve um `Optional<User>`. A classe `Optional` deve ser importada do pacote `java.util`.

```
public interface UserRepository extends Repository<User, Long>{

    Optional<User> findByEmail(String email);
}
```

4. Novamente na classe `UsersService`, altere a implementação do método `loadUserByUsername()` para que ela utilize o repositório de usuários.

```
@Service
public class UsersService implements UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {

        Optional<User> possibleUser = userRepository
            .findByEmail(username);

        return possibleUser.orElseThrow(() ->
            new UsernameNotFoundException("Não foi possível " +
                "encontrar usuário com email: " + username));
    }
}
```

5. Perceba que ainda temos um problema. Nossa implementação retorna uma `User` que ainda não é conhecido pelo Spring Security. O framework de segurança espera uma implementação de `UserDetails` para que consiga manipular as informações do usuário via polimorfismo.

Altere a classe `User` para implementar `UserDetails`. Aproveite para usar os recursos do eclipse para gerar o esqueleto dos métodos que devem ser implementados. (*Utilize CTRL + 1 > Add unimplemented methods*)

A classe deve ter o código abaixo ao final:

```
@Entity
public class User implements UserDetails {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String name;

    private String password;

    @Column(nullable = false, unique = true)
    private String email;

    @ManyToMany(fetch = FetchType.EAGER)
    private List<Role> authorities = new ArrayList<>();

    // construtores omitidos

    public String getName() {
        return name;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return this.authorities;
    }

    @Override
    public String getPassword() {
        return this.password;
    }

    @Override
    public String getUsername() {
        return this.getEmail();
    }

    public String getEmail() {
        return this.email;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
```

```

        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}

```

6. Perceba que foi necessário implementar o método `getAuthorities()` que devolve uma coleção de perfis de usuário, ou *roles*.

A classe `User` não compila. Crie a classe `Role`, no pacote `br.com.alura.forum.model`, implementando a interface `GrantedAuthority`:

```

@Entity
public class Role implements GrantedAuthority {

    public static final Role ROLE_ADMIN = new Role("ROLE_ADMIN");
    public static final Role ROLE_ALUNO = new Role("ROLE_ALUNO");

    @Id
    private String authority;

    /**
     * @deprecated
     */
    public Role () {}

    public Role(String authority) {
        this.authority = authority;
    }

    @Override
    public String getAuthority() {
        return this.authority;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Role role = (Role) o;
        return Objects.equals(authority, role.authority);
    }
}

```

7. Rode novamente a aplicação para que o Hibernate crie as novas tabelas no banco de dados.

Em seguida, no terminal, acesse a base de dados usada pela nossa API:

```
mysql -u root -pcaelum fj27_spring
```

Execute o comando de inserção de usuário na base de dados:

```
INSERT INTO user(name, email, password) VALUES(
    'Aluno',
    'aluno@gmail.com',
    '$2a$10$3Qrx0rv8qSmZ8s3RlD5qE.upleP7.Qzbg5EoIAM62evEkY4c023TK'
```

);

Para facilitar a execução desta etapa, utilize o arquivo `insert-aluno.sql` presente na pasta de arquivos do curso

8. Rode novamente a aplicação e tente acessar alguma rota não configurada como pública, por exemplo `http://localhost:8080/`. Novamente seremos redirecionados para a tela padrão de login do Spring Security.

Faça login com o email `aluno@gmail.com` e senha `123456` e veja que o acesso é liberado.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

8.8 AUTORIZAÇÃO COM SPRING SECURITY

No Forum da Alura alguns recursos são públicos, ou seja, qualquer usuário, independente de logado ou não, pode acessar. Este é o caso da lista de tópicos. Qualquer um pode navegar pelo site da Alura pesquisando cursos e acessando o fórum de dúvidas e o *dashboard*. Portanto, devemos adicionar a configuração que libera o endpoint `/api/topics` para todos os usuários. Configurar o acesso a recursos específicos está relacionado com o processo de *autorização* no contexto de segurança de uma aplicação.

O método responsável por controlar o acesso também se chama `configure()` da classe `WebSecurityConfigurerAdapter`, mas recebe como parâmetro um `HttpSecurity` que permite configuração baseada em segurança na web para requisições HTTP específicas.

Vamos, primeiro, checar a implementação padrão deste método:

```
protected void configure(HttpSecurity http) throws Exception {  
    // logger  
    http.authorizeRequests()
```

```

        .anyRequest().authenticated()
.and()
.formLogin()
.and()
.httpBasic();
}

```

O primeiro método é o `authorizeRequests()` que permite restringir acesso baseado em requisições e irá retornar o objeto para a configuração de regras de acesso. Note que logo é chamado o método `anyRequest()` seguido do método `authenticated()` o que significa que qualquer requisição deve ser feita por alguém autenticado. O método `formLogin()` gera a página de login padrão do Security através da url `/login` que usamos anteriormente e redireciona para `login?error` quando o login falha. E o método `httpBasic()` adiciona a configuração de autenticação baseado no protocolo HTTP chamada de [HTTP Basic](#).

Essa implementação é a que vem por padrão no Spring Security e obriga toda requisição ser autenticada. Vamos mudar isso para o *endpoint* `/api/topics` que lista todos os tópicos.

```

protected void configure(HttpSecurity http) throws Exception {
    // logger

    http.authorizeRequests()
        .antMatchers("/api/topics/**").permitAll()
        .anyRequest().authenticated()
    .and()
        .formLogin()
    .and()
        .httpBasic();
}

```

Ao adicionar `.antMatchers("/api/topics/**").permitAll()` logo após o método `authorizeRequests()` estamos dizendo que todos os recursos que comecem com `/api/topics` estão liberados para qualquer usuário (`permitAll`). O método `antMatcher()` recebe uma expressão regular no mesmo estilo suportado pela ferramenta ANT, famosa no mundo Java na parte de builds. Aqui é importante ressaltar que todas as restrições devem vir depois da chamada de `authorizeRequests()` e antes de `anyRequest().authenticated()` - o que faz todo sentido, já que o Security precisa saber primeiro as configurações específicas de autorização e liberar o resto apenas no caso do usuário estar no mínimo autenticado.

Também podemos controlar o acesso através dos métodos HTTP. Por exemplo, tanto a lista de tópicos quanto o *dashboard* são acessados por requisições GET. Podemos explicitar que liberaremos todos os *endpoints* que iniciem com `/api/topics` e que são feitas a partir do método GET do protocolo HTTP.

```

protected void configure(HttpSecurity http) throws Exception {
    // logger

```

```

http.authorizeRequests()
    .antMatchers(HttpMethod.GET, "/api/topics/**").permitAll()
    .anyRequest().authenticated()
    .and()
    .formLogin()
    .and()
    .httpBasic();
}

```

O método `antMatchers()` possui uma sobrecarga que aceita que o método do protocolo HTTP seja passado como argumento.

Definiremos outras regras de acesso a medida que avançarmos com o desenvolvimento da aplicação.

8.9 EXERCÍCIO: AUTORIZAÇÃO COM SPRING SECURITY

1. Agora precisamos começar a liberar o acesso aos *endpoints* públicos, como por exemplo o que devolve a lista de tópicos abertos do forum: `/api/topics`. Vamos então criar as regras de acesso às URIs.

Na classe `SecurityConfiguration` sobrescreva o método `configure()` que recebe como parâmetro um `HttpSecurity`, para personalizar as regras de acesso:

```

@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    // atributos omitidos

    // outros métodos omitidos

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers(HttpMethod.GET, "/api/topics/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .formLogin()
            .and()
            .httpBasic();
    }
}

```

2. Rode novamente a aplicação, e veja que agora o acesso para ao *endpoint* `/api/topics` foi liberado:



```
  "content": [    {      "id": 33,      "shortDescription": "Qual melhor dinâmica para a minha retro?",      "secondsSinceLastUpdate": 2045899,      "ownerName": "Alberto Souza",      "courseName": "Scrum",      "sub categoryName": "Agilidade",      "categoryName": "Business",      "numberOfResponses": 1,      "solved": false    },    {      "id": 32,      "shortDescription": "Problema ao executar procedure!",      "secondsSinceLastUpdate": 2049499,      "ownerName": "Alberto Souza",      "courseName": "MySQL I",      "sub categoryName": "SQL",      "categoryName": "Infraestrutura",      "numberOfResponses": 1,      "solved": false    },    {      "id": 31,      "shortDescription": "Problemas ao executar react-native run-ios",      "secondsSinceLastUpdate": 2139499,      "ownerName": "Alberto Souza",      "courseName": "React Native parte 2",      "sub categoryName": "Multiplataforma",      "categoryName": "Mobile",      "numberOfResponses": 1,      "solved": false    },    {      "id": 30,      "shortDescription": "Como posso personalizar o bootstrap?",      "secondsSinceLastUpdate": 2229499,    }  ]
```

Figura 8.5: Acessando recurso público

3. Rode novamente a aplicação e tente acessar alguma rota segura, por exemplo <http://localhost:8080/>. Novamente seremos redirecionados para a tela padrão de login do Spring Security.

8.10 UTILIZANDO JWT COM SPRING SECURITY

Já conseguimos realizar a autenticação com a página padrão do Spring Security, mas a aplicação cliente possui uma tela de login própria e receberemos os dados de autenticação por ela:

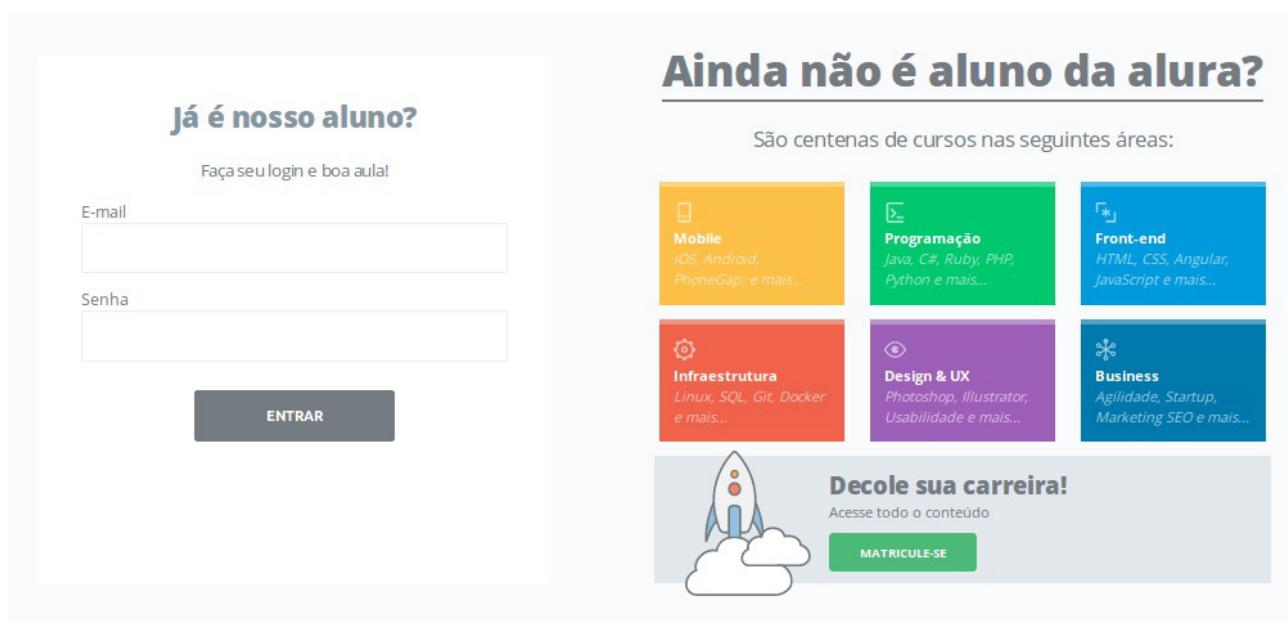


Figura 8.6: Tela login client

Portanto, não utilizaremos mais a tela de login padrão do Security e a aplicação cliente vai enviar os dados de autenticação (*email* e *password*) no formato json:

```
{
    "email": "rafael.rollo@caelum.com.br",
    "password": "rafael"
}
```

E novamente utilizaremos a estratégia do DTO para receber estes dados:

```
public class LoginInputDto {

    private String email;
    private String password;

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

Atualmente, o Security cria um objeto `HttpSession` para cada cliente que realiza uma autenticação

com sucesso. Um `HttpSession` provê uma maneira de identificar um usuário por mais de uma requisição e armazena informações sobre este usuário no servidor - ou seja, um usuário logado no sistema é aquele que possui uma sessão.

Como estamos desenvolvendo um *web service*, vamos analisar as desvantagens dessa abordagem. Imagine que podemos ter mais de um cliente, além da *Web Client React* do forum. Agora imagine várias sessões abertas de vários clientes e o servidor começa a ficar sobrecarregado. Uma solução para este cenário seria colocar um balanceamento de carga (*Load Balancer*) que distribui a carga de trabalho entre dois ou mais servidores - ou seja, replicamos a aplicação servidor em um ou mais servidores:

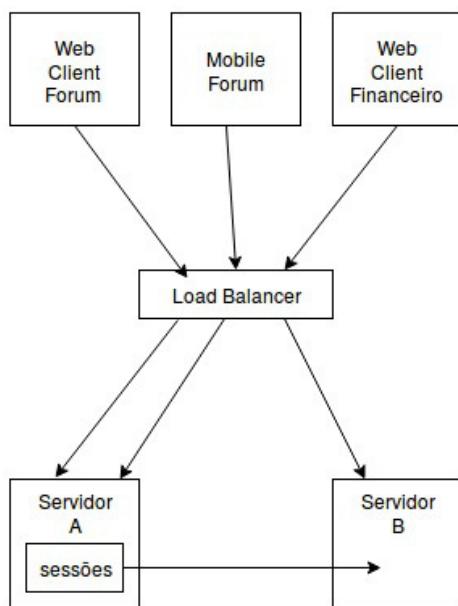


Figura 8.7: Load balancer session

Caso o servidor A venha a cair, o *load balancer* vai redirecionar as requisições para o servidor B , mas antes precisa pegar todos os dados dos clientes que estavam no servidor A e transferir para o servidor B . Ou seja, a sessão dos usuários precisa trafegar entre servidores e a manutenção começa a ganhar complexidade.

Por este motivo, quando falamos em *web services*, em que há a possibilidade de existir vários clientes e muitas sessões abertas, o mais recomendado é que o cliente armazene os dados do usuário e os envie a cada requisição para o servidor. O mais comum é usar um *token* de acesso, que é uma *string* codificada com alguns dados do usuário que o cliente manda em todas as requisições para um recurso seguro:

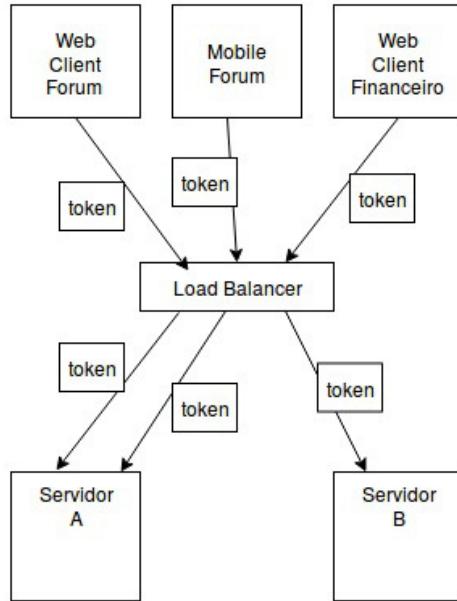


Figura 8.8: load balancer token

Dessa maneira não precisamos nos preocupar em administrar sessões quando escalar a aplicação para mais servidores. Iremos, então, criar um *token* de acesso para o cliente no momento da autenticação.

Existem vários padrões abertos para a criação de um token de acesso. O SAML (Security Assertion Markup Language) foi o primeiro a ser utilizado, usa formato XML para transferir informações mas possui tamanho significativamente maior que os demais padrões. Em seguida surgiu o SWT (Simple Web Token) que trouxe mais simplicidade do que o SAML mas apenas permite verificação simétrica. Hoje, o padrão mais utilizado é o JWT (JSON Web Token) que é mais compacto que os demais e permite verificação com chaves assimétricas - além disso, o formato JSON é o mais utilizado na web atualmente e o torna a melhor escolha para a maioria das aplicações que utilizam um *token* de acesso.

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

8.11 JWT

Um JSON Web Token (JWT) é um padrão ([RFC 7519](#)) usado para transferir informações codificadas entre duas partes, utilizando o formato JSON. Essas informações são chamadas de *claims*. O JWT ainda provê que o *token* seja assinado, garantindo a integridade do mesmo, por este motivo possui 3 partes:

- Header

O cabeçalho contém duas informações: o algoritmo usado para assinar o *token* e o tipo do *token*:

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

- Payload

O *payload* é a parte que contém as informações. Essas informações são de uma entidade (geralmente um usuário).

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "admin": true  
}
```

É indicado pela documentação que outras informações também estejam no *payload* como **iss**(emissor), **exp**(tempo de expiração), **sub** (assunto) e **aud** (audiência).

- Signature

Para criar a parte da assinatura é necessário codificar o *header* e o *payload* com uma palavra secreta (*secret*) usando o algoritmo definido no cabeçalho:

```
HMACSHA256(  
    base64UrlEncode(header) + "." +  
    base64UrlEncode(payload),  
    secret)
```

A assinatura é usada para verificar a integridade do *token*, ou seja, se a mensagem não foi modificada no meio do caminho. Em casos de uso de uma chave privada, também é possível verificar se o remetente do *token* é quem ele diz ser.

Abaixo, temos um exemplo de um JWT que provê *header*, *payload* e a chave assinada:

```
...  
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpvag4gRG9lIiwiawF0IjoxNTE2MjM5MDIyfQ.  
Sf1KxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c  
...
```

É possível decodificar o conteúdo do *token* no próprio [site do JWT](#)).

The screenshot shows the jwt.io interface. At the top, there's a navigation bar with links for Debugger, Libraries, Introduction, Ask, and Get a T-shirt!. On the right, it says "Crafted by Auth0". Below the navigation, there's a dropdown menu for "ALGORITHM" set to "HS256".

Encoded: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c

Decoded:

- HEADER: ALGORITHM & TOKEN TYPE**

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

- PAYOUT: DATA**

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

- VERIFY SIGNATURE**

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
)
```

secret base64 encoded

Figura 8.9: jwt site decoded

Portanto, o JWT não propõe esconder as informações mas trafegá-la de maneira consistente e prover a integridade da mesma.

8.12 GERANDO UM TOKEN DE ACESSO COM JJWT

Para que possamos implementar a geração de um JWT, vamos contar com a ajuda de uma biblioteca que vai ajudar a criar, assinar e validar um JWT. O site jwt.io possui bibliotecas para várias linguagens e facilita implementar autenticação com *tokens* de acesso no padrão JWT.

Precisamos, então, adicionar a dependência do `jjwt` que é a biblioteca para a linguagem Java:

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt</artifactId>
  <version>0.9.1</version>
</dependency>
```

Como não utilizaremos mais a tela de login do Spring Security e sim a da aplicação cliente, ele vai

mandar o login e senha do usuário do fórum em formato JSON através do endpoint `/api/auth`. Vamos precisar criar um controller para atender este recurso e autenticar o usuário:

```
@RestController
@RequestMapping("/api/auth")
public class UserAuthenticationController {

    @PostMapping
    public void authenticate(
        @RequestBody LoginInputDto loginInfo) {

    }
}
```

Note que receberemos os dados do usuário através de uma requisição POST. Isso quer dizer que os dados serão trafegados pelo corpo da requisição. O Spring MVC possui a anotação `@RequestBody` para ser utilizada nesses casos - dessa maneira ele não vai procurar pelos parâmetros `email` e `password` na url e sim no corpo da requisição. E usaremos o parâmetro `consumes` para definir que este método irá consumir um JSON.

Para fazer a autenticação, vamos precisar fazer o que o Spring Security está fazendo por baixo dos panos quando fazemos a autenticação pela sua tela de login padrão que é chamar o método `authenticate()` do `AuthenticationManager`. Portanto, iremos precisar de um objeto `AuthenticationManager` injetado:

```
@RestController
@RequestMapping("/api/auth")
public class UserAuthenticationController {

    @Autowired
    private AuthenticationManager authManager;

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE)
    public void authenticate(
        @RequestBody LoginInputDto loginInfo) {

    }
}
```

Mas ao rodar a aplicação neste momento recebemos um erro:

```
*****
APPLICATION FAILED TO START
*****
Description:
Field authManager in br.com.alura.forum.controller.AuthenticationController required a bean of type 'org.springframework.security.authentication.AuthenticationManager' which could not be found.
The injection point has the following annotations:
 - @org.springframework.beans.factory.annotation.Autowired(required=true)

Action:
Consider defining a bean of type 'org.springframework.security.authentication.AuthenticationManager' in your configuration.
```

Figura 8.10: auth manager error

O erro diz que não foi possível encontrar uma implementação de AuthenticationManager para satisfazer a dependência. Uma maneira de fazer isso é expor o AuthenticationManager como um bean através da implementação do método authenticationManagerBean() de WebSecurityConfigurerAdapter na classe de configuração SecurityConfiguration :

```
@EnableWebSecurity
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    // outro atributos e métodos omitidos

    @Bean(BeanIds.AUTHENTICATION_MANAGER)
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }
}
```

Utilizamos a constante BeanIds.AUTHENTICATION_MANAGER para que possamos registrar o bean com o devido nome: authenticationManager .

Voltando ao nosso controller, podemos iniciar a autenticação através do AuthenticationManager . Como o método authenticate() depende de um objeto Authentication para funcionar, vamos precisar construir este objeto com os dados de login. Authentication é uma interface e devemos prover uma implementação. O Spring Security possui a classe UsernamePasswordAuthenticationToken que é a implementação mais simples de Authentication e que necessita apenas das informações do username (no nosso caso, o email) e o password:

```
@RestController
@RequestMapping("/api/auth")
public class UserAuthenticationController {

    @Autowired
    private AuthenticationManager authManager;

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE)
    public void authenticate(
        @RequestBody LoginInputDto loginInfo) {

        Authentication authenticationToken =
            new UsernamePasswordAuthenticationToken(loginInfo.getEmail(),
                loginInfo.getPassword());

        Authentication authentication = authManager
            .authenticate(authenticationToken);
    }
}
```

Feita a autenticação, precisamos gerar um token de acesso para retornar ao cliente. É neste momento que as classes da biblioteca jjwt vão nos ajudar. Ela possui a classe Jwts que é uma fábrica para criar instâncias de JWT:

```

Jwts.builder()
    .setIssuer("Alura Fórum API")
    .setSubject("1")
    .setIssuedAt(now)
    .setExpiration(expiration)
    .signWith(SignatureAlgorithm.HS256, "segredo")
    .compact();

```

O método `builder()` retorna uma intância de `JwtBuilder` para ser configurada para a criação de um JWT. Os métodos `setIssuer()`, `setSubject()`, `setIssuedAt()` e `setExpiration()` definem emissor, assunto, data de emissão e data de expiração, respectivamente. O Método `signWith()` define o algoritmo e devemos passar um chave secreta que será utilizada para assinar o *token*. Por fim, o método `compact()` constrói o JWT de acordo com o padrão que discutimos anteriormente.

Podemos isolar a contrução de um JWT em um método `generateToken()` dentro de uma classe responsnsável por gerenciar o token:

```

@Component
public class TokenManager {

    @Value("${alura.forum.jwt.secret}")
    private String secret;

    @Value("${alura.forum.jwt.expiration}")
    private long expirationInMillis;

    public String generateToken(Authentication authentication) {

        User user = (User) authentication.getPrincipal();

        final Date now = new Date();
        final Date expiration = new Date(now.getTime() +
            this.expirationInMillis);

        return Jwts.builder()
            .setIssuer("Alura Fórum API")
            .setSubject(Long.toString(user.getId()))
            .setIssuedAt(now)
            .setExpiration(expiration)
            .signWith(SignatureAlgorithm.HS256, this.secret)
            .compact();
    }
}

```

Note que utilizaremos o `id` do usuário como `subject`. No documento de referência [RFC 7519](#) do padrão JWT, o `subject` é definido como a informação que identifica um usuário e que deve ser único no contexto do emissor. Por este motivo é muito comum utilizar o `id` como `subject`.

Aqui também poderíamos acrescentar o `email`, `nome` e outros atributos de um usuário da aplicação, mas não é recomendável por ser tratar de informações sensíveis. No próprio site [jwt.io](#) é defendido como vantagem o fato de guardar as informações de um usuário no *token* para evitar idas ao banco de dados para recuperá-las. Sem dúvida, poupar idas ao banco de dados é uma grande vantagem, desde que não afete na segurança da aplicação. Neste curso, vamos optar por guardar apenas o `id` por

entendermos que outras informações como o email são sensíveis para trafegar com o token.

Outra novidade do código acima é a anotação `@Value` que é bastante utilizada para definir um valor padrão a um atributo de classe definindo uma chave que deve existir no arquivo `application.properties` com seu respectivo valor de inicialização:

```
alura.forum.jwt.secret = minha-chave-secreta
alura.forum.jwt.expiration = 604800000
```

Dessa forma, os parâmetros `secret` e `expirationInMillis` vão iniciar com os valores `minha-chave-secreta` e `604800000` (que corresponde ao período de um semana em milisegundos), respectivamente. Também é possível receber esses valores via variáveis de ambiente ou argumentos de linha de comando - evitando deixar a chave secreta exposta no arquivo `application.properties`.

Voltando ao nosso `controller`, pediremos o `TokenManager` injetado e invocaremos o método `generateToken()` após a autenticação do usuário:

```
@RestController
@RequestMapping("/api/auth")
public class UserAuthenticationController {

    // Auth Manager omitido

    @Autowired
    private TokenManager tokenManager;

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE)
    public void authenticate(
        @RequestBody LoginInputDto loginInfo) {

        Authentication authenticationToken =
            new UsernamePasswordAuthenticationToken(loginInfo.getEmail(),
                loginInfo.getPassword());

        Authentication authentication = authManager
            .authenticate(authenticationToken);

        String jwt = tokenManager.generateToken(authentication);
    }
}
```

Agora será preciso retornar um resposta. Novamente usaremos a estratégia do DTO:

```
public class AuthenticationTokenOutputDto {

    private String tokenType;
    private String token;

    public AuthenticationTokenOutputDto(String tokenType, String token) {
        this.tokenType = tokenType;
        this.token = token;
    }

    public String getToken() {
        return token;
    }
}
```

```

    public String getTokenType() {
        return tokenType;
    }
}

```

Criaremos um tipo para o *token* já que podemos optar ou não por usar chaves assimétricas, por exemplo. O mais comum é usar o tipo `Bearer` que significa **portador**. Isso quer dizer que qualquer portador de um *token* gerado pela aplicação poderá usá-lo como um *token* válido. Note que neste caso não será exigido qualquer tipo de informação sobre o portador. Outro exemplo é o `_sign_type_` onde o *token* é gerado e mantido com uma chave secreta para encriptar o conteúdo e apenas pode ser usado por portadores capazes de decodificá-lo.

Também é possível usar implementações próprias e criar um tipo de *token*, mas não faz muito sentido já que sua aplicação vai ter que manter uma documetação própria apenas para isso ao invés de usar uma implementação padrão que facilita a comunicação entre as aplicações cliente e servidor.

```

@RestController
@RequestMapping("/api/auth")
public class UserAuthenticationController {

    // Auth Manager omitido

    @Autowired
    private TokenManager tokenManager;

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
                 produces = MediaType.APPLICATION_JSON_VALUE)
    public AuthenticationTokenOutputDto authenticate(
        @RequestBody LoginInputDto loginInfo) {

        Authentication authenticationToken =
            new UsernamePasswordAuthenticationToken(loginInfo.getEmail(),
                loginInfo.getPassword());

        Authentication authentication = authManager
            .authenticate(authenticationToken);

        String jwt = tokenManager.generateToken(authentication);

        AuthenticationTokenOutputDto tokenResponse =
            new AuthenticationTokenOutputDto("Bearer", jwt);

        return tokenResponse;
    }
}

```

Acontece que a autenticação pode falhar e devemos tratar este caso e devolver o *status code* 400 (*bad request*). Se a aplicação receber email e/ou senha inválidos, o método `authenticate()` de `AuthenticationManager` vai lançar a exceção `AuthenticationException`. Precisamos incluir um bloco `try/catch`:

```

@RestController
@RequestMapping("/api/auth")

```

```

public class UserAuthenticationController {

    // Auth Manager omitido

    @Autowired
    private TokenManager tokenManager;

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
        produces = MediaType.APPLICATION_JSON_VALUE)
    public AuthenticationTokenOutputDto authenticate(
        @RequestBody LoginInputDto loginInfo) {

        Authentication authenticationToken =
            new UsernamePasswordAuthenticationToken(loginInfo.getEmail(),
                loginInfo.getPassword());

        try {
            Authentication authentication = authManager
                .authenticate(authenticationToken);

            String jwt = tokenManager.generateToken(authentication);

            AuthenticationTokenOutputDto tokenResponse =
                new AuthenticationTokenOutputDto("Bearer", jwt);

            return tokenResponse;
        } catch(AuthenticationException e){
            // o que retornar aqui ??
        }
    }
}

```

Neste caso, para ajudar na construção de uma resposta para o cliente, o Spring MVC possui a classe `ResponseEntity` que oferece mais flexibilidade pra montar uma resposta com *status code*, *body*, *header*, etc...

```

@RestController
@RequestMapping("/api/auth")
public class AuthenticationController {

    @Autowired
    private AuthenticationManager authManager;

    @Autowired
    private TokenManager tokenManager;

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
        produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<AuthenticationTokenOutputDto> authenticate(
        @RequestBody LoginInputDto inputDto) {

        Authentication authenticationToken =
            new UsernamePasswordAuthenticationToken(inputDto.getEmail(),
                inputDto.getPassword());

        try{
            Authentication authentication =
                authManager.authenticate(authenticationToken);

            String jwt = tokenManager.generateToken(authentication);

```

```

        AuthenticationTokenOutputDto tokenResponse =
            new AuthenticationTokenOutputDto("Bearer", jwt);

        return ResponseEntity.ok(tokenResponse);
    } catch (AuthenticationException e) {
        return ResponseEntity.badRequest().build();
    }
}
}

```

A classe `ResponseEntity` possui vários método estáticos que representam o *status code* do protocolo HTTP. Por exemplo, se a autenticação for processada com sucesso, usamos o método `ok` que vai definir o *status code* como 200 (*ok*) e passamos nosso DTO como parâmetro para ser adicionado no *body* da *resposnse*. Caso falhe, usamos o método `badRequest()` que retornará o *status code* 400 (*bad request*) com o corpo vazio (por este motivo precisamo chamar o método `build()` em seguida).

Para que possamos acessar o método `authenticate()` que responde pelo *endpoint* `/api/auth`, vamos precisar liberar esse *endpoint* modificando nossa configuração inicial do método `configure()` que recebe um `HttpSecurity`:

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers(HttpMethod.GET, "/api/topics/**").permitAll()
        .antMatchers("/api/auth/**").permitAll()
        .anyRequest().authenticated()
    .and()
        .formLogin()
    .and()
        .httpBasic();
}

```

Ótimo! Agora nosso *controller* já é capaz de autenticar um usuário e gerar um *token* de acesso. Mas ainda precisamos modificar algumas configurações de segurança já que a aplicação ainda funciona com a tela de login do Security e guarda usuários em um objeto `HttpSession`.

Precisamos, ainda, excluir as chamadas dos métodos `formLogin()` e `httpBasic()` já que não usaremos mais a tela de login padrão do Security. Vamos desabilitar o uso de sessão pela aplicação através do método `sessionManagement()` definindo a política de criação de sessão como `STATELESS` e vamos desabilitar também o `csrf`, já que trabalhamos sobre uma arquitetura que abstrai essa proteção.

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers(HttpMethod.GET, "/api/topics/**").permitAll()
        .antMatchers("/api/auth/**").permitAll()
        .anyRequest().authenticated()
    .and()
        .csrf().disable()
        .sessionManagement()
}

```

```

        .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
}

```

Note que passamos `STATELESS` que significa que em nenhum momento a aplicação vai criar uma sessão. Outras opções são `ALWAYS` (sempre), `IF_REQUIRED` (apenas se requerida) e `NEVER` (aqui pode confundir já que o Spring nunca irá criar uma sessão, mas não vai impedir que uma criada pelo desenvolvedor seja usada).

Como a requisição será feita pela aplicação cliente sem cookies (a não ser que aplicação garanta seu envio), o Spring Security vai rejeitar a requisição. Uma maneira de garantir que a requisição seja processada é garantir que um `CorsFilter` seja processado antes. É possível integrar um `CorsFilter` ao Security provendo um `CorConfigurationSource`, usando a função `cors()`:

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers(HttpMethod.GET, "/api/topics/**").permitAll()
        .antMatchers("/api/auth/**").permitAll()
        .anyRequest().authenticated()
    .and()
        .cors();
    .and()
        .csrf().disable()
        .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
}

```

Acontece que já possuímos uma configuração para política de CORS na aplicação, configurada a partir do Spring MVC - nossa classe `CorsConfiguration`. Caso o Security não encontre um implementação de `CorConfigurationSource` na aplicação, ele vai usar a configuração de CORS provida pelo `SpringMVC`.

Agora, ao rodar a aplicação cliente e fazer a autenticação pela requisição `/api/auth` com dados válidos, a aplicação retornará um *token* de acesso ao cliente seguindo o padrão JWT.

E para que seja possível o teste utilizando o Swagger, precisamos liberar o acesso para `/swagger-ui.html`. Além disso, será necessário liberar o acesso para todo o conteúdo estático utilizado para renderizar a página (como arquivos javascript e css). A classe `WebSecurityConfigurerAdapter` possui outro método `configure()` que recebe um objeto do tipo `WebSecurity` que representa a cadeia de filtros do Spring Security e é a maneira mais fácil de dizer para o Security ignorar alguns padrões de url, não exigindo autenticação. Através do método `ignoring()` seguido de `antMatchers()`, podemos passar algumas urls que serão ignoradas por toda a cadeia de filtros do Security:

```

@Override
public void configure(WebSecurity web) throws Exception {
    web.ignoring().antMatchers("/**.html", "/v2/api-docs",
        "/webjars/**", "/configuration/**",
        "/swagger-resources/**");
}

```

Assim, será possível utilizar o `Swagger` como antes, sem o `Security` exigir autenticação. Nos exercícios utilizaremos o `Swagger` para testar nosso novo `controller` e gerar o `token`.

8.13 EXERCÍCIOS: UTILIZANDO JWT COM SPRING SECURITY

Neste momento a aplicação autentica um usuário pela tela de login padrão do Spring Security. Vamos mudar este comportamento e criar um *endpoint* para realizar a autenticação do nosso cliente usando um *Json Web Token*.

1. Primeiro, no arquivo `pom.xml`, inclua a dependência da lib que traz a implementação do JWT:

```
<dependencies>  
    <!-- outras dependências -->  
  
    <dependency>  
        <groupId>io.jsonwebtoken</groupId>  
        <artifactId>jjwt</artifactId>  
        <version>0.9.1</version>  
    </dependency>  
  
</dependencies>
```

2. Crie a classe `UserAuthenticationController` no pacote `br.com.alura.forum.security.controller`. Anote a classe com `@RestController` e defina o mapeamento para a URI `/api/auth`:

```
@RestController  
@RequestMapping("/api/auth")  
public class UserAuthenticationController {  
  
}
```

Não esqueça de liberar a URI `/api/auth/` no método `configure()` da classe `SecurityConfiguration` para permitir a tentativa de autenticação de um usuário. Como não usaremos mais a autenticação padrão do Spring Security, apague a parte da configuração referente à página de login e acrescente a configuração que desabilita o uso da sessão para armazenar dados de usuários. Aproveite e desabilite também o `csrf`, já que trabalhamos sobre uma arquitetura que abstrai essa proteção:

```
@Configuration  
@EnableWebSecurity  
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {  
  
    // código anterior omitido  
  
    @Override  
    protected void configure(HttpSecurity http) throws Exception {  
        http.authorizeRequests()  
            .antMatchers(HttpMethod.GET, "/api/topics/**").permitAll()  
            .antMatchers("/api/auth/**").permitAll()  
            .anyRequest().authenticated()
```

```

        .and()
        .cors()
    .and()
        .csrf().disable()
    .sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS);
}
}

```

3. Ainda no *Controller*, crie o método `authenticate()` , que deverá receber um `LoginInputDto` representando os dados do login (email e password), e devolver um `AuthenticationTokenOutputDto` representando a resposta do nosso *JWT*.

```

@RestController
@RequestMapping("/api/auth")
public class UserAuthenticationController {

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
                  produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<AuthenticationTokenOutputDto> authenticate(
        @RequestBody LoginInputDto loginInfo) {

    }
}

```

4. Nosso código ainda não compila. Precisamos criar a classe `LoginInputDto` para receber as credenciais do usuário, e a classe `AuthenticationTokenOutputDto` para modelar o JSON de resposta do *JWT*:

- No pacote `br.com.alura.forum.security.controller.dto.input` , crie a classe `LoginInputDto` :

```

public class LoginInputDto {

    private String email;
    private String password;

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public UsernamePasswordAuthenticationToken build() {
        return new UsernamePasswordAuthenticationToken(this.email, this.password);
    }
}

```

Repare que o método `build()` retorna um `UsernamePasswordAuthenticationToken`, classe do Spring Security desenhada para representar os dados de autenticação (`username` e `password`) e que implementa a interface `Authentication`.

- No pacote `br.com.alura.forum.security.controller.dto.output`, crie a classe `AuthenticationTokenOutputDto`:

```
public class AuthenticationTokenOutputDto {  
  
    private String tokenType;  
    private String token;  
  
    public AuthenticationTokenOutputDto(String tokenType, String token) {  
        this.tokenType = tokenType;  
        this.token = token;  
    }  
  
    public String getToken() {  
        return token;  
    }  
  
    public String getTokenType() {  
        return tokenType;  
    }  
}
```

5. De volta ao `UserAuthenticationController`, vamos construir um objeto do tipo `UserPasswordAuthenticationToken` usando do método `build()` do nosso DTO. Dessa maneira é possível chamar o método `authenticate()` do `AuthenticationManager` que autentica os dados e retorna um `Authentication`. Este objeto será usado para gerar o *JWT* que representa a autenticação do usuário.

```
@RestController  
@RequestMapping("/api/auth")  
public class UserAuthenticationController {  
  
    @Autowired  
    private AuthenticationManager authManager;  
  
    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,  
                 produces = MediaType.APPLICATION_JSON_VALUE)  
    public ResponseEntity<AuthenticationTokenOutputDto> authenticate(  
        @RequestBody LoginInputDto loginInfo) {  
  
        UsernamePasswordAuthenticationToken authenticationToken =  
            loginInfo.build();  
  
        Authentication authentication = authManager  
            .authenticate(authenticationToken);  
  
    }  
}
```

Não esqueça de pedir para o Spring injetar o `AuthenticationManager` e importe a interface `Authentication` do pacote `org.springframework.security`.

6. Agora será preciso criar o *token* de acesso. No pacote `br.com.alura.forum.security.jwt`, crie a classe `TokenManager`, com um método `generateToken()` recebendo um `authentication`.

```
@Component
public class TokenManager {

    @Value("${alura.forum.jwt.secret}")
    private String secret;

    @Value("${alura.forum.jwt.expiration}")
    private long expirationInMillis;

    public String generateToken(Authentication authentication) {

        User user = (User) authentication.getPrincipal();

        final Date now = new Date();
        final Date expiration = new Date(now.getTime() +
            this.expirationInMillis);

        return Jwts.builder()
            .setIssuer("Alura Fórum API")
            .setSubject(Long.toString(user.getId()))
            .setIssuedAt(now)
            .setExpiration(expiration)
            .signWith(SignatureAlgorithm.HS256, this.secret)
            .compact();
    }
}
```

A anotação `@Value` deve ser importada do pacote `org.springframework.beans`. De volta ao *Controller*, podemos utilizar a classe `TokenManager`, para obter o *JWT* e em seguida, enviar a resposta ao usuário:

```
@RestController
@RequestMapping("/api/auth")
public class UserAuthenticationController {

    // Auth Manager omitido

    @Autowired
    private TokenManager tokenManager;

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
        produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<AuthenticationTokenOutputDto> authenticate(
        @RequestBody LoginInputDto loginInfo) {

        UsernamePasswordAuthenticationToken authenticationToken =
            loginInfo.build();

        try {
            Authentication authentication = authManager
                .authenticate(authenticationToken);

            String jwt = tokenManager.generateToken(authentication);

            AuthenticationTokenOutputDto tokenResponse =
                new AuthenticationTokenOutputDto("Bearer", jwt);
        }
    }
}
```

```

        return ResponseEntity.ok(tokenResponse);

    } catch (AuthenticationException e) {
        return ResponseEntity.badRequest().build();
    }
}
}
}

```

Atente para o uso do bloco `try/catch` tratando o caso em que a autenticação falha, e devolvendo a resposta com o *HTTP Status Code 400 Bad Request*. A exceção `AuthenticationException` deve ser importada do pacote `org.springframework.security`.

7. Acrescente os valores de `secret` e `expirationInMillis` no arquivo `application.properties`:

```

alura.forum.jwt.secret = rm'!@N=Ke!~p8VTA2ZRK~nMDQX5Uvm!m'D&{@Vr?G;2?XhbC:Qa#9#eMLN\}x3?JR3.2zr~v)gYF^8\:8>:XFB:Ww75N/emt9Yj[bQMNCWW\J?N, nvH.<2\.r~w]*e~vgak)X"v8H`MH/7"2E` , ^k@n<vE-wD3g9JWPY;CrY* .Kd2_D]=><D?YhBaSua5hW%{2}_FVXzb9`8FH^b[X3jzVER&:jw2<=c38=>L/zBq`}C6tT*ccSVC^c] -L}&/
alura.forum.jwt.expiration = 604800000

```

Para facilitar o exercício, você pode usar qualquer texto como `secret`, ou caso queira utilizar o segredo acima, copie o conteúdo do arquivo `secret.txt`, presente na pasta de arquivos do curso.

8. A autenticação ainda não funciona, já que o Spring não consegue resolver a dependência do `AuthenticationManager`. Sobrescreva o método `authenticationManagerBean()` na classe `SecurityConfiguration`, anotando-o com `@Bean`, para que o framework gerencie este objeto e permita sua injeção em outros pontos da aplicação:

```

@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    // código anterior omitido

    @Override
    @Bean(BeanIds.AUTHENTICATION_MANAGER)
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }
}

```

9. Para testar se tudo funciona com o login, podemos usar mais uma vez o `Swagger UI`. Para isso precisamos adicionar as configurações necessárias para, no contexto de segurança, ignorar as requisições para os recursos utilizados pela ferramenta.

Na classe `SecurityConfiguration`, adicione a sobrescrita do método `configure(WebSecurity web)` com a seguinte implementação:

```

@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    // código anterior omitido

    @Override
    public void configure(WebSecurity web) throws Exception {
        web.ignoring().antMatchers("/**.html", "/v2/api-docs",
            "/webjars/**", "/configuration/**",
            "/swagger-resources/**");
    }
}

```

10. Agora, acesse a página do Swagger UI pelo endereço `http://localhost:8080/swagger-ui.html` e clique na seção referente ao **user-authentication-controller**

The screenshot shows a browser window displaying the Alura Forum API Documentation. The title bar says 'localhost:8080/swagger-ui.html#/user-authentication-controller'. The main content area has a green header with the 'swagger' logo. Below it, the title 'Alura Forum API Documentation' is displayed with a '1.0' badge. A note says '[Base URL: localhost:8080 /]' and provides a link 'http://localhost:8080/v2/api-docs'. A message encourages users to 'Tente enviar algum request ;)' and links to 'Alura - Website' and 'Send email to Alura'. The interface is divided into sections for 'topic-controller' and 'user-authentication-controller'. Under 'user-authentication-controller', there is a 'User Authentication Controller' section with a 'POST /api/auth authenticate' button. At the bottom, there is a 'Models' section.

Figura 8.11: Acessando swagger

Em seguida clique em `POST /api/auth` para verificar os detalhes e a documentação do *endpoint*. Clique em `Try it out`, modifique os dados de acesso inserindo email e password de um usuário em sua base de dados, e em seguida clique em `Execute`.

The screenshot shows the Swagger UI interface for a POST request to the '/api/auth' endpoint. The 'Parameters' section includes a required parameter 'loginInfo' with a JSON example: { "email": "aluno@gmail.com", "password": "123456" }. The 'Content type' dropdown is set to 'application/json'. A large blue 'Execute' button is prominent at the bottom.

Figura 8.12: Swagger post auth

Veja que nossa aplicação já é capaz de devolver o *token* para o usuário.

The screenshot shows the execution results of the POST request. It includes a 'Curl' section with the command: curl -X POST "http://localhost:8080/api/auth" -H "accept: application/json" -H "Content-Type: application/json" -d "{ \"email\": \"aluno@gmail.com\", \"password\": \"123456\" }". Below it is a 'Request URL' field with the value 'http://localhost:8080/api/auth'. The 'Server response' section shows a status code of 200. The 'Response body' contains a JSON token: { "tokenType": "Bearer", "token": "eyJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJBbHVsYSBGw7Nydw0gQVBJIiwic3ViIjoiNCIsImhdCI6MTUzNzk5NDYzNiwiZXhwIjoxNTM4NTk5NDM2fQ.-1PT-GpU8cnnfqpQY-d1ipv7Kme2qsQNzuRpGhYmbjluw" }. The 'Response headers' section lists standard HTTP headers like Cache-Control, Content-Type, Date, Expires, Pragma, and Transfer-Encoding.

Figura 8.13: Swagger token

1. (Opcional) Caso queria testar a aplicação cliente efetuando login, no terminal, acesse a pasta do

projeto da aplicação cliente `Desktop/fj27-alura-forum-client/`, e execute o comando `git checkout 08ProtegendoNossaAplicacao`, para atualizar seu estado.

Com o *client* e a *API* rodando com os novos recursos, teste novamente o funcionamento da app cliente recarregando a página (`http://localhost:3000/login`) e digite os dados de acesso para efetuar login.

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

8.14 VALIDANDO E AUTENTICANDO O TOKEN

A aplicação já é capaz de gerar um *token* para o cliente. Agora vamos implementar a lógica que valida esse *token* de acesso enviado pela aplicação cliente e que a permite acessar um recurso protegido.

Para validar um *token*, utilizaremos novamente os recursos provados pela biblioteca `jjwt`. Infelizmente não existe um método pronto que valida um *token*, mas a biblioteca provê uma maneira de decodificar um *token* JWT para ter acesso a suas informações. Caso o *token* não seja válido, essa decodificação falha lançando a exceção `JwtException` - ou `IllegalArgumentException` no caso do *token* chegar nulo.

Novamente faremos uso da classe `Jwts` para decodificar o token através do método `parser()`, passando nossa *secret key* e o próprio *token*:

```
Jwts.parser().setSigningKey("palavra secreta").parseClaimsJws(jwt);
```

Podemos criar um método na classe `TokenManager` que vai tentar decodificar o *token*. Caso obtenha sucesso, esse será um *token* válido:

```
@Component
```

```

public class TokenManager {

    // código anterior omitido

    public boolean isValid(String jwt) {
        try {
            Jwts.parser().setSigningKey(this.secret).parseClaimsJws(jwt);
            return true;
        } catch (JwtException | IllegalArgumentException e) {
            return false;
        }
    }
}

```

Mas em que momento devemos chamar este método para validar o token? O Spring Security possui uma cadeia de filtros previamente configurada. Um filtro é executado antes da `DispatcherServlet` e, portanto, antes de nossos `controllers`. Quando usamos sua tela de login padrão, o filtro utilizado para processar a autenticação é o `UsernamePasswordAuthenticationFilter`. Não queremos mais usar este filtro e suas chamadas subsequentes mas um outro, implementado por nós, que vai validar um *token* de acesso a cada requisição. Mas antes de entrarmos nos detalhes de quando e como este filtro será chamado, vamos começar a desenvolvê-lo:

```

public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                    FilterChain chain) throws ServletException, IOException {

        // valida o token

        chain.doFilter(request, response);

    }
}

```

Nosso filtro herda de `OncePerRequestFilter` que, como seu nome sugere, faz com que seu método `doFilterInternal()` seja executado apenas uma vez por requisição. Antes de validar, será necessário pegar o *token* da requisição. Um *token* de autorização é enviado através do *header* (cabecalho) da requisição:

```
{
    "Authorization": Bearer <token>
}
```

Então pegaremos o *token* através do objeto `HttpServletRequest`:

```
String bearerToken = request.getHeader("Authorization");
```

O Spring possui a classe `StringUtils` que é uma classe utilitária que ajuda no trato com *strings*. Como o *token* vem no formato "Bearer ", precisamos pegar o conteúdo desta *string* a partir da 7 posição:

```
if (StringUtils.hasText(bearerToken) && bearerToken.startsWith("Bearer ")) {
    return bearerToken.substring(7, bearerToken.length());
}
```

No final, nosso filtro ficará assim:

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                    FilterChain chain) throws ServletException, IOException {

        String jwt = getTokenFromRequest(request);

        chain.doFilter(request, response);
    }

    private String getTokenFromRequest(HttpServletRequest request) {
        String bearerToken = request.getHeader("Authorization");

        if (StringUtils.hasText(bearerToken) && bearerToken.startsWith("Bearer "))
            return bearerToken.substring(7, bearerToken.length());

        return null;
    }
}
```

Agora que já pegamos o *token* da requisição, vamos precisar validá-lo através do método `isValid()` que criamos anteriormente na classe `TokenManager`. Aqui é possível anotar o filtro com `@Component` para que o Spring injete o `TokenManager` mas optamos por outra abordagem - para dificultar a injeção deste filtro em outros pontos da aplicação, optamos por registrar nosso filtro na cadeia de filtros do Spring na mão (passo que faremos mais adiante). Então criaremos o construtor com a dependência de um `TokenManager`:

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private TokenManager tokenManager;

    public JwtAuthenticationFilter(TokenManager tokenManager) {
        this.tokenManager = tokenManager;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response, FilterChain chain)
        throws ServletException, IOException {

        String jwt = getTokenFromRequest(request);

        if (tokenManager.isValid(jwt)) {

            // o que fazer aqui?

            chain.doFilter(request, response);
        }
    }

    // método getTokenFromRequest() omitido
}
```

Se o *token* for válido, devemos seguir com o fluxo da aplicação. Além disso, vamos precisar

recuperar o usuário logado quando inserir um novo tópico no futuro (já que precisaremos do dono do tópico para criar um). Portanto vamos querer recuperar o usuário em algum *controller* durante o escopo de uma requisição específica.

Por padrão, a classe `SecurityContextHolder` do Spring Security usa um `ThreadLocal` para armazenar os detalhes do usuário, o que significa que o contexto de segurança está sempre disponível para métodos na mesma *thread* de execução, mesmo se o contexto de segurança não for explicitamente passado como um argumento para esses métodos. Usar um `ThreadLocal` dessa maneira é bastante seguro se for tomado cuidado para limpar o encadeamento depois que a requisição for processada. O Spring Security cuida disso automaticamente e não há necessidade de se preocupar com essa questão.

As informações de um usuário são representadas por um `Authentication`, definido no método `setAuthentication()` da classe `SecurityContext` que representa o contexto de segurança. O `SecurityContext` é inicializado e definido dentro de um `SecurityContextHolder` em um dos filtros da cadeia de filtros do Security que coneceremos com mais detalhes adiante. Portanto, o `SecurityContextHolder` contém o `SecurityContext` que define informações mínimas de segurança associada a *thread* corrente.

Vamos precisar colocar um `Authentication` representando o usuário logado no `SecurityContext`:

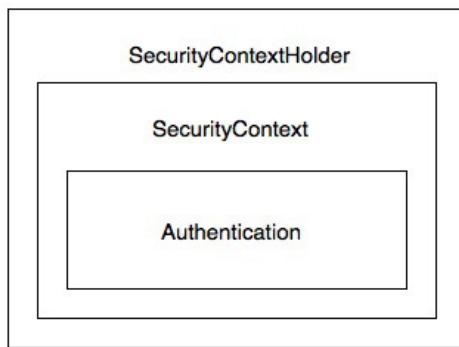


Figura 8.14: SecurityContextHolder

O `SecurityContext` irá manter as informações em uma *thread local*, facilitando sua recuperação em outros pontos do fluxo.

Para obter um `SecurityContext`, utilizamos o método estático `getContext()` da classe `SecurityContextHolder`:

```
SecurityContextHolder.getContext();
```

E definimos o usuário corrente através do método `setAuthentication()`:

```
SecurityContextHolder.getContext().setAuthentication(authentication);
```

Então, caso o *token* seja válido, precisamos construir um `Authentication` através das informações do usuário extraídas do próprio *token*:

```
@Override  
protected void doFilterInternal(HttpServletRequest request,  
    HttpServletResponse response, FilterChain chain)  
    throws ServletException, IOException {  
  
    String jwt = getTokenFromRequest(request);  
  
    if (tokenManager.isValid(jwt)) {  
  
        // extrair do token as informações do usuário  
        // construir um Authentication  
        // colocar o usuário no SecurityContext  
  
        chain.doFilter(request, response);  
    }  
}
```

Para pegar as informações do usuário, precisamos decodificar o *token* com o método `parser()` - o mesmo que utilizamos para validá-lo, mas agora com o intuito de coletar suas informações. Como colocamos apenas a informação do `id` do usuário no *token*, devemos recuperá-lo pelo seu `id`. Vamos criar este método na classe `TokenManager`:

```
@Component  
public class TokenManager {  
  
    // código anterior omitido  
  
    public Long getUserIdFromToken(String jwt) {  
        Claims claims = Jwts.parser().setSigningKey(this.secret)  
            .parseClaimsJws(jwt).getBody();  
  
        return Long.parseLong(claims.getSubject());  
    }  
}
```

Aqui, diferente do método `isValid()`, foi necessário chamar o método `getBody()` para extrair seu conteúdo que retorna um objeto do tipo `Claims` - que representa as informações do *token*. Por fim, chamamos o método `getSubject()` para extrair o `id` e fazemos o parse para um tipo `Long`. Agora podemos invocar este método em nosso filtro:

```
@Override  
protected void doFilterInternal(HttpServletRequest request,  
    HttpServletResponse response, FilterChain chain)  
    throws ServletException, IOException {  
  
    String jwt = getTokenFromRequest(request);  
  
    if (tokenManager.isValid(jwt)) {  
  
        Long userId = tokenManager.getUserIdFromToken(jwt);  
  
        // construir um Authentication
```

```

    // colocar o usuário no SecurityContext
    chain.doFilter(request, response);
}
}

```

Com o `id` em mãos, podemos recuperar as informações do usuário no banco pelo `id`. Criamos o método `findById()` em `UserRepository` e em seguida um método em `UsersService` que fará esta lógica de carregar o usuário pelo seu `id`:

```

@Service
public class UsersService implements UserDetailsService {

    // código anterior omitido

    public UserDetails loadUserById(Long userId) {
        Optional<User> possibleUser = userRepository.findById(userId);

        return possibleUser.orElseThrow(
            () -> new UsernameNotFoundException("Não foi possível encontrar
                o usuário com id: " + userId));
    }
}

```

Agora é necessário construir um `Authentication` para colocá-lo no `SecurityContext`. Podemos usar novamente a implementação `UsernamePasswordAuthenticationToken` e a sobrecarga do construtor que também recebe as *authorities* que serão usadas pelo Security dar acesso as rotas de maneira correta:

```

public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private TokenManager tokenManager;
    private UsersService usersService;

    public JwtAuthenticationFilter(TokenManager tokenManager, UsersService usersService) {
        this.tokenManager = tokenManager;
        this.usersService = usersService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
        HttpServletResponse response, FilterChain chain)
        throws ServletException, IOException {

        String jwt = getTokenFromRequest(request);

        if (tokenManager.isValid(jwt)) {

            Long userId = tokenManager.getUserIdFromToken(jwt);
            UserDetails userDetailsService = usersService.loadUserById(userId);

            UsernamePasswordAuthenticationToken authentication =
                new UsernamePasswordAuthenticationToken(userDetailsService,
                    null, userDetailsService.getAuthorities());

            SecurityContextHolder.getContext().setAuthentication(authentication);
        }
    }
}

```

```
        chain.doFilter(request, response);
    }

    // método getTokenFromRequest() omitido
}
```

Note que também foi preciso garantir a dependência de um `UserService` no construtor de nosso filtro.

Nosso filtro está pronto mas em que momento ele será chamado?

8.15 CADEIA DE FILTROS DO SECURITY

Você deve se lembrar que um filtro é registrado no `Servlet Container` - nossa aplicação está utilizando o Tomcat. Ao utilizar filtros, você precisa declará-los no arquivo de configuração `web.xml` ou eles serão ignorados pelo `Servlet Container`.

O Spring Security possui uma cadeia de filtros (*filter chain*) onde cada filtro tem uma responsabilidade particular. A `ordem` desses filtros importa já que há dependências entre eles.

No Spring Security, as classes de filtro também são *beans* do Spring definidos no `ApplicationContext` e, portanto, podem aproveitar as instalações avançadas de injeção de dependência e interfaces de ciclo de vida do Spring. A classe `DelegatingFilterProxy` do Spring fornece o link entre o `web.xml` e o contexto da aplicação.

O que a classe `DelegatingFilterProxy` faz é delegar os métodos do filtro para um *bean* que é obtido do contexto do Spring, o `FilterChainProxy`. Isso permite que o *bean* se beneficie do suporte do ciclo de vida do contexto da web do Spring e da flexibilidade de configuração.

A classe `FilterChainProxy` é vinculada a cadeia de filtros do `Servlet Container` e delega para a cadeia de filtros do Security as requisições que serão filtradas. Não é necessário configurar esta classe, a não ser que você deseje um controle mais fino da cadeia de filtros do Security. Em tempo de execução, o `FilterChainProxy` localizará o padrão de URI que corresponde à requisição e a lista de filtros especificados que serão aplicados - de acordo com as configurações contidas na classe `SecurityConfiguration`. Essa abordagem é particularmente útil para a implementação de filtros com necessidades complexas de configuração, permitindo aplicar a maquinaria completa de definição de *beans* no Spring para filtrar requisições.

A esquema abaixo ajuda a ilustrar o que acontece por baixo dos panos no ciclo dos filtros do Security quando chega uma requisição para um recurso protegido:

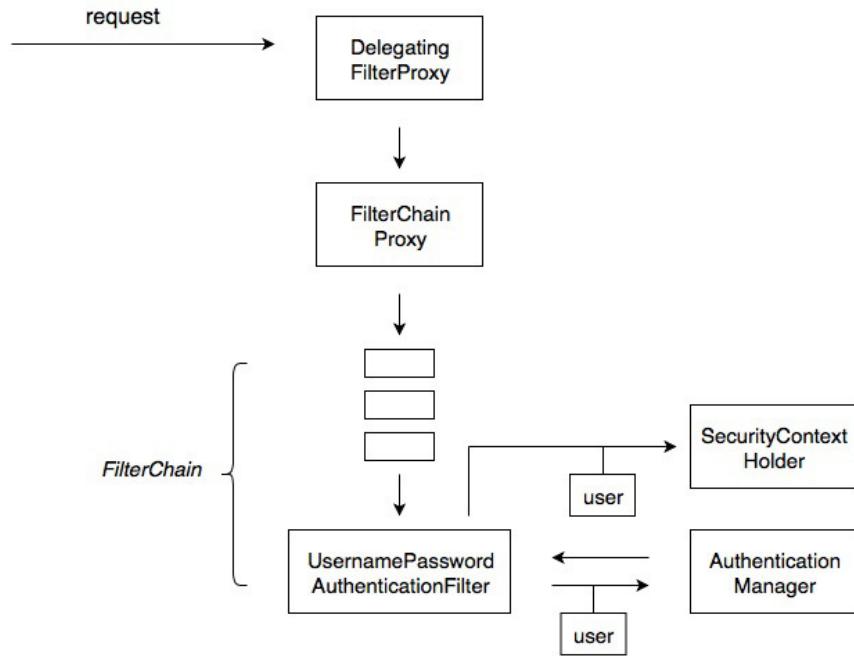


Figura 8.15: Filter Chain

Como dito anteriormente, o filtro da cadeia responsável por autenticar um usuário através da tela de login padrão do Security é o `UsernamePasswordAuthenticationFilter`. Ele faz exatamente o que nosso filtro faz, autentica um usuário através do `AuthenticationManager` e o coloca no contexto de segurança utilizando o `SecurityContextHolder`. Mas este filtro utiliza um `HttpSession` para guardar informações de usuários e não valida um *token* JWT!

Portanto, nosso filtro deve se executado antes deste filtro - lembre, a ordem é importante aqui! Para registrar nosso filtro na cadeia de filtros do Security antes de `UsernamePasswordAuthenticationFilter` e ser utilizado na autenticação de um usuário validando um *token* de acesso, podemos utilizar o método `addFilterBefore()` dentro do método `configure(HttpSecurity)` de nossa classe de configuração de segurança:

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    // outros atributos e métodos omitidos

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers(HttpMethod.GET, "/api/topics/**").permitAll()
            .antMatchers("/api/auth/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .cors()
            .and()
    }
}
```

```

        .csrf().disable()
        .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and()
            .addFilterBefore(new JwtAuthenticationFilter(tokenManager, usersService),
                UsernamePasswordAuthenticationFilter.class);
    }
}
}

```

Apesar de toda complexidade do Spring Security, o *framework* abstrai toda a parte da configuração pesada e facilita na configuração customizada de acordo com nossa regra de negócio. Você não precisa necessariamente entender todos os pormenores do que acontece mas ter um conhecimento mínimo do fluxo do Security ajuda bastante na hora de customizar as regras de segurança da aplicação, tanto na parte de autenticação quanto na parte de autorização. Recomendamos fortemente a leitura da [documentação do Spring Security](#) para um entendimento mais detalhado sobre este módulo.

8.16 ENVIANDO UM TOKEN PELA REQUISIÇÃO

No exercício vamos implementar nosso filtro e tentar acessar um rota segura enviando o *token* de acesso.

É possível utilizar a aplicação `curl` pelo terminal para processar O `curl` é uma ferramenta para processar requisições remotas em diversos protocolos. Conseguimos, com o comando abaixo, processar uma requisição enviando um *token* pelo cabeçalho:

```
curl -H "Authorization: Bearer <ACCESS_TOKEN_AQUI>" -X GET http://localhost:8080/
```

Mas para facilitar os testes dos *endpoints* podemos contar novamente com o auxílio do `Swagger UI` e, para facilitar ainda mais, vamos alterar suas configurações para adicionar um campo de texto relativo ao valor `header Authorization` a cada vez que quisermos utilizar a interface do `Swagger` para fazer requisições aos *endpoints* protegidos da nossa API enviando o *token* de acesso.

```

@Configuration
@EnableSwagger2
public class SwaggerConfiguration {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.basePackage("br.com.alura.forum"))
            .paths(PathSelectors.ant("/api/**"))
            .build()

        // algumas linhas de código omitidas

        .ignoredParameterTypes(User.class)
        .globalOperationParameters(
            Arrays.asList(
                new ParameterBuilder()
                    .name("Authorization")

```

```

        .description("Header para facilitar o envio do Authorization Bearer Token"
)
        .modelRef(new ModelRef("string"))
        .parameterType("header")
        .required(false)
        .build());
}

// código posterior omitido
}

```

O resultado no Swagger será como a figura abaixo:

Parameters	
Name	Description
Authorization string (header)	Header para facilitar o envio do Authorization Bearer Token Authorization - Header para facilitar o e

Figura 8.16: Swagger Authorization Header

Mas antes de tentarmos processar uma requisição acessando um recurso protegido, precisamos garantir uma resposta padrão em casos de erro de autenticação do *JWT*, quando, por exemplo, o *token* enviado for inválido ou já se encontrar expirado. Para isso, o Spring Security possui uma interface interna chamada `AuthenticationEntryPoint`, que idealiza um *handler* que deve gerenciar a autenticação a partir do momento que uma `AuthenticationException` é lançada no processamento do *filter chain*.

Vamos criar nossa implementação de `AuthenticationEntryPoint`, para responder o *status UNAUTHORIZED* quando a `AuthenticationException` for lançada ao tentar autenticar o *JWT*. Na classe `SecurityConfiguration` vamos adicionar a classe interna `JwtAuthenticationEntryPoint` sobrescrevendo o método `commence()`:

```

@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    // código anterior da classe omitido

    private static class JwtAuthenticationEntryPoint
        implements AuthenticationEntryPoint {

        @Override
        public void commence(HttpServletRequest request,
                             HttpServletResponse response,
                             AuthenticationException authException)
            throws IOException, ServletException {

            response.sendError(HttpServletRequest.SC_UNAUTHORIZED,
                               "Você não está autorizado a acessar esse recurso.");
        }
    }
}

```

```
}
```

Agora, quando uma `AuthenticationException` for lançada, nossa implementação de `AuthenticationEntryPoint` deve ser chamada. Para isso será preciso registrar nossa implementação no fluxo de segurança através do método `exceptionHandling()` dentro do método `configure(HttpSecurity)` que adicionará nosso *entry point* como *handler* das exceções de autenticação:

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http.authorizeRequests()  
        .antMatchers("/api/topics/**").permitAll()  
        .antMatchers("/api/auth/**").permitAll()  
        .anyRequest().authenticated()  
    .and()  
        // parte do código omitida  
    .and()  
        .addFilterBefore(new JwtAuthenticationFilter(tokenManager, usersService),  
                        UsernamePasswordAuthenticationFilter.class)  
    .exceptionHandling()  
        .authenticationEntryPoint(new JwtAuthenticationEntryPoint());  
}  
}
```

Dessa maneira, nossa aplicação vai devolver a resposta com *status* 401 quando um token inválido for enviado. Agora é possível gerar um *token* e tentar acessar um recurso protegido. Além do Swagger e a aplicação cliente é possível testar usando o `curl`



Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

8.17 EXERCÍCIOS - VALIDANDO E AUTENTICANDO O TOKEN

1. No pacote `br.com.alura.forum.security`, crie a classe `JwtAuthenticationFilter` que herda de `OncePerRequestFilter`, sobrescrevendo o método `doFilterInternal()`:

```

public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                    FilterChain chain) throws ServletException, IOException {
        ...
    }
}

```

2. Precisamos pegar o token no corpo da requisição. Para isso, crie o método privado `getTokenFromRequest()` e chame-o dentro do método `doFilterInternal()`:

```

public class JwtAuthenticationFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response,
                                    FilterChain chain) throws ServletException, IOException {
        String jwt = getTokenFromRequest(request);

        chain.doFilter(request, response);
    }

    private String getTokenFromRequest(HttpServletRequest request) {
        String bearerToken = request.getHeader("Authorization");

        if (StringUtils.hasText(bearerToken) && bearerToken.startsWith("Bearer "))
            return bearerToken.substring(7, bearerToken.length());

        return null;
    }
}

```

A classe `StringUtils` pertence ao Spring e deve ser importada do pacote `org.springframework.util`.

3. Agora é preciso checar se o *token* enviado pela aplicação cliente é válido. Na classe `TokenManager`, crie o método `isValid()` que retorna um `boolean` e que contém a regra de validação da assinatura e conversão do *token*. Aproveite para acrescentar o método `getUserIdFromToken()` que usaremos para recuperar o `id` do usuário a partir do *token*:

```

@Component
public class TokenManager {

    // código anterior omitido

    public boolean isValid(String jwt) {
        try {
            Jwts.parser().setSigningKey(this.secret).parseClaimsJws(jwt);
            return true;
        } catch (JwtException | IllegalArgumentException e) {
            return false;
        }
    }

    public Long getUserIdFromToken(String jwt) {
        Claims claims = Jwts.parser().setSigningKey(this.secret)
            .parseClaimsJws(jwt).getBody();
    }
}

```

```

        return Long.parseLong(claims.getSubject());
    }
}

```

4. Na classe `JwtAuthenticationFilter`, altere a implementação do método `doFilterInternal()` para verificar se o token é válido através do método `isValid()` do `TokenManager`.

Se o *token* for válido, o filtro deve recuperar o `id` do usuário e carregar um `UserDetails` utilizando a classe `UsersService`:

```

public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private TokenManager tokenManager;
    private UsersService usersService;

    public JwtAuthenticationFilter(TokenManager tokenManager, UsersService usersService) {
        this.tokenManager = tokenManager;
        this.usersService = usersService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response, FilterChain chain)
        throws ServletException, IOException {

        String jwt = getTokenFromRequest(request);

        if (tokenManager.isValid(jwt)) {

            Long userId = tokenManager.getUserIdFromToken(jwt);
            UserDetails userDetails = usersService.loadUserById(userId);
        }

        chain.doFilter(request, response);
    }
}

```

O código ainda não compila, pois falta criar o método `loadUserById()` na classe `UserDetails`:

```

@Service
public class UsersService implements UserDetailsService {

    // código anterior omitido

    public UserDetails loadUserById(Long userId) {
        Optional<User> possibleUser = userRepository.findById(userId);

        return possibleUser.orElseThrow(
            () -> new UsernameNotFoundException("Não foi possível encontrar o
                usuário com id: " + userId));
    }
}

```

E crie o método `findById()` na classe `UserRepository`:

```

public interface UserRepository extends Repository<User, Long> {

    // código anterior omitido

```

```
        Optional<User> findById(Long userId);  
    }
```

5. Ainda no método `doFilterInternal()`, crie um `UserPasswordAuthenticationToken` passando a representação do usuário (`UserDetails`) e adicione-o ao contexto de segurança do Spring Security:

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {  
  
    @Override  
    protected void doFilterInternal(HttpServletRequest request,  
                                    HttpServletResponse response, FilterChain chain)  
        throws ServletException, IOException {  
  
        String jwt = getTokenFromRequest(request);  
  
        if (tokenManager.isValid(jwt)) {  
  
            Long userId = tokenManager.getUserIdFromToken(jwt);  
            UserDetails userDetails = usersService.loadUserById(userId);  
  
            UsernamePasswordAuthenticationToken authentication =  
                new UsernamePasswordAuthenticationToken(userDetails,  
                null, userDetails.getAuthorities());  
  
            SecurityContextHolder.getContext().setAuthentication(authentication);  
        }  
  
        chain.doFilter(request, response);  
    }  
}
```

6. O filtro está pronto mas, para a autenticação funcionar, é preciso adicionar o filtro na cadeia do Spring Security. Modifique o método `configure(HttpSecurity security)` da classe `SecurityConfiguration` para garantir este comportamento.

Vamos adicionar nosso filtro imediatamente antes do filtro `UsernamePasswordAuthenticationFilter` do Spring Security que processa um tradicional envio de formulário de autenticação e intercepta por padrão a requisição do tipo `POST` para `/login`. Não esqueça de injetar o `TokenManager` para funcionar.

```
@Configuration  
@EnableWebSecurity  
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {  
  
    @Override  
    protected void configure(HttpSecurity http) throws Exception {  
        http.authorizeRequests()  
            .antMatchers(HttpMethod.GET, "/api/topics/**").permitAll()  
            .antMatchers("/api/auth/**").permitAll()  
            .anyRequest().authenticated()  
        .and()  
            .cors()  
        .and()  
            .csrf().disable()  
        .sessionManagement()  
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
```

```

        .and()
        .addFilterBefore(new JwtAuthenticationFilter(tokenManager, usersService),
                        UsernamePasswordAuthenticationFilter.class);
    }
}
}

```

7. Adicione a classe interna `JwtAuthenticationEntryPoint` sobrescrevendo o método `commence()` para o tratamento das exceções de autenticação:

```

@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    // código anterior da classe omitido

    private static class JwtAuthenticationEntryPoint
        implements AuthenticationEntryPoint {

        private static final Logger logger = LoggerFactory
            .getLogger(JwtAuthenticationEntryPoint.class);

        @Override
        public void commence(HttpServletRequest request,
                             HttpServletResponse response,
                             AuthenticationException authException)
            throws IOException, ServletException {

            logger.error("Um acesso não autorizado foi verificado. Mensagem: {}", authException.getMessage());

            response.sendError(HttpStatus.SC_UNAUTHORIZED,
                               "Você não está autorizado a acessar esse recurso.");
        }
    }
}

```

Importe `Logger` e `LoggerFactory` do pacote `org.slf4j` para facilitar com as mensagens de logs. Agora que temos nossa implementação criada, altere a implementação do método `configure(HttpSecurity)` para adicionar nosso *entry point* como *handler* das *exceptions* de autenticação:

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/api/topics/**").permitAll()
        .antMatchers("/api/auth/**").permitAll()
        .anyRequest().authenticated()
    .and()
        .cors()
    .and()
        .csrf().disable()
    .sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
    .and()
        .addFilterBefore(new JwtAuthenticationFilter(tokenManager, usersService),
                        UsernamePasswordAuthenticationFilter.class)
    .exceptionHandling()
        .authenticationEntryPoint(new JwtAuthenticationEntryPoint());
}

```

```
    }
}
```

8. Envie uma requisição para `/api/auth` com as informações de email e senha utilizando o Swagger para gerar o *token*, como fizemos no final do exercício anterior.
9. Copie o *token* gerado e faça uma requisição a um recurso protegido, como `http://localhost:8080/`. Para fazer a requisição podemos utilizar o comando `curl` no terminal:

```
curl -H "Authorization: Bearer <ACCESS_TOKEN_AQUI>" -X GET http://localhost:8080/
```

Perceba que a partir do envio do *JWT* nossa API confia que o portador dessa informação tem direito a acessar informações protegidas por autenticação.

10. Na classe `SwaggerConfiguration`, altere a implementação para adicionar suporte ao envio do *header*:

```
@Configuration
@EnableSwagger2
public class SwaggerConfiguration {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.basePackage("br.com.alura.forum"))
            .paths(PathSelectors.ant("/api/**"))
            .build()

            // algumas linhas de código omitidas

            .ignoredParameterTypes(User.class)
            .globalOperationParameters(
                Arrays.asList(
                    new ParameterBuilder()
                        .name("Authorization")
                        .description("Header para facilitar o envio do Authorization Bearer T
oken")
                        .modelRef(new ModelRef("string"))
                        .parameterType("header")
                        .required(false)
                        .build()));
    }

    // código posterior omitido
}
```

Acesse a interface de `Swagger` e perceba que agora será possível testar todos os *endpoints* de forma simples, inclusive nos casos onde o envio do *JWT* se faz necessário.

INSERINDO TÓPICOS E VALIDANDO DADOS DE ENTRADA

9.1 INSERINDO UM TÓPICO

Agora que nossa aplicação consegue autenticar um usuário do Fórum, precisamos implementar o *endpoint* que insere um novo tópico. A criação de um novo tópico acontece na aplicação cliente por meio de um formulário com campos para serem preenchidos:

Figura 9.1: new-topic

Note que o cliente enviará a descrição, o conteúdo e o nome do curso de um tópico. O seguinte JSON será enviado pelo cliente através da requisição POST /api/topics :

```
{
  "shortDescription": "Descrição curta da dúvida",
  "content": "Conteúdo detalhado da dúvida",
  "courseName": "Nome do curso"
}
```

Para atender à requisição vinda do cliente, precisamos criar um novo método na classe `TopicController` que responda pela URI /api/topics :

```
@RestController
@RequestMapping("/api/topics")
public class TopicController {
```

```

// atributos anteriores omitidos

// métodos anteriores omitidos

@PostMapping()
public void createTopic() {

}

```

Como se trata de uma requisição POST , utilizaremos a anotação @PostMapping para o mapeamento. O método createTopic() deve receber os dados do tópico da requisição e usaremos novamente um DTO para a transferência de dados. Portanto, nosso método deve consumir um JSON e usamos o parâmetro consumes na anotação @PostMapping para especificar isso:

```

@PostMapping(consumes=MediaType.APPLICATION_JSON_VALUE)
public void createTopic(@RequestBody NewTopicInputDto newTopicDto) {

}

```

Note que foi preciso usar a anotação @RequestBody do Spring MVC indicando que os dados da requisição do parâmetro newTopicDto do método createTopic() chegarão pelo corpo da requisição já que se trata de uma requisição POST . Em seguida, criamos a classe NewTopicInputDto representando os dados vindos na requisição:

```

public class NewTopicInputDto {

    private String shortDescription;
    private String content;
    private String courseName;

    //getters and setters necessários
}

```

O método do controller deve processar os dados de um novo tópico e adicioná-lo no banco de dados. Mas antes, para criar um tópico, precisamos, além da descrição e do conteúdo, da instância de um curso e do usuário logado exigidos pelo construtor de Topic :

```

Topic topic = new Topic(newTopicDto.shortDescription,
    newTopicDto.content, owner, course);

```

O usuário precisa estar logado para inserir um novo tópico. Portanto, o token será enviado pela aplicação cliente. Podemos obter o usuário logado através do contexto de segurança do Spring Security, que foi atribuído anteriormente pelo nosso filtro de segurança:

```

User owner = (User) SecurityContextHolder.getContext()
    .getAuthentication().getPrincipal();

```

O curso será buscado pelo seu nome. Criaremos, então, a interface CourseRepository com o método findByName() :

```

public interface CourseRepository extends Repository<Course, Long> {

```

```
        Course findByName(String courseName);  
    }
```

Agora é possível construir o tópico para ser persistido no banco:

```
@PostMapping(consumes=MediaType.APPLICATION_JSON_VALUE)  
public void createTopic(NewTopicInputDto newTopicDto) {  
  
    User owner = (User) SecurityContextHolder.getContext()  
        .getAuthentication().getPrincipal();  
  
    Course course = this.courseRepository  
        .findByName(newTopicDto.getCourseName());  
  
    Topic topic = new Topic(newTopicDto.getShortDescription()  
        newTopicDto.getContent(), owner, course);  
  
    this.topicRepository.save(topic);  
}
```

Acontece que construir um tópico não é tarefa do nosso *controller*. Vamos delegar isso para a classe que contém essas informações, ou seja, para dentro do nosso `NewTopicInputDto`:

```
public class NewTopicInputDto {  
  
    // código anterior omitido  
  
    public Topic build(User owner, CourseRepository courseRepository) {  
        Course course = courseRepository.findByName(this.courseName);  
        return new Topic(this.shortDescription, this.content, owner, course);  
    }  
}
```

Neste ponto, entendemos que passar uma dependência como argumento de método não é uma prática comum, mas para a maioria dos casos não parece complicar a manutenção e evita de ter uma explosão de classes no estilo de *mappers/converters*.

Agora, no *controller*, basta chamar o método `build()` do DTO, que retorna um `Topic`:

```
@PostMapping(consumes=MediaType.APPLICATION_JSON_VALUE)  
public void createTopic(NewTopicInputDto newTopicDto) {  
  
    Topic topic = newTopicDto.build(loggedUser, this.courseRepository);  
    this.topicRepository.save(topic);  
}
```

Ao invés de pegar o usuário do contexto de segurança, podemos pedir que o próprio Spring faça isso e o injete como parâmetro do método `createTopic()` por meio da anotação `@AuthenticationPrincipal`:

```
@PostMapping(consumes=MediaType.APPLICATION_JSON_VALUE)  
public void createTopic(NewTopicInputDto newTopicDto,  
    @AuthenticationPrincipal User loggedUser) {  
  
    Topic topic = newTopicDto.build(loggedUser, this.courseRepository);  
    this.topicRepository.save(topic);  
}
```

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

9.2 CONSTRUINDO UMA RESPOSTA MAIS EXPRESSIVA

É interessante devolver uma resposta mais expressiva para o cliente. Vamos criar um DTO de *output* com os detalhes do tópico criado:

```
public class TopicOutputDto {  
  
    private Long id;  
    private String shortDescription;  
    private String content;  
    private TopicStatus status;  
    private int numberofResponses;  
    private Instant creationInstant;  
    private Instant lastUpdate;  
    private String ownerName;  
    private String courseName;  
    private String subcategoryName;  
    private String categoryName;  
    private List<AnswerOutputDto> answers = new ArrayList<>();  
  
    public TopicOutputDto(Topic topic) {  
        this.id = topic.getId();  
        this.shortDescription = topic.getShortDescription();  
        this.content = topic.getContent();  
        this.status = topic.getStatus();  
        this.numberofResponses = topic.getNumberofAnswers();  
        this.creationInstant = topic.getCreationInstant();  
        this.lastUpdate = topic.getLastUpdate();  
        this.ownerName = topic.getOwner().getName();  
        this.courseName = topic.getCourse().getName();  
        this.subcategoryName = topic.getCourse().getSubcategoryName()  
        this.categoryName = topic.getCourse().getCategoryName();  
  
        List<AnswerOutputDto> answers = AnswerOutputDto  
            .listFromAnswers(topic.getAnswers());  
        this.answers.addAll(answers);  
    }  
  
    // getters  
}
```

Em seguida, construímos a resposta usando novamente a classe `ResponseEntity` do Spring MVC:

```
@PostMapping(consumes=MediaType.APPLICATION_JSON_VALUE,
    produces=MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<TopicOutputDto> createTopic(NewTopicInputDto newTopicDto,
    @AuthenticationPrincipal User loggedUser) {

    Topic topic = newTopicDto.build(loggedUser, this.courseRepository);
    this.topicRepository.save(topic);

    return ResponseEntity.ok(new TopicOutputDto(topic));
}
```

Tome cuidado para não esquecer de adicionar o parâmetro `produces` dentro de `@PostMapping` especificando o tipo de retorno (neste caso, JSON).

Segundo o protocolo HTTP, quando um recurso é criado no servidor, a resposta deve retornar o *status 201 (created)* e conter o conteúdo do objeto criado. Além disso, deve retornar um *location* para o cliente saber por meio de qual URI é possível acessar o objeto.

O Spring provê uma classe chamada `UriComponentsBuilder`, que possui métodos utilitários para ajudar na construção das URIs, e podemos pedir que a injeção ocorra via parâmetro do método `createTopic()`:

```
@PostMapping(consumes=MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<TopicOutputDto> createTopic(NewTopicInputDto newTopicDto,
    @AuthenticationPrincipal User loggedUser,
    UriComponentsBuilder uriBuilder) {

    Topic topic = newTopicDto.build(loggedUser, this.courseRepository);
    this.topicRepository.save(topic);

    URI path = uriBuilder.path("/api/topics/{id}")
        .buildAndExpand(topic.getId()).toUri();

    return ResponseEntity.created(path).body(new TopicOutputDto(topic));
}
```

`UriComponentsBuilder` possui o método `path()`, que recebe a `String` que representa a URI a ser construída, e o método `buildAndExpand()`, que constrói e substitui as variáveis do *endpoint*. Por fim, utilizamos o método `toUri()` para criar a instância de uma `URI`. Dessa maneira, é possível passá-la para o método `created()` de `ResponseEntity`. Utilizamos o `id` para a construção da URI já que, para acessar os detalhes de um recurso, é necessário um identificador único.

A resposta gerada será como a figura baixo:

Code	Details
201	<p>Response body</p> <pre>{ "id": 60, "shortDescription": "Descrição do tópico novo", "content": "Conteúdo do tópico novo", "status": "NOT_ANSWERED", "numberOfResponses": 0, "creationInstant": "2019-01-30T16:10:04.678Z", "lastUpdate": "2019-01-30T16:10:04.678Z", "ownerName": "Rafael Rollo", "courseName": "Scrum", "sub categoryName": "Agilidade", "categoryName": "Business", "answers": [] }</pre> <p style="text-align: right;">Download</p>

Figura 9.2: new-topic-response

9.3 EXERCÍCIOS: INSERINDO NOVOS TÓPICOS

1. Ao receber os dados de novos tópicos da aplicação cliente, nossa API deve ser capaz de processar a inserção considerando o seguinte JSON:

```
{
  "shortDescription": "Descrição curta da dúvida",
  "content": "Conteúdo detalhado da dúvida",
  "courseName": "Nome do curso"
}
```

Sendo assim, vamos criar um DTO de entrada representando este objeto. No pacote `br.com.alura.forum.controller.dto.input`, crie a classe `NewTopicInputDto`:

```
public class NewTopicInputDto {

    private String shortDescription;
    private String content;
    private String courseName;

    // getters and setters necessários
}
```

2. Para atender a requisição vinda do cliente, crie o método `createTopic()` na classe `TopicController`, recebendo uma instância do DTO de entrada:

```
@RestController
@RequestMapping("/api/topics")
public class TopicController {

    // atributos anteriores omitidos

    // métodos anteriores omitidos

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
                  produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<?> createTopic(@RequestBody NewTopicInputDto newTopicDto) {

        this.topicRepository.save(topic);
        return ResponseEntity.ok().build();
    }
}
```

3. Perceba que agora o código ainda não compila. Precisamos de uma instância de `Topic`, mas temos apenas uma instância de `NewTopicInputDto`.

Na classe `NewTopicInputDto`, crie o método `build()` para construir um `Topic` em função das informações enviadas pela aplicação cliente.

```
public class NewTopicInputDto {  
    // código anterior omitido  
  
    public Topic build(User owner, CourseRepository courseRepository) {  
  
        Course course = courseRepository.findByName(this.courseName);  
        return new Topic(this.shortDescription, this.content, owner, course);  
    }  
}
```

Será necessário criar a interface `CourseRepository` no pacote `br.com.alura.forum.repository`:

```
public interface CourseRepository extends Repository<Course, Long> {  
  
    Course findByName(String name);  
}
```

De volta ao `controller`, utilize o método `build()` para obter o `Topic`:

```
@RestController  
@RequestMapping("/api/topics")  
public class TopicController {  
  
    // atributos anteriores omitidos  
  
    @Autowired  
    private CourseRepository courseRepository;  
  
    // métodos anteriores omitidos  
  
    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,  
                 produces = MediaType.APPLICATION_JSON_VALUE)  
    public ResponseEntity<?> createTopic(@RequestBody NewTopicInputDto newTopicDto,  
                                         @AuthenticationPrincipal User loggedUser) {  
  
        Topic topic = newTopicDto.build(loggedUser, this.courseRepository);  
        this.topicRepository.save(topic);  
  
        return ResponseEntity.ok().build();  
    }  
}
```

Repare que foi necessário injetar o `CourseRepository`, para repassá-lo ao método `build` da classe `NewTopicInputDto`.

Note também que foi necessário receber o usuário autenticado usando a anotação `@AuthenticationPrincipal` do pacote `org.springframework.security.core` para satisfazer

esta dependência do método `build()`.

Para o código compilar ainda falta criar o método `save` na interface `TopicRepository`:

```
public interface TopicRepository extends Repository<Topic, Long>, JpaSpecificationExecutor<Topic>
{
    // métodos anteriores omitidos
    void save(Topic topic);
}
```

4. Perceba que até então a resposta dada é muito simples, devolvendo apenas o *status code* 200 para o cliente.

No pacote `br.com.alura.forum.controller.dto.output`, crie a classe `TopicOutputDto`, que usaremos para devolver uma resposta mais expressiva para o cliente:

```
public class TopicOutputDto {

    private Long id;
    private String shortDescription;
    private String content;
    private TopicStatus status;
    private int numberofResponses;
    private Instant creationInstant;
    private Instant lastUpdate;

    private String ownerName;
    private String courseName;
    private String subcategoryName;
    private String categoryName;

    public TopicOutputDto(Topic topic) {
        this.id = topic.getId();
        this.shortDescription = topic.getShortDescription();
        this.content = topic.getContent();
        this.status = topic.getStatus();
        this.numberofResponses = topic.getNumberOfAnswers();
        this.creationInstant = topic.getCreationInstant();
        this.lastUpdate = topic.getLastUpdate();
        this.ownerName = topic.getOwner().getName();
        this.courseName = topic.getCourse().getName();
        this.subcategoryName = topic.getCourse().getSubcategoryName();
        this.categoryName = topic.getCourse().getCategoryName();
    }

    // getters necessários
}
```

De volta ao `controller`, modifique o retorno da nossa `action` para devolver o *status code* 201 `CREATED`, por meio do método `create()` de `ResponseEntity`, além de um `TopicOutputDto`, com o conteúdo do novo tópico criado.

```
@RestController
@RequestMapping("/api/topics")
public class TopicController {
```

```

// atributos anteriores omitidos

@Autowired
private CourseRepository courseRepository;

// métodos anteriores omitidos

@PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
    produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<TopicOutputDto> createTopic(@RequestBody NewTopicInputDto newTopicDto,
    @AuthenticationPrincipal User loggedUser) {

    Topic topic = newTopicDto.build(loggedUser, this.courseRepository);
    this.topicRepository.save(topic);

    return ResponseEntity.created().body(new TopicOutputDto(topic));
}
}

```

5. Além de retornar o recurso criado, é interessante que o cliente saiba qual o seu endereço, ou seja, sua URI - Unified Resource Identifier. A classe `UriComponentsBuilder` do Spring possui métodos utilitários para ajudar na construção das URI s.

Altere a implementação da nossa *action* para gerar e devolver a `URI` do tópico criado.

```

@PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
    produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<TopicOutputDto> createTopic(
    @RequestBody NewTopicInputDto newTopicDto,
    @AuthenticationPrincipal User loggedUser,
    UriComponentsBuilder uriBuilder) {

    Topic topic = newTopicDto.build(loggedUser, this.courseRepository);
    this.topicRepository.save(topic);

    URI path = uriBuilder.path("/api/topics/{id}")
        .buildAndExpand(topic.getId()).toUri();

    return ResponseEntity.created(path).body(new TopicOutputDto(topic));
}

```

Perceba que podemos pedir a injeção de `UriComponentsBuilder` no contexto do nosso método.

6. Rode novamente a aplicação e acesse a página do Swagger UI pelo endereço <http://localhost:8080/swagger-ui.html>. Clique na seção referente ao **topic-controller** e selecione o endpoint `POST /api/topics`.
- Clique em `Try it out`, insira um token válido, altere os dados no modelo de novo tópico e clique em `Execute`:

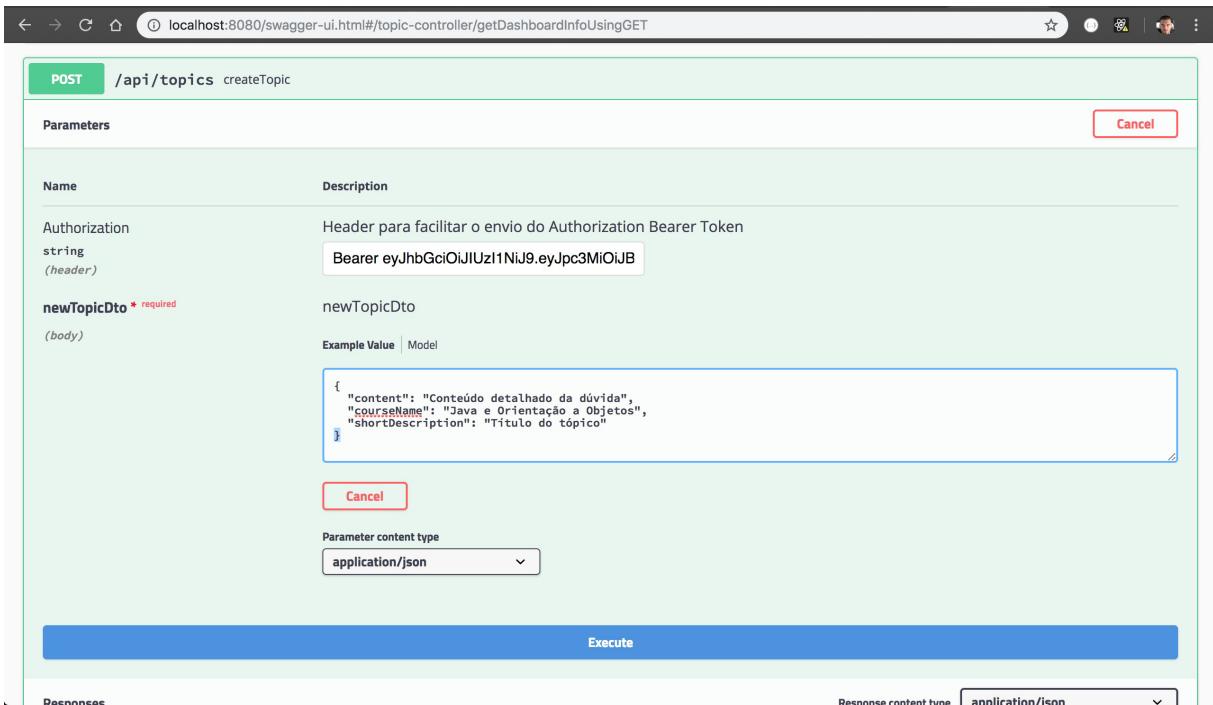


Figura 9.3: Tela do Swagger

- Veja que a aplicação devolve a informação do tópico criado com *status code* 201:

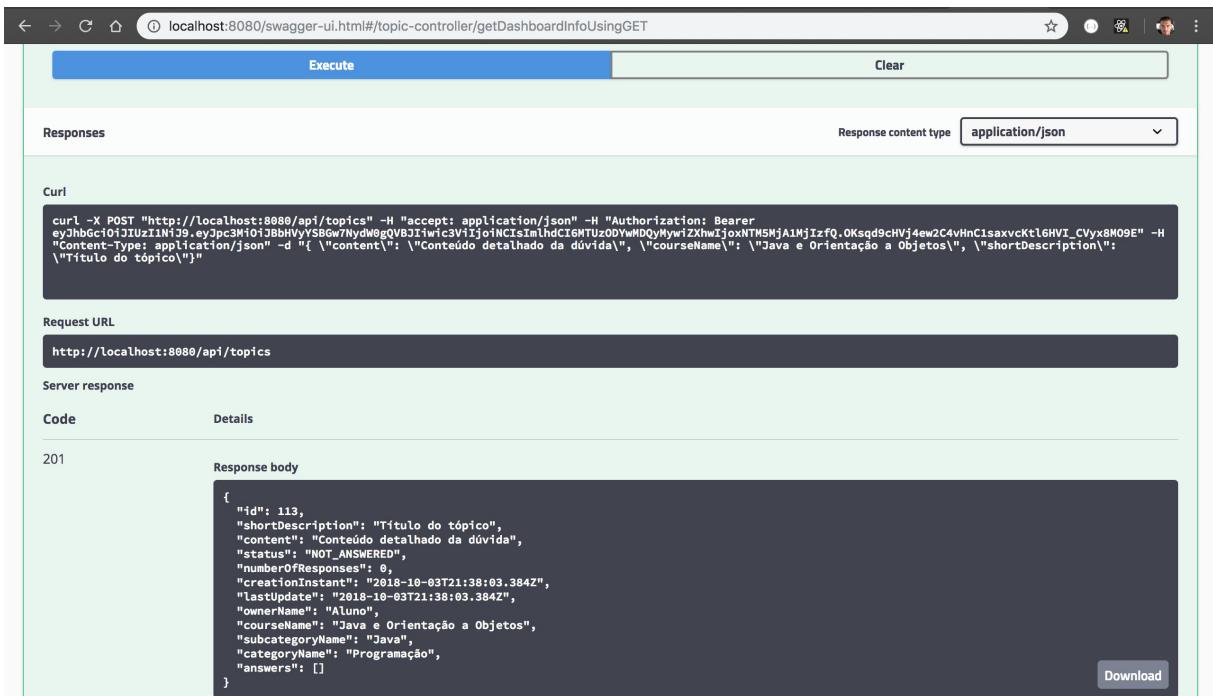


Figura 9.4: Tela do Swagger - response

7. (Opcional) Para que seja possível testar a aplicação cliente com esse novo recurso, no terminal,

acesse a pasta do projeto da aplicação cliente `Desktop/fj27-alura-forum-client/`, e execute o comando `git checkout 09InserindoTopicos`, para atualizar seu estado.

Com o *client* e a *API* rodando com os novos recursos, teste novamente o funcionamento da app cliente recarregando a página (`http://localhost:3000/`) e veja que ao clicar em *Criar Novo Tópico* você já consegue acessar o formulário de novos tópicos e para registrar novas dúvidas.

9.4 VALIDANDO UM TÓPICO

Validação de dados é uma tarefa muito comum em uma aplicação, desde a camada de apresentação até a camada de persistência. Geralmente essa validação é a mesma em cada camada e, para evitar essa repetição, os desenvolvedores agrupam a lógica de validação diretamente no modelo de domínio.

[Bean Validation](#) é uma especificação que facilita o processo de validação. Ela fornece declarações de restrição a nível de objeto. A API do Bean Validation é vista como uma extensão geral do modelo de objeto *JavaBeans* e aplica restrições em seus atributos. Sua implementação de referência é a [Hibernate Validation](#) e é a implementação padrão utilizada pelo Spring Boot dentro do *starter* `spring-boot-starter-validation`, que devemos adicionar no arquivo `pom.xml`:

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

A especificação também define pontos de integração em contextos de injeção de dependência como o Spring.

Uma restrição é expressa por meio de uma ou mais anotações, que podem ser a nível de método ou atributo. Por exemplo, no Fórum da Alura, a descrição de um tópico não pode ser nula e ainda deve ter no mínimo 10 caracteres. A biblioteca provê uma anotação para cada um desses casos: `@NotBlank` (que também vai checar se o atributo não é vazio) e `@Size`, respectivamente. Como recebemos esses dados pelo DTO, basta aplicá-las no atributo `shortDescription` de `NewTopicInputDto`:

```
public class NewTopicInputDto {

    @NotBlank
    @Size(min = 10)
    private String shortDescription;

    // outros atributos e métodos
}
```

Outras anotações nativas você pode encontrar diretamente na [documentação](#) oficial do Bean Validation.

PARA SABER MAIS

Também é possível produzir suas próprias anotações. Sugerimos a leitura deste [post](#) do blog da Caelum que aborda o assunto.

Para aplicar a validação considerando as restrições aplicadas em `NewTopicInputDto`, é preciso avisar ao Spring que aceite apenas dados válidos. O Bean Validation também provê uma anotação para isso: `@Valid`. Essa anotação é usada no parâmetro no método do *controller* onde é feita a injeção do objeto a ser validado:

```
@PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
    produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<TopicOutputDto> createTopic(
    @Valid @RequestBody NewTopicInputDto newTopicDto,
    @AuthenticationPrincipal User loggedUser,
    UriComponentsBuilder uriBuilder) {

    // código interior omitido
}
```

Dessa maneira, o Spring leva em consideração as restrições de validação especificadas na classe `NewTopicInputDto`. Caso o campo `shortDescription` chegue com menos do que 10 caracteres, um erro é mostrado para o cliente. No console, vemos o seguinte `WARN`:

```
WARN 19276 --- [nio-8080-exec-1] .w.s.m.s.DefaultHandlerExceptionResolver : Resolved [org.springframework.web.bind.MethodArgumentNotValidException: Validation failed for argument [0] in public org.springframework.http.ResponseEntity<br.com.alura.forum.controller.dto.output.TopicOutputDto> br.com.alura.forum.controller.TopicController.createTopic(br.com.alura.forum.controller.dto.input.NewTopicInputDto,br.com.alura.forum.model.User,org.springframework.web.util.UriComponentsBuilder,org.springframework.validation.BindingResult): [Field error in object 'newTopicInputDto' on field 'shortDescription': rejected value [aaa]; codes [Size.newTopicInputDto.shortDescription,Size.shortDescription,Size.java.lang.String,Size]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [newTopicInputDto.shortDescription,shortDescription]; arguments []; default message [shortDescription],2147483647,10]; default message [size must be between 10 and 2147483647]] ]
```

A exceção `MethodArgumentNotValidException` foi lançada pela aplicação e resolvida pelo `DefaultHandlerExceptionResolver` do Spring. Este *handler* é habilitado pela `DispatcherServlet` e é responsável por tratar as exceções *default* lançadas pelo Spring MVC, além de traduzir para o *status code* do HTTP correspondente para gerar uma resposta.

Ou seja, como pedimos um `NewInputTopicDto` válido por meio da anotação `@Valid` e passamos um objeto inválido (com descrição menor do que 10 caracteres), o Spring vai lançar a exceção `MethodArgumentNotValidException` e seu *handler* padrão vai tratar a exceção e construir um objeto para ser enviado para o cliente.

Na parte final dessa mensagem de *log* podemos ver informações do erro gerado quando tentamos

criar um tópico com uma descrição inválida:

```
Field error in object 'newTopicInputDto' on field 'shortDescription':  
    rejected value [aaa];  
  codes [Size.newTopicInputDto.shortDescription,Size.shortDescription,Size.java.lang.String,Size];  
  arguments [org.springframework.context.support.DefaultMessageSourceResolvable:  
    codes [newTopicInputDto.shortDescription,shortDescription];  
    arguments [];  
    default message [shortDescription],2147483647,10];  
    default message [size must be between 10 and 2147483647]
```

No Swagger podemos checar a resposta padrão construída:

The screenshot shows the Swagger UI interface. At the top, there's a table with two columns: 'Code' and 'Details'. Under 'Code', '400' is selected. Under 'Details', the 'Response body' section is expanded, showing a JSON response. The JSON structure is as follows:

```
{"timestamp": "2019-01-30T16:37:57.255+0000", "status": 400, "error": "Bad Request", "errors": [{"codes": ["Size.newTopicInputDto.shortDescription", "Size.shortDescription", "Size.java.lang.String", "Size"], "arguments": [{"codes": ["newTopicInputDto.shortDescription", "shortDescription"], "arguments": null, "defaultMessage": "shortDescription", "code": "shortDescription"}, {"code": "shortDescription", "message": "size must be between 10 and 2147483647"}], "message": "size must be between 10 and 2147483647"}]}
```

At the bottom right of the JSON preview area, there is a 'Download' button.

Figura 9.5: mensagens-erros-padrão

Acontece que essa mensagem de erro contém muitas coisas e queremos devolver ao cliente um objeto de erro mais simples e direto. Além disso, queremos customizar as mensagens de erro padrão do Bean Validation. Para isso, vamos criar um DTO representando os erros de atributos:

```
public class FieldErrorOutputDto {  
  
    private String field;  
    private String message;  
  
    FieldErrorOutputDto() {}  
  
    public FieldErrorOutputDto(String field, String message) {  
        this.field = field;  
        this.message = message;  
    }  
  
    // getters  
}
```

Como pode existir mais de um erro de validação ao inserir um tópico - no nosso caso, o conteúdo também não deve ter menos do que 10 caracteres - vamos construir outro DTO que armazena todos os erros em uma lista:

```
public class ValidationErrorsOutputDto {  
  
    private List<FieldErrorOutputDto> fieldErrors = new ArrayList<>();  
  
    public void addFieldError(String field, String message) {  
        FieldErrorOutputDto fieldError = new FieldErrorOutputDto(field, message);  
        fieldErrors.add(fieldError);  
    }  
  
    public int getNumberOfErrors() {  
        return this.fieldErrors.size();  
    }  
  
    //getters  
}
```

Como fazer, agora, para que o Spring considere nosso ValidationErrorsOutputDto para construção da resposta? Na verdade, o que precisamos fazer é outro *handler* que substitua o DefaultHandlerExceptionResolver do Spring e seja capaz de construir um ValidatorErrorOutputDto para ser devolvido ao cliente. Criamos, então, um método dentro de TopicController que lide com a exceção MethodArgumentNotValidException .

```
@RestController  
@RequestMapping("/api/topics")  
public class TopicController {  
  
    // outros atributos e métodos  
  
    @ResponseStatus(HttpStatus.BAD_REQUEST)  
    @ExceptionHandler(MethodArgumentNotValidException.class)  
    public ValidationErrorsOutputDto handleValidationErrors() {  
        return null;  
    }  
}
```

A anotação @ExceptionHandler é usada em métodos e/ou classes que são *handlers* capazes de lidar/tratar alguma exceção. Ou seja, no nosso caso, o método handleValidationErrors() será capaz de lidar com a exceção MethodArgumentNotValidException . Além disso, utilizamos a anotação @ResponseStatus passando o status code que será definido na resposta.

Agora precisamos construir um objeto ValidationErrorsOutputDto e adicionar os erros existentes em sua lista de erros. O Spring armazena os erros em um objeto do tipo BindingResult . Imagine que este objeto é uma caixa vazia e, caso algum erro de validação seja detectado, o erro é adicionado nela. Portanto, precisamos de um BindingResult para acessar os erros. A própria exceção MethodArgumentNotValidException contém um método getBindingResult() da qual podemos pedir todos os erros contidos nele:

```

@RestController
@RequestMapping("/api/topics")
public class TopicController {

    // outros atributos e métodos

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ValidationErrorsOutputDto handleValidationException(MethodArgumentNotValidException exception)

    {

        List<FieldError> fieldErrors = exception.getBindingResult().getFieldErrors();
        return null;
    }
}

```

Agora, com a lista de erros em mãos, podemos construir nosso DTO:

```

@RestController
@RequestMapping("/api/topics")
public class TopicController {

    // outros atributos e métodos

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ValidationErrorsOutputDto handleValidationException(MethodArgumentNotValidException exception)

    {

        List<FieldError> fieldErrors = exception.getBindingResult().getFieldErrors();

        ValidationErrorsOutputDto validationErrors = new ValidationErrorsOutputDto();

        fieldErrors.forEach(error -> {
            validationErrors.addFieldError(error.getField(), error.getDefaultMessage());
        });

        return validationErrors;
    }
}

```

Pronto! Dessa maneira, quando alguma exceção de validação for lançada em `TopicController`, nosso método será chamado e gerará a resposta abaixo:

Server response

Code	Details
400 <i>Undocumented</i>	<p>Error:</p> <p>Response body</p> <pre>{ "errors": [{ "field": "shortDescription", "message": "size must be between 10 and 2147483647" }], "numberOfErrors": 1 }</pre> <p>Download</p>

Figura 9.6: mensagens-erros-padrão

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

9.5 MELHORANDO O CÓDIGO

Nossa aplicação pode vir a crescer e existir funcionalidades que também dependam de dados válidos e utilizem o Bean Validation. A criação de uma resposta em um tópico é um desses casos. Teremos que verificar se os campos de uma nova resposta do fórum são válidos. Portanto, em nosso futuro `AnswerController` teremos que acrescentar um método capaz de lidar com exceções de validação idêntico ao feito em `TopicRepository`.

Copiar e colar código é uma prática que deve ser evitada para nos poupar de problemas com manutenção no futuro. Para evitar a cópia desse método em todos os `controllers`, vamos isolá-lo em uma classe para que seja reaproveitado. Criamos a classe `ValidationErrorHandler` com nosso método

`handlerValidation()` e utilizamos a anotação `@RestControllerAdvice`:

```
@RestControllerAdvice
public class ValidationErrorHandler {

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ValidationErrorsOutputDto handleValidationError(MethodArgumentNotValidException exception)
    {

        List<FieldError> fieldErrors = exception.getBindingResult().getFieldErrors();

        ValidationErrorsOutputDto validationErrors = new ValidationErrorsOutputDto();

        fieldErrors.forEach(error -> {
            validationErrors.addFieldError(error.getField(), error.getDefaultMessage());
        });

        return validationErrors;
    }
}
```

A anotação `@RestControllerAdvice` é basicamente uma junção das anotações `@ControllerAdvice` e `@ResponseBody`. A anotação `@ControllerAdvice` do Spring aplica o método globalmente para todos os *controllers*. Isso significa que, se qualquer método de *controller* lançar uma exceção de validação, esse método será chamado. Dessa maneira, reutilizamos o código para todos os *controllers* da aplicação.

9.6 EXERCÍCIOS: VALIDANDO UM TÓPICO

1. Adicione a dependência do *starter* do Spring Validation no arquivo `pom.xml`:

```
<dependencies>

    <!-- dependência do jjwt -->

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

2. As informações de `shortDescription` e `content` devem ser obrigatórias, além disso, devem conter um mínimo de 10 caracteres. O `courseName` também deve ser obrigatório. Usaremos a integração com a Bean Validation para garantir a validação dos dados de um tópico.

Na classe `NewTopicInputDto`, adicione as *annotations* necessárias a cada campo. As anotações do Bean Validation serão importadas do pacote `javax.validation.constraints`:

```
public class NewTopicInputDto {

    @NotBlank
    @Size(min = 10)
    private String shortDescription;
```

```

@NotBlank
@Size(min = 10)
private String content;

@NotEmpty
private String courseName;

// código posterior omitido
}

```

A anotação `@NotBlank` valida se uma `String` não é nula ou vazia, já a anotação `@Size` valida a quantidade de caracteres da `String` de acordo com os valores de mínimo e máximo especificados.

- Na classe `TopicController`, altere a assinatura do método `createTopic()` adicionando a annotation `@Valid` para que o Spring execute a validação do objeto de `NewTopicOutputDto` no processo de *binding*.

```

@PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
            produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<TopicOutputDto> createTopic(
    @Valid @RequestBody NewTopicInputDto newTopicDto,
    @AuthenticationPrincipal User loggedUser,
    UriComponentsBuilder uriBuilder) {

    // código interior omitido
}

```

- Rode novamente a aplicação e tente criar um novo tópico com um `shortDescription` inválido.



Figura 9.7: Tela do Swagger - response

Note que o JSON retornado representa exatamente o erro de validação.

- O JSON demonstrado no exercício anterior foi gerado a partir do objeto interno do Spring que representa os problemas ao processar a requisição com dados inválidos.

Para que tenhamos objetos mais específicos para o domínio da aplicação sendo retornados ao cliente

mesmo nos casos de erro, vamos utilizar DTOs de *output*. Crie a classe `FieldErrorResponseDto` no pacote `br.com.alura.forum.validator.dto`, que vai representar um erro de entrada para um campo, com os atributos `field` e `message`:

```
public class FieldErrorResponseDto {  
  
    private String field;  
    private String message;  
  
    FieldErrorResponseDto() {}  
  
    public FieldErrorResponseDto(String field, String message) {  
        this.field = field;  
        this.message = message;  
    }  
  
    public String getField() {  
        return field;  
    }  
  
    public String getMessage() {  
        return message;  
    }  
}
```

- i. Como mais de um campo pode ser preenchido de maneira inválida, vamos criar outro DTO. Este vai armazenar todos os erros de validação em uma lista. No pacote `br.com.alura.forum.validator.dto`, crie a classe `ValidationErrorsOutputDto`.

```
public class ValidationErrorsOutputDto {  
  
    private List<FieldErrorResponseDto> fieldErrors = new ArrayList<>();  
  
    public void addFieldError(String field, String message) {  
        FieldErrorResponseDto fieldError = new FieldErrorResponseDto(field, message);  
        fieldErrors.add(fieldError);  
    }  
  
    public List<FieldErrorResponseDto> getErrors() {  
        return fieldErrors;  
    }  
  
    public int getNumberOfErrors() {  
        return this.fieldErrors.size();  
    }  
}
```

6. Quando um erro de validação acontece, internamente o Spring MVC lança a exceção `MethodArgumentNotValidException`. Precisamos então criar uma classe capaz de lidar com os erros e com o tratamento dessa exceção, quando os mesmos acontecerem.

No pacote `br.com.alura.forum.handler`, crie a classe `ValidationErrorHandler` com um método para tratar a exceção que deve devolver nosso DTO preenchido:

```
@RestControllerAdvice  
public class ValidationErrorHandler {
```

```

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ValidationErrorsOutputDto handleValidationError(
        MethodArgumentNotValidException exception) {

    }
}

```

7. Agora, precisamos adicionar os erros e suas respectivas mensagens dentro de nosso `ValidationErrorsOutputDto`. Os erros de validação e conversão ocorridos durante o processo de *binding* ficam registrados em um objeto do tipo `BindingResult`, que pode ser acessado pela `MethodArgumentNotValidException`.

Usaremos o método `getBindingResult()` para acessar os erros e gerar nosso DTO:

```

@RestControllerAdvice
public class ValidationErrorHandler {

    @Autowired
    private MessageSource messageSource;

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ValidationErrorsOutputDto handleValidationError(
        MethodArgumentNotValidException exception) {

        List<FieldError> fieldErrors = exception.getBindingResult().getFieldErrors();

        ValidationErrorsOutputDto validationErrors = new ValidationErrorsOutputDto();

        fieldErrors.forEach(error -> {
            validationErrors.addFieldError(error.getField(), error.getDefaultMessage());
        });

        return validationErrors;
    }
}

```

8. Utilizando o `Swagger UI`, teste novamente a requisição com dados inválidos e perceba que agora o JSON retornado exibe informações bem mais específicas sobre os problemas ocorridos.

Server response	
Code	Details
400 <i>Undocumented</i>	Error:
Response body	
	<pre>{ "errors": [{ "field": "shortDescription", "message": "size must be between 10 and 2147483647" }, { "field": "shortDescription", "message": "must not be blank" }], "numberOfErrors": 2 }</pre>
	Download

Figura 9.8: DTO de erros de validação

9.7 VALIDAÇÕES DE NEGÓCIO

Até agora vimos como validar um tópico apenas por meio das anotações do Bean Validation. Acontece que o Bean Validation nos auxilia em validações comuns, usadas em qualquer aplicação, como verificar se um campo não é nulo, se não é negativo, se é maior do que 10, etc... Quando precisamos validar algo específico da regra de negócio da aplicação, a coisa muda, não tem como o Bean Validation conhecer detalhes do negócio. Por exemplo, no Fórum da Alura, para evitar possíveis *spams*, um usuário pode criar apenas 4 tópicos a cada hora. Como validar isso?

Para casos de validações específicas da regra de negócio, o Spring provê uma interface chamada `Validator`. Essa interface nos obriga a implementar dois métodos:

- `supports()` : método que verifica se este validador oferece suporte ao objeto a ser validado.
- `validate()` : método que faz a validação

Portanto, vamos precisar implementar um `Validator` customizado que deve dar suporte ao objeto `NewTopicInputDto` a ser validado. Para isso implementamos o método `supports()` :

```
public class NewTopicCustomValidator implements Validator {

    @Override
    public boolean supports(Class<?> clazz) {
        return NewTopicInputDto.class.isAssignableFrom(clazz);
    }

    @Override
    public void validate(Object target, Errors errors) {
```

```

        // processamento da validação
    }
}

```

Agora devemos implementar a regra de validação no método `validate()`. Primeiro, devemos ir até o banco de dados e buscar por todos os tópicos de um usuário criados a uma hora atrás. Aproveitaremos ainda mais os *query methods* do Spring Data e pediremos para ordenar os tópicos pelo instante de criação:

```

public interface TopicRepository extends Repository<Topic, Long>, JpaSpecificationExecutor<Topic>{

    List<Topic> findByOwnerAndCreationInstantAfterOrderByCreationInstantAsc(loggedUser,
        oneHourAgo);
}

```

Voltando ao nosso método `validate()` da classe `NewTopicCustomValidator`, pegamos o instante correspondente a uma hora atrás e chamamos o método que acabamos de criar em `TopicRepository` para buscar a lista de tópicos específica:

```

public class NewTopicCustomValidator implements Validator {

    private final TopicRepository topicRepository;
    private User loggedUser;

    public NewTopicCustomValidator(TopicRepository topicRepository, User loggedUser) {
        this.topicRepository = topicRepository;
        this.loggedUser = loggedUser;
    }

    // implementação do método supports()

    @Override
    public void validate(Object target, Errors errors) {

        Instant oneHourAgo = Instant.now().minus(1, ChronoUnit.HOURS);
        List<Topic> topics = topicRepository
            .findByOwnerAndCreationInstantAfterOrderByCreationInstantAsc(loggedUser, oneHourAgo);
    }
}

```

Note que foi preciso implementar o construtor com argumentos recebendo o `TopicRepository` e o usuário logado para que o chamador do nosso validador se encarregue de passar esses valores.

Com a lista em mãos, verificamos se o limite de tópicos não excede 4. Caso isso aconteça, devemos registrar o erro no objeto `Errors` através do método `reject()`.

```

@Override
public void validate(Object target, Errors errors) {

    Instant oneHourAgo = Instant.now().minus(1, ChronoUnit.HOURS);
    List<Topic> topics = topicRepository
        .findByOwnerAndCreationInstantAfterOrderByCreationInstantAsc(loggedUser, oneHourAgo);

    if (topics.size() >= 4) {
        Instant instantOfTheOldestTopic = topics.get(0).getCreationInstant();

```

```

        long minutesToNextTopic = Duration.between(oneHourAgo, instantOfTheOldestTopic)
            .getSeconds() / 60;

        errors.reject("newTopicInputDto.limit.exceeded", new Object[] { minutesToNextTopic },
            "O limite individual de novos tópicos por hora foi excedido");
    }
}

```

`Errors` é uma interface do Spring Validation responsável por armazenar e expor as informações de erros de validação de um objeto específico e que também é implementada pelo `BindingResult`. A interface `BindingResult` estende de `Errors` para obter suas capacidades de registro de erro, como faremos em nosso `Validator`. O método `reject()` de `Errors` é responsável por registrar erros globais e recebe três argumentos. O primeiro argumento do método representa o código de erro que usaremos como chave para acessar uma possível mensagem de erro customizada; o segundo (`Object[]`), representa os argumentos que podemos passar para flexibilizar a mensagem de erro que devolveremos para o usuário; já o terceiro argumento será usado como mensagem de erro padrão. Perceba que criamos a variável `minutesToNextTopic` para armazenar o tempo, em minutos, que falta para o usuário criar um novo tópico no caso de já ter excedido seu limite e usar este valor em uma mensagem customizada.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

9.8 REFATORANDO O CÓDIGO

Antes de customizar nossas mensagens, nosso método `validate()` é de difícil leitura. É preciso gastar algum tempo para entender o que o código realmente está fazendo. Para melhorar a semântica do código e isolar parte da lógica que não é responsabilidade do método `validate()`, vamos criar uma classe que representa um possível *spam* no fórum chamada de `PossibleSpam` e que contém a lista de tópicos criados na última hora:

```
public class PossibleSpam {
```

```

private List<Topic> topics;

public PossibleSpam(List<Topic> topics) {
    this.topics = topics;
}

public boolean hasTopicLimitExceeded() {
    return this.topics.size() >= 4;
}

public long minutesToNextTopic(Instant from) {
    Instant instantOfTheOldestTopic = topics.get(0).getCreationInstant();

    return Duration.between(from, instantOfTheOldestTopic)
        .getSeconds() / 60;
}
}

```

Note que isolamos a lógica que verifica quando o limite de tópicos foi excedido e a que calcula quantos minutos faltam para o usuário criar um novo tópico. Agora, nosso método `validate()` pode ser refatorado:

```

@Override
public void validate(Object target, Errors errors) {

    Instant oneHourAgo = Instant.now().minus(1, ChronoUnit.HOURS);
    List<Topic> topics = topicRepository
        .findByOwnerAndCreationInstantAfterOrderByCreationInstantAsc(loggedUser, oneHourAgo);

    PossibleSpam possibleSpam = new PossibleSpam(topics);

    if (possibleSpam.hasTopicLimitExceeded()) {
        long minutesToNextTopic = possibleSpam.minutesToNextTopic(oneHourAgo);
        errors.reject("newTopicInputDto.limit.exceeded", new Object[] { minutesToNextTopic },
            "O limite individual de novos tópicos por hora foi excedido");
    }
}

```

Dessa maneira, fica mais claro que um erro será registrado se um possível *spam* excede o limite.

Nossa validação ainda não funciona. Precisamos registrar nosso validador customizado na classe responsável por fazer o *binding* dos valores que chegam da requisição com os atributos de nossa classe `NewTopicInputDto`. Essa classe é a `WebDataBinder`, que possui o método `addValidators()`, que recebe uma lista de validadores para serem considerados durante o processo de *binding*.

Criaremos um método no nosso `controller` anotado com `@InitBinder`, usada para identificar os métodos que inicializam o `WebDataBinder`, e vamos adicionar o nosso validador customizado nele:

```

@RestController
@RequestMapping("/api/topics")
public class TopicController {

    // outros atributos e métodos omitidos

    @InitBinder("newTopicInputDto")
    public void initBinder(WebDataBinder binder, @AuthenticationPrincipal User loggedUser) {

```

```

        binder.addValidators(new NewTopicCustomValidator(this.topicRepository, loggedUser));
    }
}

```

Agora o Spring considera o nosso validador no processo de *binding*.

9.9 MENSAGENS CUSTOMIZADAS

Nossa validação funciona, mas a aplicação ainda não é capaz de mostrar a mensagem de erro. Por se tratar de erros de regra de negócio, esses erros são chamados de **erros globais** (ou *global errors*) e nosso `ValidationErrorsOutputDto` apenas considera os erros de campo (ou *field errors*). Para que a mensagem de validação de *spam* apareça, precisamos incluir uma lista de *global errors* em nosso DTO. Para facilitar, vamos inserir uma lista de `String` representando as mensagens de erros globais:

```

public class ValidationErrorsOutputDto {

    private List<String> globalErrorMessages = new ArrayList<>();
    private List<FieldErrorOutputDto> fieldErrors = new ArrayList<>();

    public void addError(String message) {
        globalErrorMessages.add(message);
    }

    public void addFieldError(String field, String message) {
        FieldErrorOutputDto fieldError = new FieldErrorOutputDto(field, message);
        fieldErrors.add(fieldError);
    }

    public List<String> getGlobalErrorMessages() {
        return globalErrorMessages;
    }

    public List<FieldErrorOutputDto> getErrors() {
        return fieldErrors;
    }

    public int getNumberOfErrors() {
        return this.fieldErrors.size() + this.globalErrorMessages();
    }
}

```

Precisaremos agora incluir a lista de erros globais na construção do DTO na classe `ValidationErrorHandler`. No fim, nossa classe ficará como no código abaixo:

```

@RestControllerAdvice
public class ValidationErrorHandler {

    @Autowired
    private MessageSource messageSource;

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ValidationErrorsOutputDto handleValidationException(MethodArgumentNotValidException exception) {
        List<ObjectError> globalErrors = exception.getBindingResult().getGlobalErrors();
        List<FieldError> fieldErrors = exception.getBindingResult().getFieldErrors();
    }
}

```

```

    ValidationErrorsOutputDto validationErrors = new ValidationErrorsOutputDto();

    globalErrors.forEach(error -> validationErrors.addError(error.getDefaultMessage()));

    fieldErrors.forEach(error -> {
        validationErrors.addFieldError(error.getField(), error.getDefaultMessage());
    });

    return validationErrors;
}
}

```

Pronto. Com nosso validador pronto e registrado, ao tentar inserir 5 tópicos em menos de uma hora, a mensagem de erro aparece na *response*:

Code	Details
400	Error: <i>Undocumented</i>
Response body <pre>{ "globalErrorMessages": ["O limite individual de novos tópicos por hora foi excedido"], "errors": [], "numberOfErrors": 1 }</pre>	
Download	
Response headers	

Figura 9.9: spam_error

Mas não queremos mostrar a mensagem *default*. Queremos customizar as mensagens do Bean Validation e de nosso validador - que executa a validação em casos de *spam*. Para que isso seja possível, será preciso disponibilizar as mensagens customizadas em um arquivo que o Spring possa encontrá-las.

Precisamos, então, disponibilizar um `MessageSource`. A classe `MessageSource` do Spring é uma classe que resolve as mensagens de erro caso a aplicação disponibilize uma fonte de mensagens customizadas. O Spring Boot configura o Spring MVC de modo que o arquivo de mensagens é lido diretamente da raiz do *classpath*, que no nosso caso também é representada pela pasta `src/main/resources`. Portanto, devemos criar um arquivo chamado `messages.properties`, contendo as mensagens customizadas, na pasta `src/main/resources`.

Esse arquivo funciona como uma mapa de mensagens, recebendo uma chave e um valor. O padrão

da chave das mensagens do Bean Validation segue a estrutura <ANOTATION>.<NOME DO OBJETO VALIDADO>.<NOME DO CAMPO>=mensagem . Portanto, para definir uma mensagem customizada para o tamanho do campo `shortDescription`, teremos:

```
Size.newTopicInputDto.shortDescription=The description must have a minimum of {2} characters
```

{2} acessa o segundo elemento dos argumentos que são passados para flexibilizar a mensagem de erro. No caso na anotação `@Size`, o segundo elemento corresponde ao parâmetro `min`. Uma outra opção é usar o nome do parâmetro entre chaves:

```
Size.newTopicInputDto.shortDescription=The description must have a minimum of {min} characters
```

No caso de nossa mensagem de erro de *spam*, note que o último argumento passado no método `reject()` chamado dentro do método `validate()` de `NewTopicCustomValidator` é a mensagem padrão: "O limite individual de novos tópicos por hora foi excedido". Já o primeiro argumento passado é a chave para ser usada como referência de uma possível mensagem de erro customizada:

```
errors.reject("newTopicInputDto.limit.exceeded", new Object[] { minutesToNextTopic },
    "O limite individual de novos tópicos por hora foi excedido");
```

Agora, no arquivo `messages.properties`, podemos customizar uma mensagem a partir desta chave:

```
newTopicInputDto.limit.exceeded=O limite individual de novos tópicos por hora foi excedido. Você pode
rãi registrar uma nova dúvida em {0} minutos
```

Veja que utilizamos `{0}` para acessar o primeiro elemento do array `Object[]`, passado como segundo argumento do método `reject()`, e conseguir o valor dos minutos que faltam para o usuário criar um novo tópico.

Para o Spring apresentar as mensagens de erros customizadas, vamos precisar injetar um `MessageSource` na classe `ValidationErrorHandler` e recuperar as mensagens por este objeto. No final, a classe ficará como segue:

```
@RestControllerAdvice
public class ValidationErrorHandler {

    @Autowired
    private MessageSource messageSource;

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ValidationErrorsOutputDto handleValidationException(MethodArgumentNotValidException exception) {
        List<ObjectError> globalErrors = exception.getBindingResult().getGlobalErrors();
        List<FieldError> fieldErrors = exception.getBindingResult().getFieldErrors();

        ValidationErrorsOutputDto validationErrors = new ValidationErrorsOutputDto();
        globalErrors.forEach(error -> validationErrors.addError(getErrorMessage(error)));
    }
}
```

```

        fieldErrors.forEach(error -> {
            String errorMessage = getErrorMessage(error);
            validationErrors.addFieldError(error.getField(), errorMessage);
        });

        return validationErrors;
    }

    private String getErrorMessage(ObjectError error) {
        return messageSource.getMessage(error, LocaleContextHolder.getLocale());
    }
}

```

Note que criamos um método privado `getErrorMessage()` para isolar a recuperação da mensagem dado um erro. Repare também que um objeto `Locale` é requerido - essa informação é usada para implementar mensagens de erro internacionalizadas.

Agora, quando tentamos criar um quinto tópico em menos de uma hora, a mensagem customizada aparece:

Code	Details
400	Error: <i>Undocumented</i>
	Response body
	<pre>{ "globalErrorMessages": [], "errors": [{ "field": "shortDescription", "message": "A descrição do tópico deve ter no mínimo 10 caracteres" }, { "field": "shortDescription", "message": "A descrição do tópico deve ser informada" }], "numberOfErrors": 2 }</pre>
	Download

Figura 9.10: custom-message

9.10 PARA SABER MAIS: MENSAGENS INTERNACIONALIZADAS

O Spring possui uma classe interna chamada `ResourceBundleMessageSource`, que acessa e resolve o arquivo `messages.properties`. Caso queiramos internacionalizar as mensagens, podemos criar vários arquivos de mensagens seguindo o padrão de nomenclatura do Spring. Por exemplo, para mensagens em português, devemos criar o arquivo `messages-pt.properties`, para inglês, `messages-en.properties`, etc...

Ou seja, o padrão do nome do arquivo procurado será: a palavra `messages-` seguido da *language tag* da língua desejada e finalizado com a extensão `.properties`:

```
messages-<language tag>.properties
```

A lista de *language tags* pode ser acessada [aqui](#). Veja o exemplo abaixo considerando os arquivos de mensagens em português e inglês:

- `messages-pt.properties`

```
NotBlank.newTopicInputDto.shortDescription=A descrição do tópico deve ser informada  
Size.newTopicInputDto.shortDescription=A descrição do tópico deve ter no mínimo {2} caracteres
```

```
newTopicInputDto.limit.exceeded=0 limite individual de novos tópicos por hora foi excedido. Vo  
cê poderá registrar uma nova dúvida em {0} minutos
```

- `messages_en.properties`

```
NotBlank.newTopicInputDto.shortDescription=The description information is mandatory  
Size.newTopicInputDto.shortDescription=The description must have a minimum of {2} characters
```

```
newTopicInputDto.limit.exceeded=The individual limit of new threads per hour has been exceeded  
. You may report a new issue in {0} minutes
```

Com esses arquivos dentro da aplicação, na pasta `src/main/resources/`, o Spring vai considerar qual arquivo chamar a partir do parâmetro `Accept-Language` vindo pelo cabeçalho da requisição. Esse parâmetro informa qual linguagem o cliente é capaz de entender e qual variante de localidade é preferida. Caso não seja passada a *language tag* ou não existir o respectivo arquivo de mensagens, o Spring considerará o arquivo padrão `messages.properties`.

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

9.11 EXERCÍCIOS: APLICANDO VALIDAÇÕES DE NEGÓCIO E PERSONALIZANDO MENSAGENS

1. Para evitar possíveis *spams* de dúvidas no fórum, um usuário pode criar apenas 4 tópicos a cada hora. Implementar essa validação, que refere-se diretamente ao negócio da aplicação, vai além de utilizar as *constraints* básicas providas pela Bean Validation . Para este tipo de validação precisamos de nosso próprio validador customizado.

No pacote br.com.alura.forum.validator , crie a classe NewTopicCustomValidator que implementa a interface Validator do Spring do pacote org.springframework.validation .

```
public class NewTopicCustomValidator implements Validator {  
  
    @Override  
    public boolean supports(Class<?> clazz) {  
        return NewTopicInputDto.class.isAssignableFrom(clazz);  
    }  
  
    @Override  
    public void validate(Object target, Errors errors) {  
  
        // código de validação  
    }  
}
```

Perceba que o método supports() apenas tem a responsabilidade de responder se a classe (argumento clazz) de um determinado objeto que precisa ser validado tem relação direta com a classe NewTopicCustomValidator , ou seja, se este validador oferece suporte a esse objeto, antes do framework prosseguir aplicando as regras de validação implementadas no método validate() .

2. Precisamos implementar o método validate() contendo as regras da validação de negócio, mas, antes disso, vamos criar a classe que encapsula a lógica de um possível *spam* na aplicação (um novo tópico que excede a quantidade máxima que um usuário pode cadastrar por hora).

No pacote br.com.alura.forum.model , crie a classe PossibleSpam .

```
public class PossibleSpam {  
  
    private List<Topic> topics;  
  
    public PossibleSpam(List<Topic> topics) {  
        this.topics = topics;  
    }  
  
    public boolean hasTopicLimitExceeded() {  
        return this.topics.size() >= 4;  
    }  
  
    public long minutesToNextTopic(Instant from) {  
        Instant instantOfTheOldestTopic = topics.get(0).getCreationInstant();  
  
        return Duration.between(from, instantOfTheOldestTopic)  
            .getSeconds() / 60;
```

```
    }  
}
```

A classe `Duration` deve ser importada do pacote `java.time`.

3. Agora podemos voltar à classe `NewTopicCustomValidator` e à implementação do método `validate()`.

Precisamos obter o volume de tópicos do usuário no intervalo da última hora. Para isso, nossa classe tem como dependências uma instância de `TopicRepository` e uma de `User`, representando o usuário que tenta criar este novo tópico.

Adicione o construtor que recebe a injeção dessas dependências.

```
public class NewTopicCustomValidator implements Validator {  
  
    private final TopicRepository topicRepository;  
    private User loggedUser;  
  
    public NewTopicCustomValidator(TopicRepository topicRepository, User loggedUser) {  
        this.topicRepository = topicRepository;  
        this.loggedUser = loggedUser;  
    }  
  
    // métodos supports() e validate() omitidos  
}
```

4. No método `validate()`, utilize o `TopicRepository` para buscar todos os tópicos criados pelo usuário durante a última hora e criar a instância de `PossibleSpam`.

```
public class NewTopicCustomValidator implements Validator {  
  
    // atributos, construtor e método supports() omitidos  
  
    @Override  
    public void validate(Object target, Errors errors) {  
  
        Instant oneHourAgo = Instant.now().minus(1, ChronoUnit.HOURS);  
        List<Topic> topics = topicRepository  
            .findByOwnerAndCreationInstantAfterOrderByCreationInstantAsc(loggedUser, oneHourAgo);  
  
        PossibleSpam possibleSpam = new PossibleSpam(topics);  
    }  
}
```

Não se esqueça de adicionar o novo método na interface `TopicRepository`, cuja implementação será gerada pelo Spring Data.

5. Caso o limite dos tópicos tenha sido alcançado, devemos rejeitar a criação deste novo tópico. Acrescente essa condicional e utilize o método `reject()` da classe `Errors` para registrar o erro de validação.

```
public class NewTopicCustomValidator implements Validator {  
  
    // atributos, construtor e método supports() omitidos
```

```

@Override
public void validate(Object target, Errors errors) {

    Instant oneHourAgo = Instant.now().minus(1, ChronoUnit.HOURS);
    List<Topic> topics = topicRepository
        .findByOwnerAndCreationInstantAfterOrderByCreationInstantAsc(
            loggedUser, oneHourAgo
        );

    PossibleSpam possibleSpam = new PossibleSpam(topics);

    if (possibleSpam.hasTopicLimitExceeded()) {

        long minutesToNextTopic = possibleSpam.minutesToNextTopic(oneHourAgo);
        errors.reject("newTopicInputDto.limit.exceeded",
                      new Object[] {minutesToNextTopic},
                      "O limite individual de novos tópicos por hora foi excedido");
    }
}
}

```

O primeiro argumento do método `reject()` representa o código de erro que usaremos como chave para acessar uma possível mensagem de erro customizada; o segundo (`Object[]`), representa os argumentos que podemos passar para flexibilizar a mensagem de erro que devolveremos para o usuário; já o terceiro argumento será usado como mensagem de erro padrão.

6. Agora, para que nosso *controller* registre o *validator* que será utilizado pelo Spring durante o processo de *binding*, crie o método `initBinder()` adicionando a instância de `NewTopicCustomValidator` entre os validadores de um novo tópico.

```

@RestController
@RequestMapping("/api/topics")
public class TopicController {

    // outros métodos omitidos

    @InitBinder("newTopicInputDto")
    public void initBinder(WebDataBinder binder, @AuthenticationPrincipal User loggedUser) {
        binder.addValidators(new NewTopicCustomValidator(this.topicRepository, loggedUser));
    }
}

```

O método `initBinder()` recebe o objeto de `WebDataBinder`, que encapsula os detalhes da infraestrutura do processo de *binding* provida pelo Spring MVC. Podemos utilizá-lo, entre outras coisas, para registrar a necessidade de validadores ou conversores de tipo mais específicos ao contexto da aplicação.

7. Para que as mensagens de erros globais sejam apresentadas na resposta para o cliente, acrescente o atributo `globalErrorMessages` na classe `ValidationErrorsOutputDto`. Não esqueça de implementar o *getter* correspondente, o método `addError()` que adiciona mensagens na lista de erros globais e, por fim, considerar esses erros no método `getNumberOfErrors()`:

```
public class ValidationErrorsOutputDto {
```

```

private List<String> globalErrorMessages = new ArrayList<>();
private List<FieldErrorOutputDto> fieldErrors = new ArrayList<>();

public void addError(String message) {
    globalErrorMessages.add(message);
}

public void addFieldError(String field, String message) {
    FieldErrorOutputDto fieldError = new FieldErrorOutputDto(field, message);
    fieldErrors.add(fieldError);
}

public List<String> getGlobalErrorMessages() {
    return globalErrorMessages;
}

public List<FieldErrorOutputDto> getErrors() {
    return fieldErrors;
}

public int getNumberOfErrors() {
    return this.fieldErrors.size() + this.globalErrorMessages.size();
}
}

```

8. Na classe `ValidationErrorHandler` , recupere e acrescente os erros globais no `ValidationErrorsOutputDto` . Além disso, injete um `MessageSource` e modifique o código para que seja considerado o conteúdo do arquivo `messages.properties` que criaremos a seguir com as mensagens customizadas.

No final, o código de sua classe `ValidationErrorHandler` deve ficar parecido com o código abaixo:

```

@RestControllerAdvice
public class ValidationErrorHandler {

    @Autowired
    private MessageSource messageSource;

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ValidationErrorsOutputDto handleValidationException(MethodArgumentNotValidException exception) {

        List<ObjectError> globalErrors = exception.getBindingResult().getGlobalErrors();
        List<FieldError> fieldErrors = exception.getBindingResult().getFieldErrors();

        ValidationErrorsOutputDto validationErrors = new ValidationErrorsOutputDto();

        globalErrors.forEach(error -> validationErrors.addError(getErrorMessage(error)));

        fieldErrors.forEach(error -> {
            String errorMessage = getErrorMessage(error);
            validationErrors.addFieldError(error.getField(), errorMessage);
        });
    }

    return validationErrors;
}

```

```

private String getErrorMessage(ObjectError error) {
    return messageSource.getMessage(error, LocaleContextHolder.getLocale());
}
}

```

9. Para que tenhamos mensagens personalizadas para os erros de validação, crie o arquivo `messages.properties` na pasta `src/main/resources` com o conteúdo desejado.

Exemplo:

`NotBlank.newTopicInputDto.shortDescription=A descrição do tópico deve ser informada
Size.newTopicInputDto.shortDescription=A descrição do tópico deve ter no mínimo {2} caracteres`

`NotBlank.newTopicInputDto.content=O conteúdo da dúvida deve ser informado
Size.newTopicInputDto.content=O conteúdo da dúvida deve ter no mínimo {2} caracteres`

`NotNull.newTopicInputDto.courseId=O id do curso deve ser informado`

`newTopicInputDto.limit.exceeded=O limite individual de novos tópicos por hora foi excedido. Você poderá registrar uma nova dúvida em {0} minutos`

Note que para facilitar a associação das mensagens aos valores rejeitados devemos seguir um padrão para declarar as chaves das mensagens: `<NOME DA ANNOTATION>.<NOME DO OBJETO VALIDADO>.<NOME DO CAMPO>=mensagem`. Em casos como o erro de validação de negócio usamos o mesmo código de erro registrado ao rejeitar o valor: `newTopicInputDto.limit.exceeded`.

10. Rode novamente a aplicação e tente criar um novo tópico com informações inválidas. Perceba como agora as mensagens personalizadas são exibidas.

Code	Details
400	<p>Error: <i>Undocumented</i></p> <p>Response body</p> <pre>{ "globalErrorMessages": [], "errors": [{ "field": "shortDescription", "message": "A descrição do tópico deve ter no mínimo 10 caracteres" }, { "field": "shortDescription", "message": "A descrição do tópico deve ser informada" }], "numberOfErrors": 2 }</pre> <p>Download</p>

Figura 9.11: mensagens personalizadas

11. (Opcional) Adicione outro arquivos de properties com mensagens internacionalizadas, como no exemplo a seguir.

messages_en.properties

```
NotBlank.newTopicInputDto.shortDescription=The description information is mandatory  
Size.newTopicInputDto.shortDescription=The description must have a minimum of {2} characters  
  
NotBlank.newTopicInputDto.content=The topic content is mandatory  
Size.newTopicInputDto.content=The topic content must have a minimum of {2} characters  
  
NotNull.newTopicInputDto.courseId=The course ID must be informed  
  
newTopicInputDto.limit.exceeded=The individual limit of new threads per hour has been exceeded. You may report a new issue in {0} minutes
```

PS: O suporte à internacionalização de mensagens do Spring por padrão atende ao idioma indicado na requisição pelo *HTTP Accept-Language header*.

RESPONDENDO DÚVIDAS E ENVIANDO EMAIL

10.1 SUPER DESAFIO: IMPLEMENTE AS RESPOSTAS EM DÚVIDAS ABERTAS

Além da criação de tópicos, a aplicação deve permitir que outros usuários respondam a um determinado tópico. Agora é com você! Implemente as funcionalidades abaixo de acordo com as especificações requeridas.

1. Antes de adicionar novas respostas a um tópico, a aplicação cliente necessita apresentar a página com os detalhes do mesmo ao usuário. Portanto, acessando o endpoint `/api/topics/{id}` através de uma requisição do tipo `GET`, ela espera receber o seguinte JSON com as informações gerais sobre um tópico.

```
{
  "id": 100,
  "shortDescription": "Descrição curta do tópico",
  "content": "Conteúdo detalhado do tópico",
  "status": "NOT_ANSWERED", // status do tópico
  "creationInstant": "2018-01-01T12:00:00.000Z", // instante em que foi criado
  "lastUpdate": "2018-01-01T12:00:00.000Z", // instante da última atualização

  "courseName": "Nome do curso ao qual o tópico é relacionado",
  "subcategoryName": "Subcategoria do curso",
  "categoryName": "Categoria do curso",
  "ownerName": "Nome do usuário autor do tópico",

  "numberOfResponses": 0, // numero de respostas
  "answers": [ // lista com todas as respostas
    {
      "id": 100,
      "content": "Conteúdo da resposta",
      "creationTime": "2018-01-01T12:00:00.000Z",
      "solution": false, // resposta é a solução para o tópico ?
      "ownerName": "Nome do Usuário autor da resposta"
    }
  ]
}
```

Implemente a funcionalidade seguindo o escopo definido acima.

2. Você deve criar também a funcionalidade para adicionar novas respostas a um determinado tópico. Para realizar tal ação, a aplicação cliente enviará o seguinte JSON representando a nova resposta:

```
{  
    "content" : "Conteúdo da resposta"  
}
```

Além disso, após registrar a nova resposta, o cliente espera receber o JSON abaixo na resposta para sua requisição:

```
{  
    "id": 100,  
    "content": "Conteúdo da resposta",  
    "creationTime": "2018-01-01T12:00:00.000Z",  
    "solution": false, // resposta é a solução para o tópico ?  
    "ownerName": "Nome do Usuário autor da resposta"  
}
```

A aplicação cliente está preparada para efetuar uma requisição do tipo `POST` para `/api/topics/{idDoTopico}/answers`.

A partir das informações acima implemente a funcionalidade.

3. Utilize o *Swagger UI* para testar enquanto desenvolve.

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

10.2 ENVIANDO EMAILS COM SPRING E JAVA MAIL

Com o super desafio resolvido, um usuário já é capaz de responder um tópico existente. Quando essa ação acontecer no fórum, a aplicação deve disparar um email para o dono do tópico avisando que há uma nova resposta.

Precisamos configurar um servidor de email para disparar os emails da aplicação. Vamos utilizar o servidor SMTP do Google por ser gratuito e de fácil configuração. É importante considerar que, por ser gratuito, o Google limita o envio de 100 emails por dia.

10.3 CONFIGURANDO SERVIDOR SMTP DO GOOGLE

Nós já criamos uma conta `fj27.spring@gmail.com`, que será usada pela API do fórum. **Caso você não queira utilizar outro email para aplicação, pule essa parte.**

Para configurar o servidor de email do Google, a primeira coisa a fazer é criar um email no Gmail. Você terá que liberar o acesso para aplicações menos seguras. Vá no painel de configurações de sua conta do Google, escolha o menu de *Segurança* e ative este acesso:

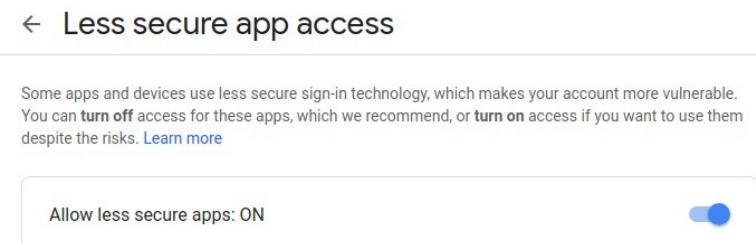


Figura 10.1: google-access-less-security

Ainda no painel de segurança, em *Signing in to Google*, escolha a opção *2-Step Verification*. Uma janela parecida com a imagem abaixo vai aparecer:

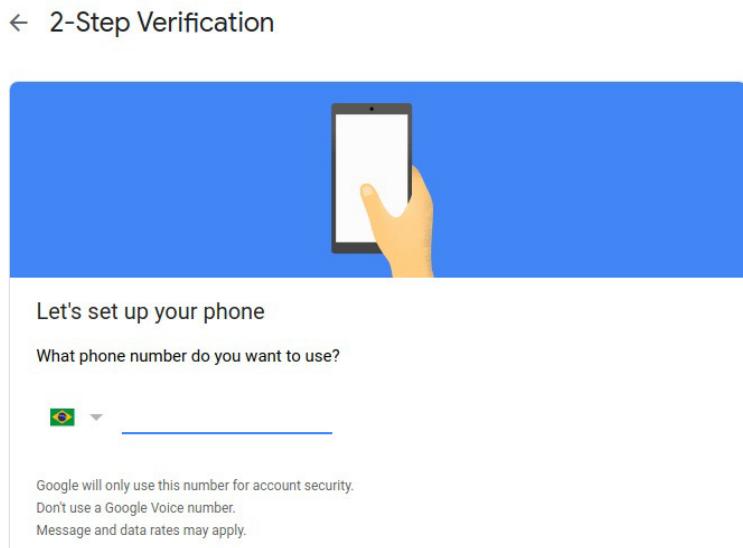


Figura 10.2: 2-step-verification

Preencha com um número de celular para receber o código de verificação via SMS e clique em *NEXT*. Após receber o código, insira-o na próxima página:

← 2-Step Verification

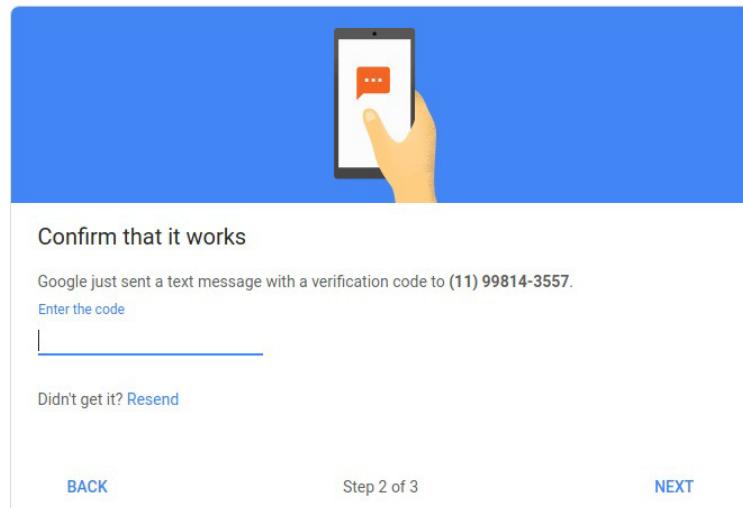


Figura 10.3: confirm-code

Clique em *NEXT* e, se tudo ocorrer como o esperado, você verá a janela abaixo:

← 2-Step Verification



Figura 10.4: worked

Clique em *TURN ON*. De volta ao menu de segurança da sua conta do Google, vá novamente em *Signing in to Google* e clique em *App passwords*. Entre com sua senha novamente para ser redirecionado para a tela abaixo:

Select the app and device you want to generate the app password for.

Select app

▼ Select device ▼

GENERATE

Figura 10.5: set-app

Em *select a app*, selecione *Other*, escreva um nome para o aplicativo e clique em *GENERATE*:

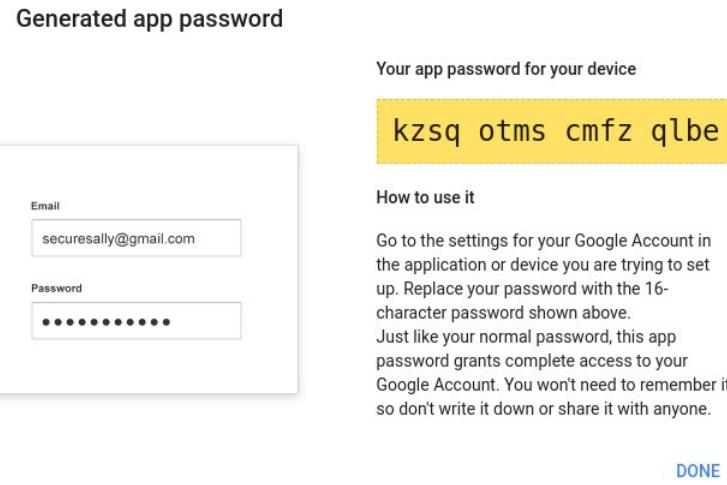


Figura 10.6: generated-pass

Copie a senha de 16 dígitos gerada em um arquivo para ser usada na aplicação. Neste curso já criamos a conta `fj27.spring@gmail.com` e a senha de 16 dígitos gerada foi `ozlwonkcofjwllms`.

10.4 JAVAMAIL

A API JavaMail é usada para enviar e receber emails independente de protocolo (SMTP, POP3,IMAP). Antes da construção da mensagem a ser enviada, é necessário um objeto `Session` que representa uma sessão de email. Mas, antes de criar uma sessão, precisamos definir as propriedades do servidor através de uma instância de `Properties` do `java.util`:

```
Properties props = new Properties();
props.put("mail.smtp.host", "smtp.gmail.com");
props.put("mail.smtp.socketFactory.port", "587");
props.put("mail.smtp.socketFactory.class", "javax.net.ssl.SSLSocketFactory");
props.put("mail.smtp.auth", "true");
props.put("mail.smtp.starttls.enable", "true");
```

Com as propriedades definidas e os dados do email configurados anteriormente, criamos a sessão:

```
Session session = Session.getDefaultInstance(props,
    new javax.mail.Authenticator(){
        protected javax.mail.PasswordAuthentication getPasswordAuthentication() {
            return new PasswordAuthentication("fj27.spring@gmail.com", "ozlwonkcofjwllms");
        }
    });
};
```

Agora, basta criar a mensagem e enviar. O JavaMail possui a interface `Message`, que representa a mensagem a ser enviada. Sua implementação mais usada é a `MimeMessage`, que exige uma `session` para ser instanciada:

```
Message message = new MimeMessage(session);
Address[] toUser = InternetAddress.parse(answer.getTopic().getOwnerEmail());
```

```
message.setTo("email do destinatário");
message.setRecipients(Message.RecipientType.TO, toUser);
message.setSubject("assunto do email");
message.setText("conteúdo do email");
```

A classe `MimeMessage` possui vários métodos para a construção da mensagem, como a lista de destinatários, assunto do email e seu conteúdo. O envio é feito pelo método `send()` da classe `Transport` do JavaMail:

```
Transport.send(message)
```

A mensagem será enviada para todos os destinatários especificados no `array` de `Address` do código anterior. Caso algum erro ocorra, uma `SendFailedException` será lançada.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

10.5 ENVIANDO EMAIL COM SPRING MAIL

Precisamos, antes de tudo, adicionar a dependência do Spring Mail no arquivo `pom.xml`, representada pelo `starter spring-boot-starter-mail`:

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

Esta dependência possui várias classes e interfaces que ajudam a diminuir todo esse código de envio de mensagem. Não precisamos criar uma `Session` e podemos isolar as propriedades do servidor no arquivo `application.properties` usando o `namespace` `spring.mail`:

```
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=fj27.spring@gmail.com
spring.mail.password=ozlwonkcofjwllms
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

Quando o Spring identifica a existência do `host` de email, ele cria uma instância de `MailSender`, que provê funcionalidades para envio de emails. Podemos, então, criar um serviço e pedir essa classe injetada:

```
@Component
public class ForumMailService {

    @Autowired
    private MailSender mailSender;

    public void sendNewReplyMail(Answer answer) {

    }
}
```

Criamos um método `sendNewReplyMail()` recebendo uma `Answer`, já que precisaremos dos dados da resposta que foi criada para construir o email. O Spring possui uma classe chamada `SimpleMailMessage` usada para criar mensagens simples de email:

```
@Component
public class ForumMailService {

    private static final Logger logger = LoggerFactory
        .getLogger(ForumMailService.class);

    @Autowired
    private MailSender mailSender;

    public void sendNewReplyMail(Answer answer) {
        SimpleMailMessage simpleMessage = new SimpleMailMessage();
        simpleMessage.setTo(answer.getTopic().getOwnerEmail());
        simpleMessage.setSubject("Novo comentário em " + answer.getTopic()
            .getShortDescription());
        simpleMessage.setText("Olá " + answer.getTopic().getOwnerName() + "\n\n" +
            "Há uma nova mensagem no fórum! " + answer.getOwnerName() +
            " comentou no tópico: " + answer.getTopic().getShortDescription());

        try{
            mailSender.send(simpleMessage);
        catch(MailException) {
            logger.error("Não foi possível enviar email para " + answer.getTopic()
                .getOwnerEmail(), e.getMessage());
        }
    }
}
```

O método `setTo()` recebe o email do destinatário, ou seja, o dono do tópico; o método `setSubject()` define o assunto do email e o `setText()`, a mensagem a ser enviada. Por fim, enviamos o email através do método `send()` de `MailSender`.

Veja que o Spring nos poupou de grande parte do código escrito anteriormente e focamos apenas na construção da mensagem a ser enviada. Além disso, quando for necessário criar outra mensagem para enviar outro tipo de email, basta pedir um `MailSender` injetado que tudo já está previamente configurado.

Com o serviço de email pronto, podemos chamar seu método logo após a criação da resposta no método `answerTopic()` em `AnswerController`:

```
@RestController
@RequestMapping("/api/topics/{topicId}/answers")
public class AnswerController {

    @Autowired
    private TopicRepository topicRepository;

    @Autowired
    private AnswerRepository answerRepository;

    @Autowired
    private ForumMailService forumMailService;

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
        produces=MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<AnswerOutputDto> answerTopic(@PathVariable Long topicId,
        @Valid @RequestBody NewAnswerInputDto newAnswerDto,
        @AuthenticationPrincipal User loggedUser,
        UriComponentsBuilder uriBuilder) throws MessagingException {

        Topic topic = this.topicRepository.findById(topicId);
        Answer answer = newAnswerDto.build(topic, loggedUser);

        this.answerRepository.save(answer);
        this.forumMailService.sendNewReplyMail(answer);

        //restante do código omitido
    }
}
```

Agora, quando um usuário responde um tópico, um email é enviado para o dono do tópico com uma mensagem padrão logo após a mensagem ser salva no banco. Veja o exemplo da mensagem recebida por email:

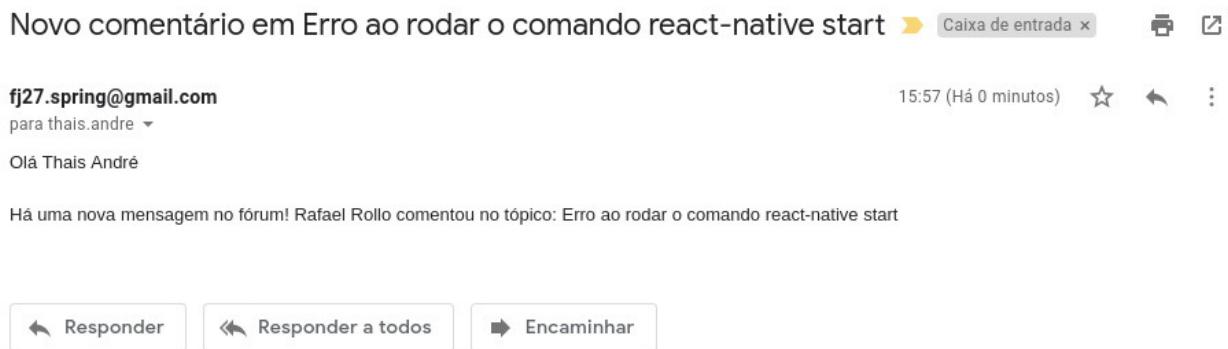


Figura 10.7: simple-message

Podemos encapsular o processamento de uma nova resposta em um serviço já que, além de persistir uma resposta, nosso `controller` está com a responsabilidade de, também, chamar o serviço de email. Vamos criar a classe `NewReplyProcessorService`, que será responsável por processar uma nova resposta:

```

@Service
public class NewReplyProcessorService {

    @Autowired
    private AnswerRepository answerRepository;

    @Autowired
    private ForumMailService forumMailService;

    public void execute(Answer answer) {
        this.answerRepository.save(answer);
        this.forumMailService.sendNewReplyMail(answer);
    }
}

```

Voltando à AnswerController , apagamos a injeção de AnswerRepository e ForumMailService e incluímos a de NewReplyProcessorService :

```

@RestController
@RequestMapping("/api/topics/{topicId}/answers")
public class AnswerController {

    @Autowired
    private TopicRepository topicRepository;

    @Autowired
    private AnswerRepository answerRepository;

    @Autowired
    private ForumMailService forumMailService;

    @Autowired
    private NewReplyProcessorService newReplyProcessorService;

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
                 produces=MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<AnswerOutputDto> answerTopic(@PathVariable Long topicId,
                                                       @Valid @RequestBody NewAnswerInputDto newAnswerDto,
                                                       @AuthenticationPrincipal User loggedUser,
                                                       UriComponentsBuilder uriBuilder) throws MessagingException {

        Topic topic = this.topicRepository.findById(topicId);
        Answer answer = newAnswerDto.build(topic, loggedUser);

        this.answerRepository.save(answer);
        this.forumMailService.sendEmail(answer);

        this.newReplyProcessorService.execute(answer);

        //restante do código omitido
    }
}

```

10.6 PROCESSAMENTO ASSÍNCRONO

Para uma mensagem ser enviada, a aplicação precisa se conectar com o servidor de email para que todo o processo seja executado. Esse processamento gera lentidão na aplicação e é sentido pelo usuário que está respondendo o fórum - dado que a postagem da resposta não é imediata. Para que haja melhora

nesse processamento, podemos habilitar processamentos assíncronos, ou seja, em paralelo.

Portanto, queremos que um usuário, ao responder um tópico, veja sua resposta sendo criada imediatamente, independente se o email já foi enviado ou não. O envio de email será colocado, então, em uma nova *thread* durante o processamento da requisição de uma nova resposta. Enquanto isso, o restante do método `answerTopic()` segue seu fluxo. Para garantir este comportamento, o Spring provê a anotação `@Async`. A utilizaremos no método que deve ser executado em uma nova *thread*, ou seja, executado de maneira assíncrona. Este método será o `sendNewReplyMail()` da nossa `ForumMailService`:

```
@Component
public class ForumMailService {

    // atributos omitidos

    @Async
    public void sendNewReplyMail(Answer answer) {
        // código interior omitido
    }
}
```

Para deixar claro que se trata de um método assíncrono, podemos modificar o nome de método para `sendNewReplyMailAsync()`:

```
@Component
public class ForumMailService {

    // atributos omitidos

    @Async
    public void sendNewReplyMailAsync(Answer answer) {
        // código interior omitido
    }
}
```

E modificar a chamada dentro do método `execute()` do nosso serviço `NewReplyProcessorService` para `sendNewReplyMailAsync()`:

```
public void execute(Answer answer) {
    this.answerRepository.save(answer);
    this.forumMailService.sendNewReplyMailAsync(answer);
}
```

Mas, para que tudo funcione, será preciso habilitar execuções assíncronas no Spring com a anotação `@EnableAsync` na classe principal da aplicação:

```
@EnableAsync
@EnableSpringDataWebSupport
@SpringBootApplication
public class AluraForumApplication {

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

```
}
```

Agora, ao responder um tópico, o processamento de criação de uma nova resposta é imediato e independente do envio de email.

10.7 EXERCÍCIO: ENVIANDO E-MAILS COM SPRING E JAVAMAIL

Para darmos prosseguimento ao projeto, precisaremos contar com as classes criadas no exercício anterior. Como explicado pelo instrutor e por esta apostila, existem n possíveis soluções para o exercício proposto, mas para que possamos prosseguir precisamos de uma implementação base conhecida que sustente os próximos exercícios passo a passo. Por isso, na pasta de arquivos do curso existe uma das possíveis soluções para o exercício já pronta em um projeto, a qual vamos nos basear.

1. Caso você **não** tenha feito o *super desafio* anterior, acesse a pasta do curso em `/caelum/cursos/27` e utilize o projeto `/forum-v3` para dar prosseguimento às atividades.
2. Adicione a dependência do módulo Spring Mail no arquivo `pom.xml`:

```
<dependencies>
    <!-- dependência do spring validation -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-mail</artifactId>
    </dependency>
</dependencies>
```

3. Quando uma nova resposta for adicionada a um tópico, nossa aplicação deve ser capaz de enviar um email para o autor do tópico notificando-o sobre o fato.

No pacote `br.com.alura.forum.service.infra`, crie a classe `ForumMailService` com o método `sendNewReplyMail()`, que recebe a informação da resposta e encapsula os detalhes de preparação e envio da mensagem.

```
@Service
public class ForumMailService {

    private static final Logger logger = LoggerFactory
        .getLogger(ForumMailService.class);

    @Autowired
    private MailSender mailSender;

    public void sendNewReplyMail(Answer answer) {
        Topic answeredTopic = answer.getTopic();

        SimpleMailMessage message = new SimpleMailMessage();
        message.setTo(answeredTopic.getOwnerEmail());
        message.setSubject("Novo comentário em: " + answeredTopic.getShortDescription());

        message.setText("Olá " + answeredTopic.getOwnerEmail() + "\n\n" +
            "Há uma nova mensagem no fórum! " + answer.getOwnerName() +
```

```

        " comentou no tópico: " + answeredTopic.getShortDescription());

    try {
        mailSender.send(message);

    } catch (MailException e) {
        logger.error("Não foi possível enviar email para " + answer.getTopic()
            .getOwnerEmail(), e.getMessage());
    }
}
}

```

4. Para que seja possível a injeção de um `MailSender`, precisamos definir os detalhes de configuração deste *bean*. Como já fizemos anteriormente, vamos nos aproveitar da autoconfiguração provida pelo Spring Boot, adicionando algumas chaves no arquivo `application.properties`:

```

# propriedades anteriores omitidas

# mail properties
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=fj27.spring@gmail.com
spring.mail.password=ozlwonkcofjwllms
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.starttls.enable=true

```

5. Agora que já é possível utilizar nosso serviço de envio de emails do fórum, no pacote `br.com.alura.forum.service`, crie a classe `NewReplyProcessorService`, para que ela encapsule a regra de negócio relativa ao registro de uma nova resposta.

```

@Service
public class NewReplyProcessorService {

    @Autowired
    private AnswerRepository answerRepository;

    @Autowired
    private ForumMailService forumMailService;

    public void execute(Answer answer) {
        this.answerRepository.save(answer);
        this.forumMailService.sendNewReplyMail(answer);
    }
}

```

6. Na classe `AnswerController`, utilize o serviço de processamento de novas resposta ao invés de apenas persistir seus dados através do *repository*.

```

@Controller
@RequestMapping("/api/topics/{topicId}/answers")
public class AnswerController {

    @Autowired
    private TopicRepository topicRepository;

    @Autowired
    private AnswerRepository answerRepository;

```

```

    @Autowired
    private NewReplyProcessorService newReplyProcessorService;

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<AnswerOutputDto> answerTopic(@PathVariable Long topicId,
        @Valid @RequestBody NewAnswerInputDto newAnswerDto,
        @AuthenticationPrincipal User loggedUser,
        UriComponentsBuilder uriBuilder) {

        Topic topic = this.topicRepository.findById(topicId);
        Answer answer = newAnswerDto.build(topic, loggedUser);

        this.answerRepository.save(answer);
        this.newReplyProcessorService.execute(answer);

        URI path = uriBuilder
            .path("/api/topics/{topicId}/answers/{answer}")
            .buildAndExpand(topicId, answer.getId())
            .toUri();

        return ResponseEntity.created(path).body(new AnswerOutputDto(answer));
    }
}

```

7. Teste novamente a funcionalidade na aplicação cliente adicionando uma nova resposta.

Perceba que o envio do email faz o tempo de processamento da requisição ter um aumento considerável. A nova resposta é apresentada na página somente após alguns segundos.

8. Para que não seja preciso esperar o término do envio de email para devolver a resposta ao usuário, podemos processar o envio de forma assíncrona, executando esta etapa em uma nova *Thread*.

Na classe `AluraForumApplication`, adicione a anotação `@EnableAsync` sobre a declaração da classe.

```

@EnableAsync
@EnableSpringDataWebSupport
@SpringBootApplication
public class AluraForumApplication {

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

```

Na classe `ForumMailService`, adicione a anotação `@Async` sobre o método `sendNewReplyMail()` e modifique seu nome para `sendNewReplyMailAsync()`:

```

@Service
public class ForumMailService {

    // atributo omitido

    @Async
    public void sendNewReplyMailAsync(Answer answer) {

```

```
// código interno omitido  
}  
}
```

Não esqueça de também modificar a chamada do método na classe `NewReplyProcessorService`.

9. Teste novamente adicionando uma nova resposta, e perceba que a performance melhora consideravelmente.
10. (Opcional) Para que seja possível testar a aplicação cliente com a adição de novas respostas, no terminal, acesse a pasta do projeto da aplicação cliente `Desktop/fj27-alura-forum-client/`, e execute o comando `git checkout 10RespondendoDuvidas`, para atualizar seu estado.

Com o *client* e a *API* rodando com os novos recursos, teste novamente o funcionamento da app cliente recarregando a página (`http://localhost:3000/`) e veja que é possível acessar a página de detalhes de um tópico para adicionar uma nova resposta.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

10.8 UTILIZANDO TEMPLATES DE EMAIL E TRABALHANDO COM MÚLTIPLOS PROFILES

Vimos como é simples enviar um email com `MailSender`. Porém, a mensagem é muito simples e o email enviado para o usuário poderia ter um template mais elegante como a imagem abaixo:

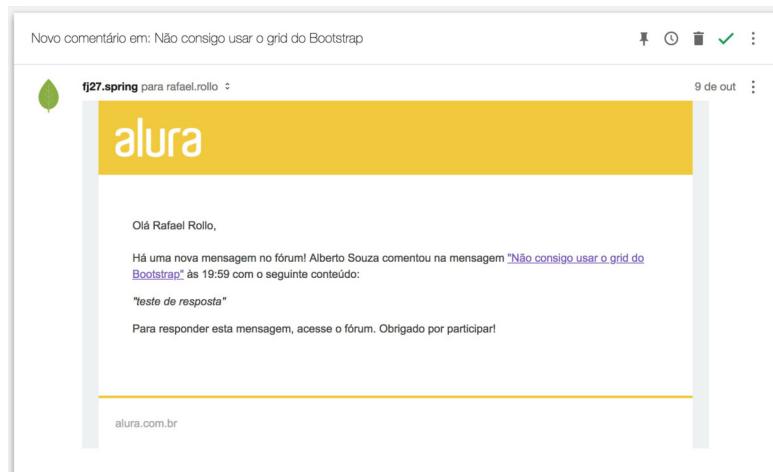


Figura 10.8: template-email

Podemos utilizar qualquer *template engine* para sua construção. Utilizaremos o `Thymeleaf`, que, além de dar suporte ao HTML5, é recomendado pelo próprio Spring para criação de páginas.

10.9 THYMELEAF: BREVE INTRODUÇÃO

O Thymeleaf é muito extensível e permite que você defina seu próprio conjunto de atributos de template, com os nomes que desejar. Ele provê um dialeto padrão que define várias funcionalidades, que serão mais do que suficientes na maioria dos cenários. Você consegue facilmente identificar quando esse dialeto é usado no template já que usa atributos contendo `th` como prefixo, exemplo:

```
<span th:text="....">
```

É possível atribuir as seguintes sintaxes de expressão do Thymeleaf, uma para cada caso:

- `${...}` : variáveis
- `*{...}` : seleções
- `#{...}` : mensagens(i18n)
- `@{...}` : links
- `~{...}` : fragmentos

Integrar com o Spring é fácil e basta inserir a dependência do Thymeleaf no arquivo `pom.xml` - que já foi adicionada ao projeto através do `starter spring-boot-starter-thymeleaf`.

A classe `SpringTemplateEngine` é a classe utilizada para configuração. O Spring Boot já vem com uma configuração padrão e vai procurar todos os templates da aplicação na pasta `src/main/resources/templates` com a extensão `.html` - que são definidos pela classe `ServletContextTemplateResolver`. Ou seja, já irá mapear os nomes das `views` retornadas por `controllers`.

O atributo de tag `th:text=${nome-do-atributo}` é usado para mostrar o valor de um atributo do modelo. Por exemplo, considere o *controller* abaixo:

```
@Controller
public class UserController{

    @GetMapping("/my_page")
    public String showUserDetails(Model model) {
        model.addAttribute("name", "Rafael");
        return "my_page";
    }
}
```

A classe `Model` do Spring MVC é responsável por disponibilizar informações do modelo para a *view*. No primeiro parâmetro do método `addAttribute()` é passado a chave (`name`) para recuperar o valor (`Rafael`) - que é passado como segundo parâmetro. Para apresentar a informação `name` na página `my_page.html`, fazemos:

O nome é ``

Que será renderizado como:

O nome é Rafael

O Thymeleaf também permite executar blocos condicionais e *loops*. Exemplo:

```
<td>
    <span th:if="${user.gender} == 'M'" th:text="Male" />
    <span th:unless="${user.gender} == 'M'" th:text="Female" />
</td>
```

O mesmo resultado acima pode ser obtido usando *switch and case*:

```
<td th:switch="${user.gender}">
    <span th:case="'M'" th:text="Male" />
    <span th:case="'F'" th:text="Female" />
</td>
```

É possível, também, disponibilizar uma lista de usuários para a *view*:

```
List<User> users = this.userRepository.findAll();
model.addAttribute("users", users);
```

Para apresentar o nome de cada usuário presente na lista, utilizamos `th:each` para fazer um *for loop*:

```
<tr th:each="user: ${users}">
    <td th:text="${user.name}" />
</tr>
```

Nosso *template* já está pronto e foi desenvolvido pela equipe de *frontend*. Precisamos disponibilizar algumas informações do modelo e utilizar o atributo de tag `th:text=${nome-do-atributo}` do Thymeleaf dentro da tag `` do HTML para apresentá-las. A parte do código do *template* que teremos que construir será a seguinte:

```

<div style="padding:20px 40px">

    Olá <!-->NOME DO AUTOR DO TOPICO<-->,
    <br>
    <br>Há uma nova mensagem no fórum! <!-->NOME DO AUTOR DA RESPOSTA<--> comentou na mensagem
    <a href="http://localhost:3000/" target="_blank"><!-->DESCRICAÇÃO DO TOPICO<-->"</a>
        às <!-->HORA DE CRIAÇÃO DA RESPOSTA<--> com o seguinte conteúdo:
    <p style="word-wrap:break-word;font-style:italic">
        "<!-->CONTEÚDO DA RESPOSTA<-->"
    </p>
    Para responder esta mensagem, acesse o fórum. Obrigado por participar!
    <br>
    <br>
</div>

```

Portanto, depois das informações disponibilizadas, basta substituir os comentários delimitados por " <!--> <--> " por *tags* do HTML com atributos do Thymeleaf.

10.10 TEMPLATE ENGINE

Antes de construir o template, precisamos disponibilizar as informações da mensagem de email. Como se trata de um template de email e não um página retornada por um *controller*, será preciso contar com a ajuda de outra classe do Thymeleaf chamada `TemplateEngine`.

A classe `TemplateEngine` é a classe principal do Thymeleaf para a execução de templates. Criaremos a classe `NewReplyMailFactory`, que será responsável pela fabricação do template, e pediremos que o Spring injete um `TemplateEngine`:

```

@Component
public class NewReplyMailFactory {

    @Autowired
    private TemplateEngine templateEngine;

    public String generateNewReplyMailContent() {
        return null;
    }
}

```

Agora, precisamos das informações para a construção da mensagem do email. Vamos, portanto, receber um objeto `answer` do tipo `Answer`. Além disso, precisamos disponibilizar essas informações para o contexto do *template*. O Thymeleaf provê uma classe `Context`, que possui um método `setVariable()`, que funciona de forma semelhante ao método `addAttribute()` da `Model` do Spring.

```

Context thymeleafContext = new Context();

thymeleafContext.setVariable("topicOwnerName",
    answer.getTopic().getOwnerName());

```

Repare que o método `setVariable()` também recebe uma chave e um valor. Para apresentar o nome do dono do tópico no template, substituimos o comentário pelo atributo `th:text=${...}`,

passando a chave `topicOwnerName` :

```
<div style="padding:20px 40px">

    Olá <span th:text="${topicOwnerName}"></span>,
    <br>
    <br>Há uma nova mensagem no fórum! <!-->NOME DO AUTOR DA RESPOSTA<--> comentou na mensagem
    <a href="http://localhost:3000/" target="_blank">"<!-->DESCRICAÇÃO DO TÓPICO<-->"</a>
        às <!-->HORA DE CRIAÇÃO DA RESPOSTA<--> com o seguinte conteúdo:
    <p style="word-wrap:break-word;font-style:italic">
        "<!-->CONTEÚDO DA RESPOSTA<-->"
    </p>
    Para responder esta mensagem, acesse o fórum. Obrigado por participar!
    <br>
    <br>
</div>
```

No fim, o código de nossa classe `NewReplyMailFactory` ficará parecido como o conteúdo abaixo:

```
@Component
public class NewReplyMailFactory {

    @Autowired
    private TemplateEngine templateEngine;

    public String generateNewReplyMailContent(Answer answer) {
        Topic answeredTopic = answer.getTopic();

        Context thymeleafContext = new Context();
        thymeleafContext.setVariable("topicOwnerName",
            answeredTopic.getOwnerName());
        thymeleafContext.setVariable("topicShortDescription",
            answeredTopic.getShortDescription());
        thymeleafContext.setVariable("answerAuthor", answer.getOwnerName());
        thymeleafContext.setVariable("answerCreationInstant",
            getFormattedCreationTime(answer));
        thymeleafContext.setVariable("answerContent", answer.getContent());

        return this.templateEngine.process("email-template.html",
            thymeleafContext);
    }

    private String getFormattedCreationTime(Answer answer) {
        return DateTimeFormatter.ofPattern("kk:mm")
            .withZone(ZoneId.of("America/Sao_Paulo"))
            .format(answer.getCreationTime());
    }
}
```

O `template` será preenchido com os demais valores disponibilizados. O método privado `getFormattedCreationTime()` foi preciso para formatar a data de criação do tópico apresentada no `template`. A classe `DateTimeFormatter` é do pacote `java.time`.

O retorno do método `generateNewReplyMailContent()` será a `String` representando o `template` - gerada pelo método `process()` de `TemplateEngine`. Para funcionar, nosso `template` deverá ter o nome `email-template.html` dentro da pasta `src/main/resources/templates`.

Editora Casa do Código com livros de uma forma diferente



Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.

Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.](#)

10.11 ENVIANDO O TEMPLATE DO EMAIL

Com a fábrica de *template* pronta, precisamos modificar nosso serviço de email para apresentar o *template* ao invés da mensagem de texto simples que construímos no exercício anterior. A interface `MailSender` não suporta este tipo de mensagem. A extensão `MIME` (*Multipurpose Internet Mail Extensions*) é uma norma para o formato de mensagens de email e provê mecanismos para envio de diversos tipos de informação como imagens, sons e templates html. Portanto, será necessário um objeto capaz de enviar `MimeMessage`s.

O Spring provê uma sub-interface de `MailSender` chamada `JavaMailSender`, capaz de enviar mensagens de extensão `MIME`. Ao invés de um `MailSender`, pediremos a injeção de um `JavaMailSender`:

```
@Component
public class ForumMailService {

    @Autowired
    private MailSender mailSender;

    @Autowired
    private JavaMailSender mailSender;

    // código omitido
}
```

Agora, quando o Spring detectar um *host* de email na aplicação, vai prover uma implementação de `JavaMailSender`.

Ao enviar o template, será necessário utilizar a classe `MimeMessage` ao invés de `SimpleMailMessage` para a construção da mensagem. É conveniente usar a classe `MimeMessageHelper` do Spring e poupar linhas de código que escreveríamos utilizando apenas a API

do JavaMail. Também usaremos a interface `MimeMessagePreparator` do Spring, para preparar a mensagem. Essa interface possui um método chamado `prepare()`, que recebe uma `MimeMessage`.

```
@Component
public class ForumMailService {

    private static final Logger logger = LoggerFactory
        .getLogger(ForumMailService.class);

    @Autowired
    private JavaMailSender mailSender;

    @Autowired
    private NewReplyMailFactory newReplyMailFactory;

    @Async
    public void sendMaisendNewReplyMailAsync(Answer answer) {

        MimeMessagePreparator mimeMessagePreparator = (mimeMessage) -> {

            MimeMessageHelper messageHelper = new MimeMessageHelper(mimeMessage);

            messageHelper.setTo(answer.getTopic().getOwnerEmail());
            messageHelper.setSubject("Novo comentário em "
                + answer.getTopic().getShortDescription());
            messageHelper.setText(this.newReplyMailFactory
                .generateNewReplyMailContent(answer), true);
        };

        try {
            mailSender.send(mimeMessagePreparator);
        } catch (MailException e) {
            logger.error("Não foi possível enviar email para " + answer.getTopic()
                .getOwnerEmail(), e.getMessage());
        }
    }
}
```

Note que a classe `MimeMessageHelper` possui praticamente os mesmos métodos da `SimpleMailMessage` para a construção de mensagens. A única diferença, neste caso, será o método `setText()`, que, ao invés de receber a mensagem anterior, vai receber nosso template retornado pelo método `generateNewReplyMailContent()`. Além disso, será necessário passar um segundo parâmetro booleano com valor `true`, especificando que se trata de um template `html` e não uma `String` qualquer.

Por fim, utilizamos o `JavaMailSender` para enviar a mensagem. A interface possui um método `send()` que recebe um `MimeMessagePreparator`:

```
mailSender.send(mimeMessagePreparator);
```

Agora, ao testar o envio de uma nova resposta, o template será considerado para renderizar a mensagem no corpo do email.

10.12 TRABALHANDO COM MÚLTIPLOS PROFILES

Até o momento, cada nova dúvida que é adicionada durante um teste de funcionalidade dispara um envio de email, que é processado pelo servidor de emails que será usado em produção. Assim, com o volume de envios gerado a cada repetição de um teste, consumiríamos muito do recurso dedicado aos nossos clientes.

Para ter um ambiente de desenvolvimento capaz de suportar a execução simulando o mesmo envio de email, vamos utilizar o Mock SMTP Server Green Mail, para fins de desenvolvimento. Um Mock é um objeto *fake* que simula o comportamento de uma outra classe. Precisaremos, portanto, das dependências de Green Mail na aplicação:

```
<dependency>
    <groupId>com.icegreen</groupId>
    <artifactId>greenmail</artifactId>
    <version>1.5.8</version>
</dependency>
```

As configurações do Green Mail ficarão em uma classe de configuração na aplicação:

```
@Configuration
public class GreenMailLocalSsmtpConfiguration {

}
```

Teremos que passar as propriedades de um servidor de email, como as do Google, para configurar o Green Mail.

```
@Configuration
public class GreenMailLocalSsmtpConfiguration {

    @Value("${spring.mail.host}")
    private String hostAddress;

    @Value("${spring.mail.port}")
    private String port;

    @Value("${spring.mail.username}")
    private String username;

    @Value("${spring.mail.password}")
    private String password;
}
```

A anotação `@Value` do Spring, usada nos atributos, indica um valor inicial padrão. Usaremos a linguagem de expressão do Spring - SpEL (*Spring Expression Language*) - para indicar que os valores serão definidos no arquivo `application.properties`, referenciados com a mesma chave definida na anotação. Modificaremos as propriedades de email do arquivo `application.properties` para utilizar o servidor *fake* do Green Mail ao invés do servidor real do Google:

```
# mail properties
spring.mail.protocol=smtp
spring.mail.host=localhost
```

```

spring.mail.port=3025
spring.mail.username=fj27.spring@gmail.com
spring.mail.password=123456

```

Será necessário iniciar o servidor do Green Mail. A biblioteca provê a classe utilitária `GreenMail`, que gerencia um servidor e que exige um `setup` inicial para ser instanciada. O `setup` é representado pela classe `ServerSetup`, também do Green Mail, em que passamos as configurações do servidor como a porta, o `host` e o protocolo:

```

ServerSetup serverSetup = new ServerSetup("port", "host", "protocol");
GreenMail smtpServer = new GreenMail(serverSetup);

```

Criaremos o atributo do tipo `GreenMail` e o método `setup()` dentro de `GreenMailLocalSmtpConfiguration` para iniciar o servidor do Green Mail:

```

@Configuration
public class GreenMailLocalSmtpConfiguration {

    private GreenMail smtpServer;

    @Value("${spring.mail.host}")
    private String hostAddress;

    @Value("${spring.mail.port}")
    private String port;

    @Value("${spring.mail.username}")
    private String username;

    @Value("${spring.mail.password}")
    private String password;

    public void setup() {
        ServerSetup serverSetup = new ServerSetup(Integer.parseInt(this.port),
            this.hostAddress, "smtp");

        this.smtpServer = new GreenMail(serverSetup);
        this.smtpServer.setUser(username, username, password);
        this.smtpServer.start();
    }
}

```

Dentro do método `setup()`, depois de configurar o servidor, foi preciso definir o usuário do servidor pelo método `setUser()` e iniciá-lo com o método `start()`. Para que o método `setup()` funcione e os atributos da classe `GreenMailLocalSmtpConfiguration` sejam inicializados pela anotação `@Value`, será necessário o uso da anotação `@PostConstructor` no método. O Spring, após inicializar corretamente tudo, chama o método que estiver anotado com `@PostConstruct`.

```

@PostConstructor
public void setup() {
    ServerSetup serverSetup = new ServerSetup(Integer.parseInt(this.port),
        this.hostAddress, "smtp");

    this.smtpServer = new GreenMail(serverSetup);
    this.smtpServer.setUser(username, username, password);
    this.smtpServer.start();
}

```

```
}
```

Criaremos também o método `destroy()` para parar o servidor:

```
@PreDestroy  
public void destroy() {  
    this.smtpServer.stop();  
}
```

A anotação `@PreDestroy` é usada em métodos como uma notificação de retorno de chamada para sinalizar que a instância está sendo removida pelo contêiner. Geralmente é usada para liberar recursos após descartar um objeto gerenciado pelo Spring.

Com a configuração do Green Mail pronta, avisaremos ao Spring que essa configuração será considerada apenas em ambiente de desenvolvimento. Para isso, anotamos a classe com `@Profile` especificando o ambiente:

```
@Profile("dev")  
@Configuration  
public class GreenMailLocalSmtpConfiguration {  
  
    // código interior omitido  
}
```

Em produção, deverá ser considerado o servidor real do Google. O arquivo `application.properties` é, por padrão, considerado em ambiente de desenvolvimento pelo Spring. Vamos, então, criar um outro arquivo de configuração que será utilizado em ambiente de produção. Basta criarmos o arquivo `application-prod.properties` em `src/main/resources` com o mesmo conteúdo de `application.properties`, modificando apenas as configurações de email:

application-prod.properties

```
# data source  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  
spring.datasource.url=jdbc:mysql://localhost:3306/fj27_spring?createDatabaseIfNotExist=true&useSSL=false  
spring.datasource.username=root  
spring.datasource.password=caelum  
  
# jpa properties  
spring.jpa.hibernate.ddl-auto=update  
spring.jpa.properties.hibernate.show_sql=true  
spring.jpa.properties.hibernate.format_sql=true  
  
# jwt info  
alura.forum.jwt.secret=rm'!@N=Ke!~p8VTA2ZRK~nMDQX5Uvm!m'D&]{@Vr?G;2?XhbC:Qa#9#eMLN\}x3?JR3.2zr~v)gYF^  
8\:8>:xFB:Ww75N/emt9Yj[bQMNCWwW\J?N, nvH.<2\.r~w]*e~vgak)X"v8H`MH/7"2E` ,^k@n<vE-wD3g9JWPy;CrY*.Kd2_D])  
=><D?YhBaSua5hW%{2]_FVXzb9`8FH^b[X3jzVER:&jw2<=c38=>L/zBq`}C6tT*cCSVc^c]-L}&/  
alura.forum.jwt.expiration=604800000  
  
# mail properties  
spring.mail.host=smtp.gmail.com  
spring.mail.port=587  
spring.mail.username=fj27.spring@gmail.com  
spring.mail.password=ozlwonkcofjwllms
```

```
spring.mail.properties.mail.smtp.auth=true  
spring.mail.properties.mail.smtp.starttls.enable=true
```

Caso você deseje subir a aplicação em ambiente de produção, basta avisar isso ao Spring ativando o profile desejado no arquivo **application.properties**:

```
# ativa o profile  
spring.profiles.active=prod
```

O *profile* também pode ser passado como argumento de linha de comando. Podemos deixar um *default* no arquivo `application.properties` e quando executar a aplicação passar o argumento `-Dspring.profiles.active=prod` pela linha de comando - no Eclipse é possível definir este argumento em *Run Configurations*.

Como não queremos disparar emails a todo momento, vamos deixar ativado o ambiente de desenvolvimento:

```
# ativa o profile  
spring.profiles.active=dev
```

10.13 EXERCÍCIOS: UTILIZANDO TEMPLATES DE EMAIL E TRABALHANDO COM MULITIPLOS PROFILES

Na aplicação real do forum da Alura, os emails enviados tem um design mais interessante contando com a identidade visual da aplicação. Para isso enviamos um conteúdo HTML como corpo do email e utilizamos processadores de templates para processar os dados dinamicamente.

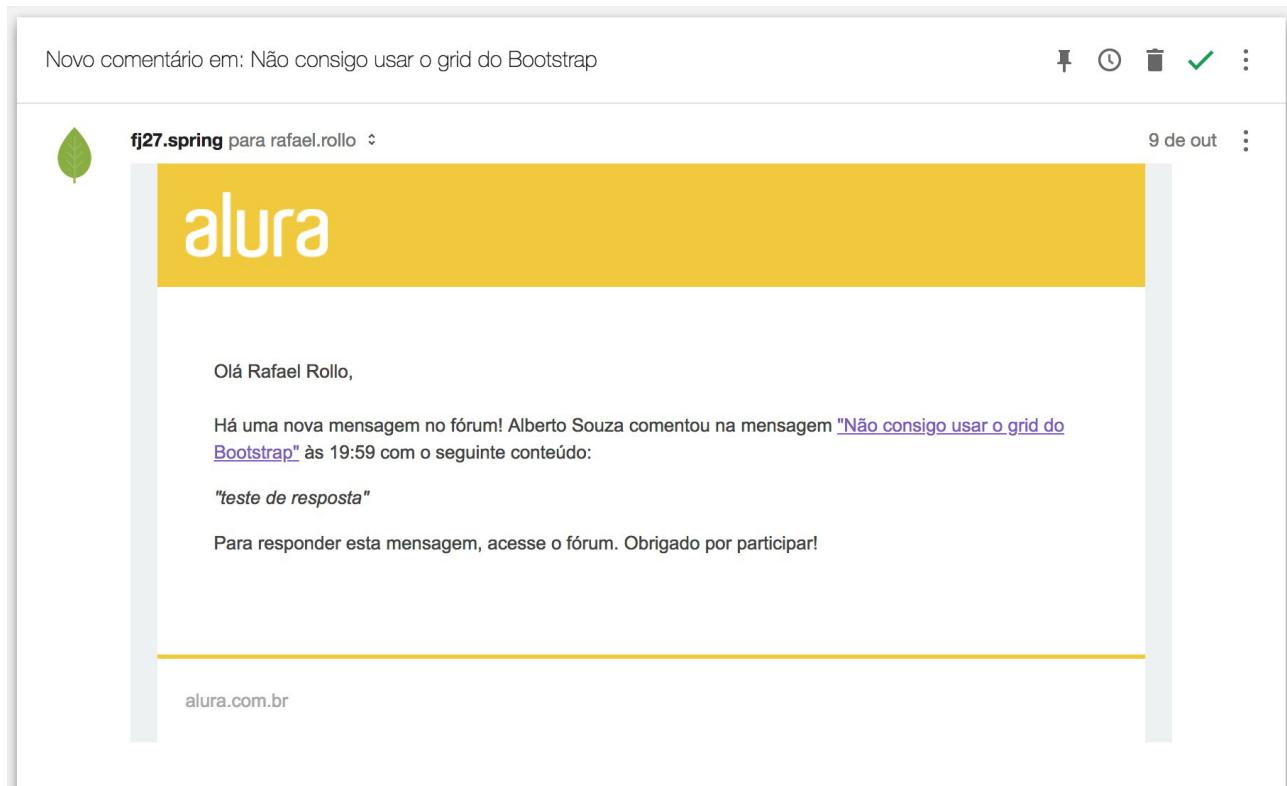


Figura 10.9: Email do Forum

1. Primeiro, adiciona a dependência do *Thymeleaf* no arquivo `pom.xml`:

```
<dependencies>
    <!-- dependência do spring mail -->

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

2. Na pasta de arquivos do curso, existe o template html quase pronto. Copie o arquivo em `/caelum/cursos/27/html/email-template.html` para a pasta `src/main/resources/templates` do seu projeto.

Conteúdo do template html:

```
<div style="background-color:#eef1f2;color:#444;font-family:Arial,sans-serif">
    <div style="width:95%;display:block;margin:0 auto;background-color:#fff">
        <div style="padding:20px;margin-bottom:30px;background-color:#f2ca26">
            <a href="http://localhost:3000/" target="_blank">
                
            </a>
        </div>
        <div style="padding:20px 40px">Olá <!-->NOME DO AUTOR DO TÓPICO<-->,
            <br>
            <br> Há uma nova mensagem no fórum! <!-->NOME DO AUTOR DA RESPOSTA<--> comentou na me
nsagem
```

```

<a href="http://localhost:3000/" target="_blank"><!--DESCRICAÇÃO DO TÓPICO--></a>
    às <!-->HORA DE CRIAÇÃO DA RESPOSTA<--> com o seguinte conteúdo:
<p style="word-wrap:break-word;font-style:italic">
    <!-->CONTEÚDO DA RESPOSTA<-->
</p>
Para responder esta mensagem, acesse o fórum. Obrigado por participar!
<br>
<br>
</div>
<div style="padding-top:20px;margin-top:30px;border-top:solid 3px #f2ca26;padding:20px">
    <a style="color:#a9a9a9;text-decoration:none" href="http://alura.com.br" target="_blank">
        data-saferedirecturl="https://www.google.com/url?q=http://alura.com.br&urce=gmail&ust=1538850140649000&usg=AFQjCNGguW7ViyKGBaLYPstv71szto2E0w">
            <span class="lg">alura</span>.com.br</a>
    </div>
</div>
</div>

```

3. Altere o código do template de email, para que sejam substituídos os pontos de comentário, como por exemplo <!-->NOME DO AUTOR DO TÓPICO<--> , pelo código necessário para capturar os valores dinamicamente.

Exemplo:

No lugar de Olá <!-->NOME DO AUTOR DO TÓPICO<--> , teremos Olá .

Faça as alterações para todas as informações necessárias. Para o nome do autor do tópico usaremos a variável `topicOwnerName` , para o nome do autor da resposta, `answerAuthor` . Para descrição do tópico, instante de criação e conteúdo da resposta usaremos: `topicShortDescription` , `answerCreationInstant` e `answerContent` , respectivamente.

4. Para processar o template e gerar o conteúdo que deve ser enviado no corpo do email, precisamos contar com a implementação de `TemplateEngine` do Thymeleaf. Crie então a classe `NewReplyMailFactory` , no pacote `br.com.alura.forum.infra` . Ela deve encapsular os detalhes da geração do conteúdo.

```

@Component
public class NewReplyMailFactory {

    @Autowired
    private TemplateEngine templateEngine;

    public String generateNewReplyMailContent(Answer answer) {
        Topic answeredTopic = answer.getTopic();

        Context thymeleafContext = new Context();
        thymeleafContext.setVariable("topicOwnerName",
            answeredTopic.getOwnerName());
        thymeleafContext.setVariable("topicShortDescription",
            answeredTopic.getShortDescription());
        thymeleafContext.setVariable("answerAuthor", answer.getOwnerName());
        thymeleafContext.setVariable("answerCreationInstant",

```

```

        getFormattedCreationTime(answer));
thymeleafContext.setVariable("answerContent", answer.getContent());

return this.templateEngine.process("email-template.html",
        thymeleafContext);
}

private String getFormattedCreationTime(Answer answer) {
    return DateTimeFormatter.ofPattern("kk:mm")
        .withZone(ZoneId.of("America/Sao_Paulo"))
        .format(answer.getCreationTime());
}
}

```

A classe `Context` deve ser importada do pacote `org.thymeleaf.context`.

5. Com a *Factory* do conteúdo pronta, podemos modificar a implementação do serviço de envio de email, para que seja possível enviar um conteúdo de mais alto nível.

Primeiro, ao invés de um `MailSender`, prediremos a injeção de um `JavaMailSender` que é capaz de enviar mensagens de extensão MIME :

```

@Component
public class ForumMailService {

    @Autowired
    private MailSender mailSender;

    @Autowired
    private JavaMailSender mailSender;

    // código omitido
}

```

Agora, será necessário utilizar uma `MimeMessage` no lugar de `SimpleMailMessage`, que por padrão suporta apenas o envio de texto puro. Altere a implementação para usar as classes utilitárias `MimeMessagePreparator` e `MimeMessageHelper` para construir a mensagem adequadamente.

```

@Service
public class ForumMailService {

    private static final Logger logger = LoggerFactory
        .getLogger(ForumMailService.class);

    @Autowired
    private JavaMailSender mailSender;

    @Autowired
    private NewReplyMailFactory newReplyMailFactory;

    @Async
    public void sendNewReplyMail(Answer answer) {
        Topic answeredTopic = answer.getTopic();

        MimeMessagePreparator messagePreparator = (mimeMessage) -> {
            MimeMessageHelper messageHelper = new MimeMessageHelper(mimeMessage);

            messageHelper.setTo(answeredTopic.getOwnerEmail());
        };
    }
}

```

```

        messageHelper.setSubject("Novo comentário em: " +
            answeredTopic.getShortDescription());

        String messageContent = this.newReplyMailFactory
            .generateNewReplyMailContent(answer);
        messageHelper.setText(messageContent, true);
    };

    try {
        mailSender.send(messagePreparator);

    } catch (MailException e) {
        logger.error("Não foi possível enviar email para " + answer.getTopic()
            .getOwnerEmail(), e.getMessage());
    }
}
}

```

6. Teste novamente adicionando uma nova resposta, e perceba que o email enviado tem um formato muito mais interessante.
7. Até o momento cada nova dúvida que é adicionada durante um teste de funcionalidade, dispara um envio de email que é processado pelo servidor de emails usado em produção. Assim, com o volume de envios gerado a cada repetição de um teste, consumiríamos muito do recurso dedicado aos nossos clientes.

Para termos um ambiente de desenvolvimento capaz de suportar a execução simulando o mesmo envio de email, vamos utilizar o *Mock SMTP Server Green Mail*, com fins de desenvolvimento.

Adicione a dependência da lib GreenMail ao arquivo `pom.xml`.

```

<dependencies>
    <!-- dependência do thymeleaf -->

    <dependency>
        <groupId>com.icegreen</groupId>
        <artifactId>greenmail</artifactId>
        <version>1.5.8</version>
    </dependency>
</dependencies>

```

8. Crie a classe `GreenMailLocalSmtpConfiguration`, no pacote de classes de configuração, com os métodos necessários para subir o servidor ao rodar a aplicação, e destruir ao encerrar.

```

@Configuration
public class GreenMailLocalSmtpConfiguration {

    private GreenMail smtpServer;

    @Value("${spring.mail.host}")
    private String hostAddress;

    @Value("${spring.mail.port}")
    private String port;

    @Value("${spring.mail.username}")
    private String username;

```

```

    @Value("${spring.mail.password}")
    private String password;

    @PostConstruct
    public void setup() {
        ServerSetup serverSetup = new ServerSetup(Integer.parseInt(this.port),
            this.hostAddress, "smtp");

        this.smtpServer = new GreenMail(serverSetup);
        this.smtpServer.setUser(username, username, password);
        this.smtpServer.start();
    }

    @PreDestroy
    public void destroy() {
        this.smtpServer.stop();
    }
}

```

Para que a configuração seja executada apenas em ambientes de desenvolvimento, adicione a essa classe a anotação que define um perfil de dev .

```

@Profile("dev")
@Configuration
public class GreenMailLocalSmtpConfiguration {

    // implementação omitida
}

```

PS: Perceba que novamente utilizamos o recurso de injeção de valores presentes nos arquivos properties com a anotação @Value.

9. Agora, como temos configurações diferentes para ambientes de desenvolvimento (dev) e produção (prod), precisamos ter propriedades de configuração separadas por perfil.

Crie um segundo arquivo *properties* em `src/main/resources` chamado `application-prod.properties`, e cole em seu corpo o conteúdo atual do arquivo `application.properties`.

application-prod.properties

```

# data source
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/fj27_spring?createDatabaseIfNotExist=true&useSSL=false
spring.datasource.username=root
spring.datasource.password=caelum

# jpa properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true

# jwt info
alura.forum.jwt.secret=rm'!@N=Ke!~p8VTA2ZRK~nMDQX5Uvm!m'D&]{@Vr?G;2?XhbC:Qa#9#eMLN\}x3?JR3.2zr~v)
gYF^8\:>:xFB:Ww75N/emt9Yj[bQNCWwW\J?N, nvH.<2\..r~w]*e~vgak)X"v8H`MH/7"2E` ,^k@n<vE-wD3g9JWPy;CrY*
Kd2_D])=><D?YhBaSua5h%{2}_FVXzb9`8FH^b[X3jzVER&:jw2<=c38=>L/zBq`}C6tT*cCSVc^c]-L}&/
alura.forum.jwt.expiration=604800000

```

```
# mail properties
spring.mail.host=smtp.gmail.com
spring.mail.port=587
spring.mail.username=fj27.spring@gmail.com
spring.mail.password=ozlwonkcofjwllms
spring.mail.properties.mail.smtp.auth=true
spring.mail.properties.mail.smtp.starttls.enable=true
```

10. No arquivo `application.properties`, usado no ambiente de desenvolvimento, altere as propriedades de configuração do `JavaMailSender`, para apontar para o STMP local do GreenMail.

application.properties

```
# propriedades anteriores omitidas

# mail properties
spring.mail.protocol=smtp
spring.mail.host=localhost
spring.mail.port=3025
spring.mail.username=fj27.spring@gmail.com
spring.mail.password=123456
```

11. Rode novamente a aplicação e responda um tópico de seu usuário.

Perceba que dessa vez tudo corre normalmente, porém nada é enviado ao seu email. Isso acontece porque o servidor de email mock que estamos usando recebe o email que a aplicação deseja enviar, mas não o envia ao destinatário.

PS: Caso seja necessário obter os emails enviados ao servidor SMTP fake do GreenMail é possível utilizar métodos específicos de sua API, como por exemplo o método `getReceivedMessages()` de uma instância de GreenMail.

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

AGENDANDO TAREFAS E RENDERIZANDO VIEWS COM THYMELEAF

Nossa aplicação deve ser capaz de gerar um relatório com todas as dúvidas que permanecem sem resposta ao fim de cada dia. Para isso, vamos agendar uma tarefa que busca todas as dúvidas abertas em determinada hora do dia e registra o histórico no banco de dados.

Criaremos a classe `RegisterUnansweredTopicsTask` com um método `execute()` responsável por executar essa tarefa:

```
public class RegisterUnansweredTopicsTask{
}
```

Antes de implementar este método, será preciso buscar por todos os tópicos abertos por categoria. Além disso, será necessário existir este histórico na aplicação. Vamos criar uma entidade, `OpenTopicByCategory`, que representa cada registro deste histórico, contendo, além do `id`, o nome da categoria, o número de tópicos abertos dessa categoria e a data de registro:

```
@Entity
public class OpenTopicByCategory {

    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    private Long id;

    private String categoryName;

    private int topicCount;

    private LocalDate date;

    @Deprecated
    public OpenTopicByCategory() {
    }

    public OpenTopicByCategory(String categoryName, Number topicCount,
        Date instant) {
        this.categoryName = categoryName;
        this.topicCount = topicCount.intValue();
        this.date = instant.toInstant()
            .atZone(ZoneId.systemDefault())
            .toLocalDate();
    }
}
```

```

    }

    // getters necessários omitidos
}

```

11.1 PROJECTIONS

Depois de criada a entidade, precisamos, dentro de todos os tópicos, buscar pelos tópicos abertos por cada categoria. Portanto, criaremos esta *query* dentro de `TopicRepository`. Nosso `TopicRepository` retorna uma ou múltiplas instâncias do objeto que ele gerencia, ou seja, `Topic` - acontece que, agora, desejamos retornar uma lista de `OpenTopicsByCategory` de acordo com alguns de seus atributos.

O Spring Data permite essa flexibilidade e modela tipos de retorno criando **projeções** (*projections*) baseadas em alguns atributos do tipo gerenciado pelo *repository*. Podemos, então, criar a *query* abaixo usando *projections*:

```

@Query("select new br.com.alura.forum.model.OpenTopicByCategory(" +
        "t.course.subcategory.category.name as categoryName, " +
        "count(t) as topicCount, " +
        "now() as instant) from Topic t " +
        "where t.status = 'NOT_ANSWERED' " +
        "group by t.course.subcategory.category")
List<OpenTopicByCategory> findOpenTopicsByCategory();

```

Note que o objeto `OpenTopicByCategory` é instanciado e montado durante a execução do comando *select* na tabela de tópicos, considerando o *status NOT_ANSWERED* e agrupando pelo nome da categoria. Portanto, o retorno será um `List<OpenTopicByCategory>`, como queríamos.

Depois de buscar por todos os tópicos abertos, será preciso salvá-los. Criaremos o `OpenTopicByCategoryRepository` com o método `saveAll()`, passando a lista de tópicos que recuperamos anteriormente:

```

public interface OpenTopicByCategoryRepository
    extends Repository<OpenTopicByCategory, Long> {

    void saveAll(Iterable<OpenTopicByCategory> topics);
}

```

Agora, basta executar as *queries* dentro do método `execute()` de nossa *task*:

```

@Component
public class RegisterUnansweredTopicsTask{

    @Autowired
    private TopicRepository topicRepository;

    @Autowired
    private OpenTopicByCategoryRepository openTopicByCategoryRepository;

    public void execute() {
        List<OpenTopicByCategory> topics =
            this.topicRepository.findOpenTopicsByCategory();
        this.openTopicByCategoryRepository.saveAll(topics);
    }
}

```

```
    }  
}
```

Para que o Spring consiga injetar os *repositories*, foi preciso anotar a classe com `@Component` - ou seja, essa classe também passa a ser gerenciada pelo Spring. Com nossa tarefa pronta, agora será preciso que este código seja executado em determinado horário do dia - no nosso caso, será rodado todos os dias às 20h. Como fazer isso?

Saber inglês é muito importante em TI



Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

[Pratique seu inglês na Alura Língua.](#)

11.2 AGENDANDO TAREFAS COM JAVA

Existem classes do pacote `java.util` do Java que nos ajudam nessa tarefa. Primeiro, vamos precisar de um `Timer`, que facilita o agendamento de tarefas futuras utilizando *threads*. A classe `Timer` nos permite programar tarefas:

```
Timer timer = new Timer();
```

Ao instanciar um `Timer`, uma *thread* é aberta. Outra classe, a `TimerTask`, representa a tarefa a ser agendada por um `Timer`. `TimerTask` é uma classe abstrata, portanto proveremos uma classe concreta. Como `TimerTask` implementa a interface `Runnable`, nos obrigará a implementar o método `run()`:

```
public class ScheduledTask extends TimerTask {  
  
    @Override  
    public void run() {  
        System.out.println("executando uma tarefa");  
    }  
}
```

Por fim, precisamos agendar a tarefa:

```

@Component
public class MyTask() {

    public void execute() {
        Timer timer = new Timer();
        ScheduledTask task = new ScheduledTask();
        timer.schedule(task, 0, 1000);
    }
}

```

O código acima será executado a cada 1 segundo. Para rodar às 20h de cada dia, criamos um `LocalTime` com a hora específica e convertemos para um `Date`, já que o método `schedule()`, além de um `long`, aceita um `Date` representando o tempo que deve ser executado:

```

@Component
public class MyTask() {

    @PostConstructor
    public void execute() {
        LocalTime localTime = LocalTime.of(20, 0, 0);
        long timeInMillis = localTime.getLong(ChronoField.CLOCK_HOUR_OF_DAY);
        Date time = new Date(timeInMillis);

        Timer timer = new Timer();
        ScheduledTask task = new ScheduledTask();
        timer.schedule(task, time);
    }
}

```

Anotamos a classe com `@Component` e o método com `@PostConstructor` para o Spring executar o método assim que inicializar a classe `MyTask`. Dessa maneira, é possível agendar uma tarefa em um horário específico. A classe `TimerTask` é bastante útil, mas o Spring facilita bastante todo esse processo com seu recurso de agendamento.

11.3 AGENDANDO TAREFAS COM SPRING

Agendar tarefas com Spring é bem mais simples e poupa grande parte da escrita de código necessária quando usamos apenas as classes internas do Java. Além disso, abrir *threads* na mão deve ser desencorajado a fim de não causar nenhum conflito com outros *beans* gerenciados pelo próprio Spring.

O Spring possui a anotação `@Schedule`, que deve ser usada nos métodos em que o processamento será agendado. Voltando à nossa tarefa de registrar os tópicos abertos, devemos usar a anotação no método `execute()`:

```

@Component
public class RegisterUnansweredTopicsTask{

    // atributos omitidos

    @Scheduled
    public void execute() {
        List<OpenTopicByCategory> topics =
            this.topicRepository.findOpenTopicsByCategory();
    }
}

```

```

        this.openTopicByCategoryRepository.saveAll(topics);
    }
}

```

Para que funcione, será necessário também habilitar a capacidade de execução de tarefas agendadas no Spring, anotando a classe principal com `@EnableScheduling`:

```

@EnableScheduling
@EnableAsync
@EnableSpringDataWebSupport
@SpringBootApplication
public class AluraForumApplication {

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

```

Mas, ao executar o código, recebemos a seguinte exceção:

```

Caused by: java.lang.IllegalStateException: Encountered invalid @Scheduled method 'execute': Exactly one of the 'cron', 'fixedDelay(String)', or 'fixedRate(String)' attributes is required
at org.springframework.scheduling.annotation.ScheduledAnnotationBeanPostProcessor.processScheduled(ScheduledAnnotationBeanPostProcessor.java:496) ~[spring-context-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at org.springframework.scheduling.annotation.ScheduledAnnotationBeanPostProcessor.lambda$null$1(ScheduledAnnotationBeanPostProcessor.java:359) ~[spring-context-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at java.lang.Iterable.forEach(Iterable.java:75) ~[na:1.8.0_201]
at org.springframework.scheduling.annotation.ScheduledAnnotationBeanPostProcessor.postProcessAfterInitialization$2(ScheduledAnnotationBeanPostProcessor.java:359) ~[spring-context-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at org.springframework.LinkedHashMapForEach.<lambda>(LinkedHashMap.java:684) ~[na:1.8.0_201]
at org.springframework.scheduling.annotation.ScheduledAnnotationBeanPostProcessor.postProcessAfterInitialization(ScheduledAnnotationBeanPostProcessor.java:358) ~[spring-context-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.postProcessorsAfterInitialization(AbstractAutowireCapableBeanFactory.java:434) ~[spring-beans-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.initializeBean(AbstractAutowireCapableBeanFactory.java:1749) ~[spring-beans-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanFactory.java:576) ~[spring-beans-5.1.4.RELEASE.jar:5.1.4.RELEASE]
... 20 common frames omitted

```

Figura 11.1: scheduled-exception

O erro diz: "*Exactly one of the 'cron', 'fixedDelay(String)', or 'fixedRate(String)' attributes is required*" - ou seja, o atributo que representa o tempo que esta tarefa será executada é obrigatório. De fato, não tem como o Spring adivinhar de quanto em quanto tempo esta tarefa deve ser executada e precisamos dizer isso a ele. Caso a tarefa seja executada a cada segundo, podemos usar o parâmetro `fixedDelay`:

```

@Scheduled(fixedDelay = 1000)
public void execute() {
    List<OpenTopicByCategory> topics =
        this.topicRepository.findOpenTopicsByCategory();
    this.openTopicByCategoryRepository.saveAll(topics);
}

```

O parâmetro `fixedDelay` garante que haja um atraso de um valor passado em milissegundos entre o tempo de término de uma execução de uma tarefa e o horário de início da próxima execução da tarefa. Outro parâmetro que podemos usar é o `fixedRate`, que executa a tarefa agendada a cada valor em milissegundos definido.

Por vezes, `fixedDelay` e `fixedRate` não são suficientes, e precisamos da flexibilidade maior que uma *cron expression* (expressão *cron*) pode oferecer para controlar o cronograma de nossas tarefas. *Cron* é um utilitário disponível em sistemas baseados em Unix que permite programar tarefas a serem executadas periodicamente em um tempo específico.

A anotação `@Scheduled` dá suporte a *cron expressions*. Como especificado no *javadoc*, o padrão de

uma *cron expression* é uma lista de seis campos separados por espaço único, representando respectivamente: segundo, minuto, hora, dia, mês, dia da semana. Os nomes dos dias e dos dias da semana podem ser dados com as três primeiras letras dos nomes em inglês. Exemplos:

- "0 0 " = todas as horas de todos os dias.
- "/10 " = a cada dez segundos.
- "0 0 3-6 *" = 3, 4 e 5 horas de cada dia.
- "0 0 8,15 *" = 8:00 AM e 3:00 PM todos os dias.
- "0 0/30 8-10 *" = 8:00, 8:30, 9:00, 9:30, 10:00 e 10:30 todos os dias.
- "0 0 9-17 MON-FRI" = das 9:00 às 17:00 de segunda a sexta.
- "0 0 0 25 12 ?" = todos os dias de Natal à meia-noite

Com o tempo, o uso de *cron expressions* se tornou amplamente adotado e pode ser usado em vários outros contextos.

Voltando à nossa tarefa, vamos usar a *cron expression* para definir que nosso código deve ser executado às 20h de cada dia:

```
@Scheduled(cron = "0 0 20 * * *")
public void execute() {
    List<OpenTopicsByCategory> topics =
        this.topicRepository.findOpenTopicsByCategory();
    this.openTopicByCategoryRepository.saveAll(topics);
}
```

Agora conseguimos o resultado desejado: este código será executado todos os dias às 20h. Note que o Spring facilita bastante a implementação de agendamento com a anotação `@Scheduled`.

11.4 EXERCÍCIO: AGENDANDO TAREFAS COM SPRING

Nossa aplicação deve ser capaz de gerar um relatório com todas as dúvidas que permanecem sem resposta ao fim de cada dia. Para isso, vamos agendar uma tarefa que busca todas as dúvidas abertas em determinada hora do dia e registrar o histórico no banco de dados.

1. No pacote `br.com.alura.forum.task`, crie a classe `RegisterUnansweredTopicsTask` com o método `execute()`, que deve buscar todos os tópicos ainda abertos por categoria:

```
@Component
public class RegisterUnansweredTopicsTask {

    @Autowired
    private TopicRepository topicRepository;

    public void execute() {
        List<OpenTopicByCategory> topics =
            topicRepository.findOpenTopicsByCategory();
    }
}
```

2. Nossa código ainda não compila. Precisamos criar a classe `OpenTopicByCategory` primeiro. No pacote `br.com.alura.forum.model`, crie essa classe, que será uma entidade representando os tópicos que permanecem abertos por categoria:

```
@Entity
public class OpenTopicByCategory {

    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    private Long id;

    private String categoryName;

    private int topicCount;

    private LocalDate date;

    @Deprecated
    public OpenTopicByCategory() {
    }

    public OpenTopicByCategory(String categoryName, Number topicCount,
                               Date instant) {
        this.categoryName = categoryName;
        this.topicCount = topicCount.intValue();
        this.date = instant.toInstant()
            .atZone(ZoneId.systemDefault())
            .toLocalDate();
    }

    // getters necessários omitidos
}
```

As anotações `@Entity`, `@Id` e `@GeneratedValue` são do JPA e pertencem ao pacote `javax.persistence`, e a classe `Date` é importada do pacote `java.util`. Agora, crie o método `findOpenTopicsByCategory()` na classe `TopicRepository`:

```
public interface TopicRepository extends
    Repository<Topic, Long>, JpaSpecificationExecutor<Topic> {

    // outros métodos omitidos

    @Query("select new br.com.alura.forum.model.OpenTopicByCategory(" +
        "t.course.subcategory.category.name as categoryName, " +
        "count(t) as topicCount, " +
        "now() as instant) from Topic t " +
        "where t.status = 'NOT_ANSWERED' " +
        "group by t.course.subcategory.category")
    List<OpenTopicByCategory> findOpenTopicsByCategory();
}
```

3. Depois de buscar todos os tópicos abertos, precisamos salvá-los. No pacote `br.com.alura.forum.repository`, crie a interface `OpenTopicByCategoryRepository` com o método `saveAll()`:

```
public interface OpenTopicByCategoryRepository
```

```

    extends Repository<OpenTopicByCategory, Long> {

    void saveAll(Iterable<OpenTopicByCategory> topics);
}

```

4. De volta à classe `RegisterUnansweredTopicsTask`, adicione a dependência de `OpenTopicByCategoryRepository` e chame o método `saveAll()` para persistir os dados:

```

@Component
public class RegisterUnansweredTopicsTask {

    @Autowired
    private TopicRepository topicRepository;

    @Autowired
    private OpenTopicByCategoryRepository openTopicByCategoryRepository;

    public void execute() {
        List<OpenTopicByCategory> topics =
            topicRepository.findOpenTopicsByCategory();
        this.openTopicByCategoryRepository.saveAll(topics);
    }
}

```

5. Nossa código está pronto. Agora, precisamos que ele seja executado em determinado horário do dia. Vamos **agendar** esta tarefa para rodar todos os dias, às 20h.

- Primeiro, precisamos habilitar o agendamento do Spring com a anotação `@EnableScheduling` na classe `AluraForumApplication`:

```

@EnableScheduling
@EnableAsync
@EnableSpringDataWebSupport
@SpringBootApplication
public class AluraForumApplication {

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

```

- E utilizar a anotação `@Scheduled` passando a *cron expression* que agendará a execução para às 20h de cada dia:

```

@Component
public class RegisterUnansweredTopicsTask {

    // código anterior omitido

    @Scheduled(cron = "0 0 20 * *")
    public void execute() {
        List<OpenTopicByCategory> topics =
            topicRepository.findOpenTopicsByCategory();
        this.openTopicByCategoryRepository.saveAll(topics);
    }
}

```

PS: Para testar durante o desenvolvimento você pode alterar a cron expression para executar a

tarefa a cada 10 segundos (cron = “/10 * * * *”). Lembre apenas de retornar à cron original após algumas inserções.*

6. Rode a API e veja como os dados são registrados automaticamente.

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

11.5 RENDERIZANDO RELATÓRIO DE TÓPICOS ABERTOS COM THYMELEAF

Nossa próxima tarefa é gerar um relatório das dúvidas abertas do fórum com *Thymeleaf*. A equipe de *front end* já desenvolveu o template `report.html`, que será usado na aplicação e disponibilizado pelo endpoint `/admin/reports/open-topics-by-category`. A Alura deseja acompanhar como anda a taxa de tópicos não respondidos por dia e no mês corrente para criar estratégias de contingência e seu time diminuir essa taxa. Devemos, então, gerar um relatório com a lista de todos os tópicos abertos no mês corrente.

O template `report.html` necessita da lista de tópicos abertos do mês corrente para funcionar. Precisamos construir essa `query` em nosso `OpenTopicByCategoryRepository`:

```
public interface OpenTopicByCategoryRepository
    extends Repository<OpenTopicByCategory, Long> {
    // código anterior omitido
    @Query("select t from OpenTopicByCategory t "
        + "where year(t.date) = year(current_date) "
        + "and month(t.date) = month(current_date)")
    List<OpenTopicsByCategory> findAllByCurrentMonth();
}
```

A funções `year()` e `month()` são funções internas do SQL que ajudam a construir essa *query*. Nossa *query* vai buscar por todos `OpenTopicByCategory` do mês e ano correntes.

Agora, para responder à requisição do *endpoint*, será necessário construir um método mapeado dentro de um *controller*. Vamos criar a classe `ReportsController` para buscar e disponibilizar a lista de tópicos abertos. Usaremos a classe `Model` do Spring para disponibilizar a lista para a *view*, como visto anteriormente:

```
@Controller
@RequestMapping("/admin/reports")
public class ReportsController {

    @GetMapping("/open-topics-by-category")
    public String showOpenTopicsByCategoryReport(Model model){
        List<OpenTopicByCategory> openTopics =
            this.openTopicByCategoryRepository.findAllByCurrentMonth();
        model.addAttribute("openTopics", openTopics);
        return "report";
    }
}
```

Note que anotamos a classe com `@Controller` já que seu retorno é uma *view*, no caso, a nossa `report.html`. Como o retorno do método é uma `String`, o Spring vai procurar por um arquivo `report.html` dentro da pasta `src/main/resources/templates` - portanto, nosso arquivo `report.html` será colocado lá.

Para funcionar, ainda precisamos recuperar a lista dentro do *template*. Atualmente, o código está como abaixo:

```
<!-- código omitido -->
```

```
<tbody>
    <tr>
        <td>1</td>
        <td>Programação</td>
        <td>11/10/2018</td>
        <td>21</td>
    </tr>
</tbody>
```

```
<!-- código omitido -->
```

Ou seja, está mostrando, em uma estrutura de tabela, apenas um resultado fixo e não a lista esperada. Queremos que a lista atualizada, do mês corrente, seja mostrada. Precisaremos iterar pela lista. Como vimos, o *Thymeleaf* dá suporte para *for loops* por meio do atributo `th:each`:

```
<tbody>
    <tr th:each="item : ${openTopics}">

    </tr>
</tbody>
```

Podemos ler esse código da seguinte maneira: "Para cada *item* dentro de *openTopics*, faça:". É como um *forEach* do Java. Agora cada coluna deverá mostrar seu conteúdo específico:

```
<!-- código omitido -->

<tbody>
    <tr th:each="item : ${openTopics}">
        <td th:text="${item.id}">1</td>
        <td th:text="${item.categoryName}">Programação</td>
        <td th:text="#${temporals.format(item.date, 'dd/MM/yyyy - EEEE',
            'pt')}">11/10/2018</td>
        <td th:text="${item.topicCount}">21</td>
    </tr>
</tbody>

<!-- código omitido -->
```

A novidade aqui é o `#temporals.format()` - função do *Thymeleaf* para formatar datas. No caso, o padrão `dd/MM/yyyy - EEEE`, `pt` vai renderizar a data no formato " 11/10/2018 - Quinta-feira ". Para saber mais sobre formatação de data usando *Thymeleaf*, acesse diretamente a parte da [documentação](#) sobre datas.

Esse código ainda não funciona pois precisamos liberar o acesso para `/admin/reports/**` na configuração de segurança do método `configure(HttpSecurity http)` da classe `SecurityConfiguration` :

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers(HttpMethod.GET, "/api/topics/**").permitAll()
        .antMatchers("/api/auth/**").permitAll()
        .antMatchers("/admin/reports/**").permitAll()
        .anyRequest().authenticated()
    .and()
    // código omitido
}
```

Além disso, será preciso liberar acesso para os arquivos estáticos utilizados no template, como `css`, `javascript` e arquivos de imagens, dentro do método `configure(WebSecurity webSecurity)` da mesma classe de configuração:

```
@Override
public void configure(WebSecurity web) throws Exception {
    web.ignoring().antMatchers("/**.html", "/v2/api-docs", "/webjars/**",
        "/configuration/**", "/swagger-resources/**", "/css/**",
        "**.ico", "/js/**");
}
```

Os arquivos estáticos da aplicação devem ficar em uma pasta chamada `static` dentro de `scr/main/resources`. Por padrão, o Spring Boot busca estes arquivos neste local específico.

Agora, ao acessar `http://localhost:8080/admin/reports/open-topics-by-category` , o template do relatório será renderizado no navegador.

11.6 EXERCÍCIO: RENDERIZANDO RELATÓRIO DE TÓPICOS ABERTOS COM THYMELEAF

Já conseguimos agendar o registro das dúvidas abertas, agora vamos gerar um relatório para exibir essa informação usando *Thymeleaf*.

1. Na pasta `/caelum/cursos/27/relatorio`, copie o arquivo `report.html` para a pasta `src/main/resources/templates/`.
2. Nossa `report.html` depende de arquivos estáticos, como folhas de estilo (.css) e scripts (.js) para funcionar. Copie as pastas `css`, `js` e `img` para a pasta `src/main/resources/static`.

O acesso a esses arquivos precisa ser público, então inclua essa configuração na classe `SecurityConfiguration`:

```
@Override  
public void configure(WebSecurity web) throws Exception {  
    web.ignoring().antMatchers("/**.html", "/v2/api-docs", "/webjars/**",  
        "/configuration/**", "/swagger-resources/**", "/css/**",  
        "/**.ico", "/js/**");  
}
```

3. No pacote de `controllers`, crie a classe `ReportsController` contendo o método `showOpenTopicsByCategoryReport()`. Este método deve buscar as informações sobre as dúvidas abertas ao fim de cada dia do mês corrente, e disponibilizar a lista que o *Thymeleaf* irá acessar para gerar a resposta ao cliente.

```
@Controller  
@RequestMapping("/admin/reports")  
public class ReportsController {  
  
    @Autowired  
    private OpenTopicByCategoryRepository openTopicByCategoryRepository;  
  
    @GetMapping("/open-topics-by-category")  
    public String showOpenTopicsByCategoryReport(Model model) {  
  
        List<OpenTopicByCategory> openTopics =  
            openTopicByCategoryRepository.findAllByCurrentMonth();  
        model.addAttribute("openTopics", openTopics);  
  
        return "report";  
    }  
}
```

Note que o retorno do método é a `String "report"`, sinalizando ao Spring qual página deve ser renderizada ao final da requisição: `report.html`. A classe `Model`, que deve ser importada do pacote `org.springframework.ui`, representa o modelo a ser disponibilizado para nossa página e que podemos preencher com a lista de dúvidas abertas.

4. O código ainda não compila. Crie o método `findAllByCurrentMonth()` na classe

`OpenTopicByCategoryRepository`, que busca todas as dúvidas abertas no mês corrente:

```
public interface OpenTopicByCategoryRepository
    extends Repository<OpenTopicByCategory, Long> {

    void saveAll(Iterable<OpenTopicByCategory> topics);

    @Query("select t from OpenTopicByCategory t " +
        "where year(t.date) = year(current_date) " +
        "and month(t.date) = month(current_date)")
    List<OpenTopicByCategory> findAllByCurrentMonth();
}
```

5. Com a lista disponível na *View*, abra o arquivo `report.html` e altere a implementação para que o *Thymeleaf* gere uma linha para cada elemento usando `th:each`.

```
<!-- código omitido -->

<tbody>
    <tr th:each="item : ${openTopics}">
        <td th:text="${item.id}">1</td>
        <td th:text="${item.categoryName}">Programação</td>
        <td th:text="#temporals.format(item.date, 'dd/MM/yyyy - EEEE',
            'pt')}>11/10/2018</td>
        <td th:text="${item.topicCount}">21</td>
    </tr>
</tbody>

<!-- código omitido -->
```

6. Rode a aplicação e acesse o endereço `http://localhost:8080/admin/reports/open-topics-by-category` para acessar o relatório.

Como o acesso ao endereço é bloqueado pelo Spring Security, adicione momentaneamente a liberação nas configurações de segurança:

```
//código omitido
.antMatchers("/admin/reports/**").permitAll()
//código omitido
```

Open Topics Report				
Show 10 entries		Search:		
ID	Categoria	Dia de Referência	Número de Tópicos	
1	Programação	11/10/2018 - Quinta-feira	6	
2	Design	11/10/2018 - Quinta-feira	6	
3	Mobile	11/10/2018 - Quinta-feira	3	
4	Front-end	11/10/2018 - Quinta-feira	2	
5	Infraestrutura	11/10/2018 - Quinta-feira	1	
6	Business	11/10/2018 - Quinta-feira	2	
7	Programação	11/10/2018 - Quinta-feira	6	
8	Design	11/10/2018 - Quinta-feira	6	
9	Mobile	11/10/2018 - Quinta-feira	3	
10	Front-end	11/10/2018 - Quinta-feira	2	

11.7 MÚLTIPLOS PERFIS DE SEGURANÇA

A aplicação já é capaz de gerar o relatório de dúvidas abertas mas ele está público e apenas usuários com permissão de administradores devem ter acesso a este recurso. Além do mais, o relatório é acessado diretamente do navegador e não está atrelado à aplicação cliente do Fórum.

Antes de acessar, o administrador precisa se autenticar e, portanto, precisa de outra tela de login que não aquela da aplicação cliente. Precisamos de uma nova estratégia de segurança para este contexto. Em termos de código, precisamos de uma nova configuração de segurança específica para usuários administradores acessarem o relatório.

Vamos considerar que, em nossa aplicação, recursos que não tem vínculo direto com a aplicação cliente terão os *endpoints* seguindo o padrão `/admin/**` e aqueles atrelados à aplicação cliente seguirão o padrão `/api/**` - também denominamos cada um deles de *entry points* (pontos de entrada).

Criaremos, então, a classe `AdminSecurityConfiguration` para configurar as regras de administradores:

```
@Configuration
@EnableWebSecurity
public class AdminSecurityConfiguration extends WebSecurityConfigurerAdapter{

    @Autowired
    private UserService usersService;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest().hasRole("ADMIN")
            .and()
```

```

        .httpBasic();
    }

@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(this.userService)
        .passwordEncoder(new BCryptPasswordEncoder());
}
}

```

Até aqui, nada de novo do que já vimos no capítulo de segurança. Os usuários administradores estão persistidos junto com os demais usuários, na mesma tabela `user` do banco - portanto, usaremos a mesma implementação de `UserDetailsService` para carregar o usuário (nossa `UserService`) e o Spring vai fazer a autenticação seguindo o mesmo fluxo anterior do Spring Security.

A diferença é que usaremos a autenticação via `HTTP Basic` para facilitar o login. Uma autenticação do tipo `HTTP Basic` é a maneira mais simples de forçar um controle de acesso a recursos seguros na web por não exigir nem mesmo uma página de login. Por padrão, ao acessar um recurso seguro utilizando `HTTP Basic`, o navegador vai mostrar uma janela *pop-up* pedindo os dados de autenticação.

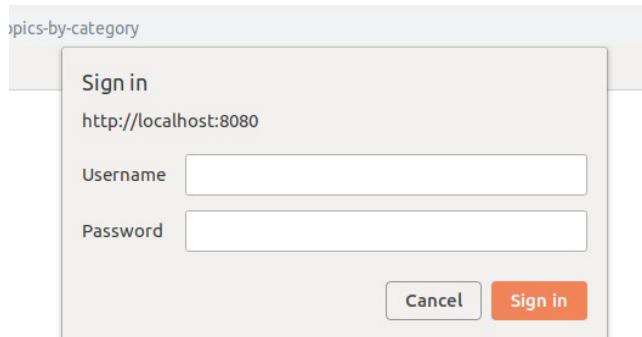


Figura 11.3: http-basic-login

Além disso, o método `hasRole("ADMIN")` libera acesso apenas para usuários com `ROLE_ADMIN`.

Para especificar que esta configuração será apenas considerada para o *entry point* `/admin`, precisamos, antes de chamar o `authorizeRequests()` do `HttpSecurity`, chamar o método `antMatcher()` com o *match* correto:

```

@Configuration
@EnableWebSecurity
public class AdminSecurityConfiguration extends WebSecurityConfigurerAdapter{

    // atributos omitidos

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/admin/**")
            .authorizeRequests()
                .anyRequest().hasRole("ADMIN")
            .and()
    }
}

```

```

        .httpBasic();
    }

    // outros método omitidos
}

```

Dessa maneira, esta configuração funcionará apenas para os *endpoints* iniciados com `/admin/`. A mesma coisa faremos para nosso outro contexto de segurança, só que com o *entry point* `/api/`:

```

@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    // atributos omitidos

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/api/**")
            .authorizeRequests()
                .antMatchers(HttpMethod.GET, "/api/topics/**").permitAll()
                .antMatchers("/api/auth/**").permitAll()
                .anyRequest().authenticated()
            .and()
        // código omitido
    }

    // outros método omitidos
}

```

Depois de pronta a configuração de cada contexto, caímos em outro problema. Qual será a configuração que o Spring considera quando tentamos fazer uma autenticação para `/admin/reports/open-topics-by-category`? Caso ele considere a `SecurityConfiguration`, vai liberar para qualquer usuário, já que nossa `SecurityConfiguration` barra acesso apenas para o *entry point* `/api/`.

Queremos que uma configuração tenha precedência sobre a outra. Como nenhum usuário com permissão diferente de `ADMIN` conseguirá acessar o *entry point* `/admin/`, já que deixamos isso claro na parte `.anyRequest().hasRole("ADMIN")`, precisamos falar ao Spring para ele considerar primeiro o contexto de segurança de administradores, ou seja, a `AdminSecurityConfiguration`. Caso a autorização falhe neste contexto, ele tenta pelo segundo contexto. Dessa maneira, garantimos que tudo funcione como o esperado.

O Spring possui a anotação `@Order` para este tipo de caso, em que um componente tem ordem de prioridade sobre outro. Como nossa `AdminSecurityConfiguration` deve ser considerada primeiro, passamos o parâmetro `1`, pois valores menores tem maior prioridade:

```

@Order(1)
@Configuration
@EnableWebSecurity
public class AdminSecurityConfiguration extends WebSecurityConfigurerAdapter{

    // código interior omitido
}

```

E anotamos a classe `SecurityConfiguration` com `@Order(2)` seguindo o mesmo raciocínio, dado que tem prioridade menor do que `AdminSecurityConfiguration`:

```
@Order(2)
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
    // código interior omitido
}
```

Agora tudo funciona como o esperado.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

11.8 EXERCÍCIO: TRABALHANDO COM MÚLTIPLOS PERFIS DE SEGURANÇA

A aplicação já é capaz de gerar o relatório de dúvidas abertas mas ele está público. Vamos criar um novo contexto de segurança para que apenas administradores do sistema tenham acesso ao relatório.

1. No pacote `br.com.alura.forum.security.configuration` , crie a classe `AdminSecurityConfiguration`. Recursos de administrador serão acessados a partir da URI `/admin/**` e o login será feito utilizando o `httpBasic` :

```
@Configuration
@EnableWebSecurity
public class AdminSecurityConfiguration extends WebSecurityConfigurerAdapter {

    @Autowired
    private UsersService usersService;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/admin/**")
            .authorizeRequests().anyRequest().hasRole("ADMIN")
            .and()
```

```

        .httpBasic();
    }

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.userDetailsService(this.userService)
            .passwordEncoder(new BCryptPasswordEncoder());
    }
}

```

2. Na classe `SecurityConfiguration`, modifique o método `configure()` para que seja adicionada a URI base `/api/**` direcionando a classe para o *entry point* de segurança relativo aos *requests* de usuários do fórum.

```

@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    // código anterior omitido

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.antMatcher("/api/**") // adicionado o antMatcher base
            .authorizeRequests()
                .antMatchers(HttpMethod.GET, "/api/topics/**").permitAll()
                .antMatchers("/api/auth/**").permitAll()
                .anyRequest().authenticated()
                // ...
        // código posterior omitido
    }

    // código posterior omitido
}

```

3. Com os dois *entry points* de segurança configurados, precisamos sinalizar ao Spring Security qual a precedência desses pontos de entrada. O *entry point* admin terá maior precedência, sendo assim, anote a classe `AdminSecurityConfiguration` com a anotação `@Order(1)`:

```

@Configuration
@Order(1)
@EnableWebSecurity
public class AdminSecurityConfiguration extends WebSecurityConfigurerAdapter

    // código omitido
}

```

- E anote a classe `SecurityConfiguration` com `Order(2)`:

```

@Configuration
@Order(2)
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

    // código omitido
}

```

4. Rode a aplicação e acesse o endereço `http://localhost:8080/admin/reports/open-topics-by-`

category para acessar o relatório. Faça o login com rafael.rollo@caelum.com.br e senha rafael e veja que o novo contexto de segurança autentica o usuário admin e libera acesso aos recursos.

*Lembre-se de remover o .antMatchers("/admin/reports/**").permitAll() do contexto de segurança dos usuários do fórum.*

TESTANDO A API COM SPRING BOOT TEST

12.1 TESTANDO NOSSA APLICAÇÃO

Um detalhe muito importante, mas que não foi tratado até este momento, é a questão de testabilidade da nossa aplicação. Por mais interessante que seja a experiência de interagir com nossas lógicas através de ferramentas como o Swagger UI, não queremos ficar rodando tudo manualmente para saber se as coisas estão funcionando ao implementar novas funcionalidades.

Como dito no início deste curso, o Spring Boot, por padrão, já acrescenta a dependência do módulo de testes - já que espera que você siga as boas práticas e crie testes para sua aplicação. Essa dependência já habilita suporte ao uso de banco de dados em memória, como o H2 ou HSQLDB, de forma automática, bastando apenas contar com a presença da dependência de um deles no projeto - usaremos o H2 .

Por padrão, projetos que utilizam Maven criam o *source folder* `src/test/java` - diretório reservado para a criação de testes. O Spring Boot, por padrão, já vem com um teste criado para a classe principal da aplicação, no nosso caso a `AluraForumApplication` , chamada de `AluraForumApplicationTests` :

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class AluraForumApplicationTests {

    @Test
    public void contextLoads() {
    }
}
```

Vamos entender esse código de testes. Na anotação `@RunWith(SpringRunner.class)` , a classe `SpringRunner` é um *alias* para a classe `SpringJUnit4ClassRunner` , que especifica que o Spring vai utilizar o JUnit 4 para rodar os testes - o `JUnit` é um *framework* que dá suporte à criação de testes automatizados e um dos mais utilizados na linguagem Java. Já a anotação `@SpringBootTest` provê algumas funcionalidades extras para rodar classes de testes no Spring Boot. Essa anotação diz para o Spring Boot procurar a configuração principal (aquela com `@SpringBootApplication`) e usá-la para iniciar o contexto de aplicação do Spring. Você pode rodar este teste direto no Eclipse (clique com botão

direito no código, vá em Run As -> JUnit Test) ou na linha de comando (mvn test) e ele deve passar. A anotação @Test do JUnit indica que este é um método de teste e deve ser executado como tal.

Para convencer você que este contexto está criando algum componente da aplicação, por exemplo um controller ou service, vamos injetar um componente e adicionar a assertão assertThat abaixo:

```
// outros imports

import static org.assertj.core.api.Assertions.assertThat;

@RunWith(SpringRunner.class)
@SpringBootTest
public class AluraForumApplicationTests {

    @Autowired
    private UsersService usersService;

    @Test
    public void contextLoads() {
        assertThat(usersService).isNotNull();
        assertThat(usersService).isInstanceOf(UserDetailsService.class);
    }
}
```

assertThat , traduzido ao pé da letra para o português, quer dizer afirme que . Este método será importado pelo import estático import static org.assertj.core.api.Assertions.assertThat . O teste vai verificar se o que estamos afirmando é verdade - caso alguma afirmação feita seja falsa, o teste falha. Na primeira linha, está sendo verificado se a instância de UsersService injetada é mesmo não nula. A segunda linha verifica se essa mesma instância é um UserDetailsService - como nossa classe UsersService implementa a interface UserDetailsService , este teste deve passar:

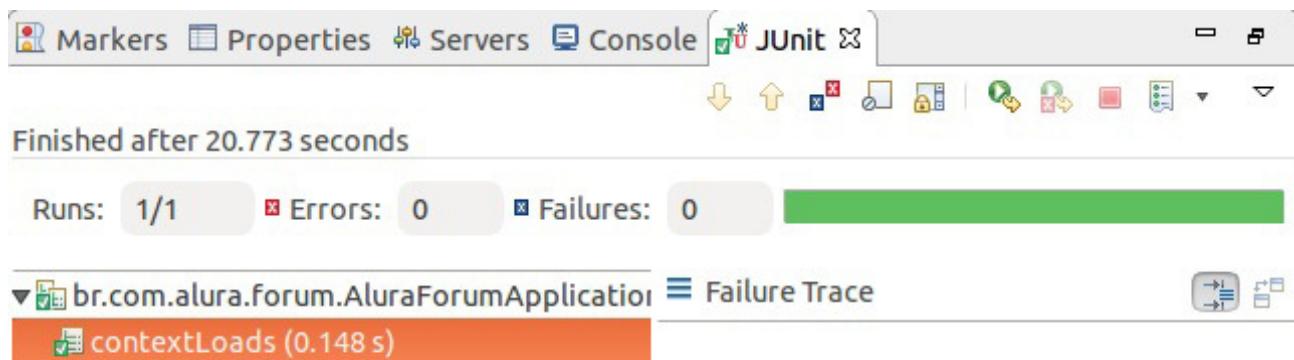


Figura 12.1: junit test

A imagem acima é do console do JUnit , que é aberto quando um teste é executado. A tarja verde ao lado direito significa que o teste passou.

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

12.2 TESTANDO O REPOSITORY

Primeiro, vamos testar nosso `TopicRepository` isoladamente. Criaremos um teste para saber se o método `save()` de `TopicRepository` está funcionando como o esperado. Criaremos nosso primeiro teste no pacote `br.com.alura.forum.repository` no diretório `src/test/java` e dentro deste pacote criaremos a classe `TopicRepositoryTests`:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public TopicRepositoryTests {

    @Test
    public void shouldSaveATopic() {
    }
}
```

Aqui, precisaremos avisar o Spring que rodaremos essa classe de testes com o `Junit 4`, portanto, usaremos novamente a anotação `@RunWith(SpringRunner.class)`. Também devemos usar a anotação `@SpringBootTest` para iniciar o contexto de aplicação do Spring e injetarmos as classes necessárias para fazer o teste.

Criamos o método `shouldSaveATopic()` anotado com `@Test`, que avisa o `JUnit` que este será um método que deve ser rodado como um teste. Vamos, então, criar um novo tópico e salvá-lo no banco de dados:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public TopicRepositoryTests {

    @Autowired
    private TopicRepository topicRepository;

    @Test
}
```

```

    public void shouldSaveATopic() {
        Topic topic = new Topic("Descrição do tópico", "Conteúdo do tópico", null, null);
        Topic persistedTopic = this.topicRepository.save(topic);
    }
}

```

Agora precisamos garantir que esse tópico foi salvo. Perceba que não podemos utilizar o `topicRepository` para buscar este tópico que acabamos de criar - visto que nem sequer testamos o método `findAll()` ou `findById()`. Para isso, precisamos de outro objeto, de uma camada abaixo, que também acessa o banco. Precisamos de uma instância do `EntityManager` (da especificação do JPA), que está associado ao nosso contexto de persistência e é utilizado pelo `repository` por baixo dos panos. O Spring Boot dispõe de uma classe chamada de `TestEntityManager`, que é uma alternativa ao `EntityManager` para testes e provê um conjunto de métodos úteis para essa tarefa. Para que o Spring Boot habilite sua injeção, precisamos utilizar a anotação `@DataJpaTest`:

```

@RunWith(SpringRunner.class)
@DataJpaTest
public TopicRepositoryTests {

    @Autowired
    private TopicRepository topicRepository;

    @Autowired
    private TestEntityManager testEntityManager;

    // método de teste omitido
}

```

Não será mais necessário o uso da anotação `@SpringBootTest` já que a `@DataJpaTest` encapsula funcionalidades dessa anotação e outras funcionalidades para testes da camada de persistência da aplicação, como `@Transactional` para o controle de transações. Agora podemos buscar o tópico adicionado no banco e verificar se ele existe e se a descrição é a mesma que adicionamos:

```

@RunWith(SpringRunner.class)
@DataJpaTest
public TopicRepositoryTests {

    @Autowired
    private TopicRepository topicRepository;

    @Autowired
    private TestEntityManager testEntityManager;

    @Test
    public void shouldSaveATopic() {
        Topic topic = new Topic("Descrição do tópico", "Conteúdo do tópico", null, null);

        Topic persistedTopic = this.topicRepository.save(topic);
        Topic topicoEncontrado = this.testEntityManager.find(Topic.class, persistedTopic.getId());

        assertThat(topicoEncontrado).isNotNull();
        assertThat(topicoEncontrado.getShortDescription()).isEqualTo(persistedTopic.getShortDescription());
    }
}

```

Mas, ao rodar o teste, recebemos a exceção abaixo:

```
Caused by: java.lang.IllegalStateException: Failed to replace DataSource with an embedded database for tests. If you  
at org.springframework.util.Assert.state(Assert.java:73) ~[spring-core-5.1.4.RELEASE.jar:5.1.4.RELEASE]  
at org.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration$EmbeddedDataSourceFactory  
at org.springframework.boot.test.autoconfigure.jdbc.TestDatabaseAutoConfiguration$EmbeddedDataSourceFactory  
at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.invokeInitMethods(AbstractAutowireCapableBeanFactory.java:348)  
at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.initializeBean(AbstractAutowireCapableBeanFactory.java:318)  
... 64 common frames omitted
```

Figura 12.2: H2 Exception

A exceção diz que falhou ao substituir o `DataSource` com outra base de dados. Isso acontece porque a anotação `@DataJpaTest`, por padrão, não utiliza o banco de dados usado em produção para os testes e espera que exista um banco de dados em memória para ser usado nesses casos. Um banco em memória não persiste os dados em disco e, por este motivo, são amplamente utilizados para testes. Um dos mais utilizados é o H2, e o Spring Boot dá suporte e fornece uma pré configuração para ele. Basta adicionarmos a dependência abaixo no nosso projeto:

```
<dependencies>  
  
    <!-- dependência do greenmail -->  
  
    <dependency>  
        <groupId>com.h2database</groupId>  
        <artifactId>h2</artifactId>  
        <scope>test</scope>  
    </dependency>  
</dependencies>
```

Agora, ao rodar o teste novamente, vemos que ele passa:

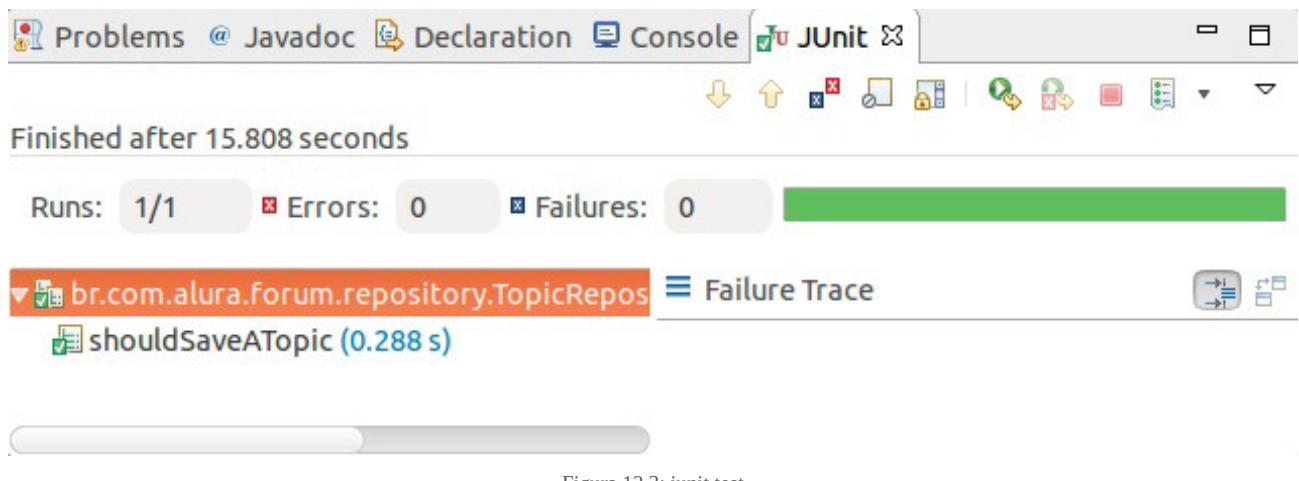


Figura 12.3: junit test

Atente-se em como não precisamos de nenhum esforço para configurar o acesso a dados com o H2. Caso rode uma segunda vez o teste, verá que tudo continua funcionando, dado que, em um teste de repositório do Spring Data JPA, todos os métodos são `@Transactional` e, por padrão, ao final da execução aplicam *roll back* nas alterações no estado da base, facilitando a execução de testes adicionais.

12.3 TESTANDO NOSSAS QUERIES

Testamos um método do Spring Data JPA, ou seja, testamos um método cuja *query* foi construída pelo próprio framework que já foi exaustivamente testado. Portanto, não é de grande valia para nós testar *queries* geradas pelo próprio framework (como é o caso do método `save()`). Vamos adicionar testes que verifiquem a implementação de nossas *queries*, como no caso do método `findOpenTopicsByCategory()` de `TopicRepository`:

```
public interface TopicRepository extends Repository<Topic, Long>, JpaSpecificationExecutor<Topic> {  
  
    @Query("select new br.com.alura.forum.model.OpenTopicByCategory(" +  
        "t.course.subcategory.category.name as categoryName, " +  
        "count(t) as topicCount, " +  
        "now() as instant) from Topic t " +  
        "where t.status = 'NOT_ANSWERED' " +  
        "group by t.course.subcategory.category")  
    List<OpenTopicByCategory> findOpenTopicsByCategory();  
  
}
```

Dentro de `TopicRepositoryTests`, vamos criar o método `shouldReturnOpenTopicsByCategory()` para testar essa *query*:

```
@RunWith(SpringRunner.class)  
@DataJpaTest  
public class TopicRepositoryTests {  
  
    @Test  
    public void shouldReturnOpenTopicsByCategory() {  
    }  
}
```

Primeiro, vamos verificar se o resultado desta busca não é vazio. Chamamos o método `findOpenTopicsByCategory` de nosso `TopicRepository` e, em seguida, fazemos a verificação `isNotEmpty()` do `assertThat`:

```
@RunWith(SpringRunner.class)  
@DataJpaTest  
public class TopicRepositoryTests {  
  
    // outros atributos e métodos  
  
    @Test  
    public void shouldReturnOpenTopicsByCategory() {  
        List<OpenTopicByCategory> openTopics =  
            this.topicRepository.findOpenTopicsByCategory();  
  
        assertThat(openTopics).isNotEmpty();  
    }  
}
```

Ao executar este método, ele falha:

Runs: 2/2 Errors: 0 Failures: 1

The screenshot shows a test runner interface with a summary at the top: "Runs: 2/2", "Errors: 0", and "Failures: 1". Below this, a list of tests is shown, with one test failing. The failing test is "shouldReturnOpenTopicsByCategory (0.0)" from the class "br.com.alura.forum.repository.TopicRepos". The failure trace is displayed on the right, showing the exception: "java.lang.AssertionError: Expecting actual not to be empty" and the stack trace: "at br.com.alura.forum.repository.TopicRepos...".

Figura 12.4: teste falha

A faixa vermelha à direita indica que o teste falhou. Ou seja, a nossa *query* não está trazendo nada do banco. Será que isso faz sentido?

Neste momento, pare e pense um pouco em tudo que fizemos até aqui. Adicionamos a dependência do H2 *database* para ser usado em testes e aprendemos que a anotação `@DataJpaTest` adiciona `@Transactional` em todos os métodos, ou seja, aplicam *roll back* nas alterações do estado da base ao final da execução. Como nossa base está vazia, faz todo o sentido nosso teste falhar.

Precisamos, então, garantir que exista alguns tópicos persistidos na base de dados de testes para fazer essa verificação.

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class TopicRepositoryTests {

    // outros atributos e métodos

    @Test
    public void shouldReturnOpenTopicsByCategory() {
        Category programacao = this.testEntityManager
            .persist(new Category("Programação"));
        Category front = this.testEntityManager
            .persist(new Category("Front-end"));

        Category javaWeb = this.testEntityManager
            .persist(new Category("Java Web", programacao));
        Category javaScript = this.testEntityManager
            .persist(new Category("JavaScript", front));

        Course fj27 = this.testEntityManager
            .persist(new Course("Spring Framework", javaWeb));
        Course fj21 = this.testEntityManager
            .persist(new Course("Servlet Api e MVC", javaWeb));
        Course js46 = this.testEntityManager
            .persist(new Course("React", javaScript));

        Topic springTopic = new Topic("Tópico Spring", "Conteúdo do tópico", null, fj27);
        Topic servletTopic = new Topic("Tópico Servlet", "Conteúdo do tópico", null, fj21);
        Topic reactTopic = new Topic("Tópico React", "Conteúdo do tópico", null, js46);
    }
}
```

```

        this.testEntityManager.persist(springTopic);
        this.testEntityManager.persist(servletTopic);
        this.testEntityManager.persist(reactTopic);

        List<OpenTopicByCategory> openTopics =
            this.topicRepository.findOpenTopicsByCategory();

        assertThat(openTopics).isNotEmpty();
        assertThat(openTopics).hasSize(2);
    }
}

```

Agora, ao rodar os testes, tudo passa:

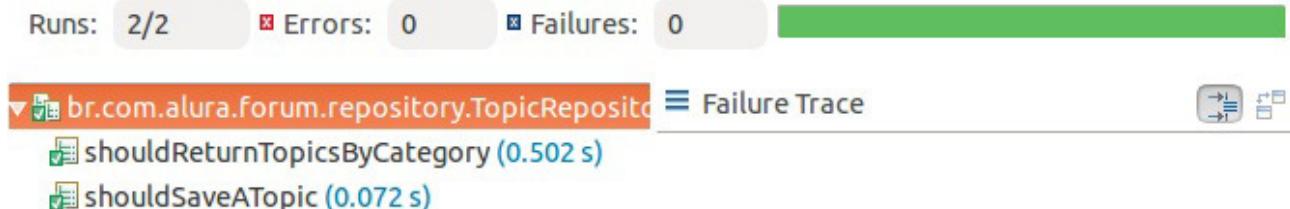


Figura 12.5: teste passa

Antes de seguirmos com novas verificações, vamos melhorar nosso código. Note que podemos reutilizar estes tópicos persistidos em outros testes de nosso *repository* e não é responsabilidade de nosso método `shouldReturnOpenTopicsByCategory()` fazer esta tarefa de popular o banco. É muito comum, quando escrevemos testes, isolar esse *setup* inicial. No pacote `br.com.alura.forum.repository` dentro de `src/test/java`, vamos criar a classe `TopicRepositoryTestsSetup` para isolar essa configuração inicial do nosso teste em um método chamado `openTopicsByCategorySetup()`:

```

public class TopicRepositoryTestsSetup {

    private TestEntityManager testEntityManager;

    public TopicRepositoryTestsSetup(TestEntityManager testEntityManager) {
        this.testEntityManager = testEntityManager;
    }

    public void openTopicsByCategorySetup() {
        Category programacao = this.testEntityManager
            .persist(new Category("Programação"));
        Category front = this.testEntityManager
            .persist(new Category("Front-end"));

        Category javaWeb = this.testEntityManager
            .persist(new Category("Java Web", programacao));
        Category javaScript = this.testEntityManager
            .persist(new Category("JavaScript", front));

        Course fj27 = this.testEntityManager
            .persist(new Course("Spring Framework", javaWeb));
        Course fj21 = this.testEntityManager
            .persist(new Course("Servlet Api e MVC", javaWeb));
    }
}

```

```

        Course js46 = this.testEntityManager
            .persist(new Course("React", javaScript));

        Topic springTopic = new Topic("Tópico Spring", "Conteúdo do tópico", null, fJ27);
        Topic servletTopic = new Topic("Tópico Servlet", "Conteúdo do tópico", null, fJ21);
        Topic reactTopic = new Topic("Tópico React", "Conteúdo do tópico", null, js46);

        this.testEntityManager.persist(springTopic);
        this.testEntityManager.persist(servletTopic);
        this.testEntityManager.persist(reactTopic);
    }
}

```

Agora, chamamos este *setup* dentro do nosso método de teste:

```

@RunWith(SpringRunner.class)
@DataJpaTest
public class TopicRepositoryTests {

    // outros atributos e métodos

    @Test
    public void shouldReturnOpenTopicsByCategory() {
        TopicRepositoryTestsSetup setup = new TopicRepositoryTestsSetup(this.testEntityManager);
        setup.openTopicsByCategorySetup();

        List<OpenTopicByCategory> openTopics = this.topicRepository.findOpenTopicsByCategory();
        assertThat(openTopics).isNotEmpty();
        assertThat(openTopics).hasSize(2);
    }
}

```

Veja que nosso método ficou mais limpo sem todo o *setup* inicial dentro dele e possibilita focarmos apenas nos testes. Vamos continuar com mais alguns testes verificando se o primeiro resultado buscado é um tópico com categoria igual a `Programação` e se o total de tópicos desta categoria é 2 (já que em nosso *setup* adicionamos dois tópicos da categoria `Programação`).

```

@RunWith(SpringRunner.class)
@DataJpaTest
public class TopicRepositoryTests {

    // outros atributos e métodos

    @Test
    public void shouldReturnOpenTopicsByCategory() {
        TopicRepositoryTestsSetup setup = new TopicRepositoryTestsSetup(this.testEntityManager);
        setup.openTopicsByCategorySetup();

        List<OpenTopicByCategory> openTopics = this.topicRepository.findOpenTopicsByCategory();
        assertThat(openTopics).isNotEmpty();
        assertThat(openTopics).hasSize(2);

        assertThat(openTopics.get(0).getCategoryName()).isEqualTo("Programação");
        assertThat(openTopics.get(0).getTopicCount()).isEqualTo(2);
    }
}

```

Rode novamente e veja que o teste passa. Faça o mesmo com a categoria `Front-end`:

```
@RunWith(SpringRunner.class)
```

```

@DataJpaTest
public class TopicRepositoryTests {

    // outros atributos e métodos

    @Test
    public void shouldReturnOpenTopicsByCategory() {
        TopicRepositoryTestsSetup setup = new TopicRepositoryTestsSetup(this.testEntityManager);
        setup.openTopicsByCategorySetup();

        List<OpenTopicByCategory> openTopics = this.topicRepository.findOpenTopicsByCategory();
        assertThat(openTopics).isNotEmpty();
        assertThat(openTopics).hasSize(2);

        assertThat(openTopics.get(0).getCategoryName()).isEqualTo("Programação");
        assertThat(openTopics.get(0).getTopicCount()).isEqualTo(2);

        assertThat(openTopics.get(1).getCategoryName()).isEqualTo("Front-end");
        assertThat(openTopics.get(1).getTopicCount()).isEqualTo(1);
    }
}

```

Você pode mudar o *setup* acrescentando novos tópicos e depois acrescentar mais verificações para garantir o funcionamento da sua *query*.

Para que seja possível testar a camada de persistência em um ambiente mais próximo do de produção, iremos configurar nossos testes para rodar também com o MySQL. Adicionamos a anotação

`@AutoConfigureTestDatabase(replace= AutoConfigureTestDatabase.Replace.NONE)` sobre a classe de testes, para que o Spring Test não substitua as configurações do DataSource padrão pela integração automática com o H2.

```

@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace= AutoConfigureTestDatabase.Replace.NONE)
public class TopicRepositoryTests {

    // código interno omitido
}

```

Como rodaremos os testes no MySQL, não queremos que o Spring utilize a mesma base de dados utilizada em produção. Vamos criar uma base de dados específica para testes chamada de `fj27_spring_test`. Para que isso seja possível, criaremos uma configuração específica para os testes. Adicionaremos o arquivo `application-test.properties` com o seguinte conteúdo:

```

# data source
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/fj27_spring_test?createDatabaseIfNotExist=true&useSSL=false
spring.datasource.username=root
spring.datasource.password=caelum

# jpa properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL55Dialect
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true

```

Por fim, avisaremos ao Spring para considerar este *profile* para testes com a anotação `@ActiveProfiles` :

```
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace= AutoConfigureTestDatabase.Replace.NONE)
@ActiveProfiles("test")
public class TopicRepositoryTests {

    // código interno omitido
}
```

Ao rodar novamente a aplicação, vemos que tudo funciona - com a diferença que agora estamos testando no MySQL, um cenário muito mais próximo da realidade da aplicação.

12.4 EXERCÍCIO: TESTANDO A INTEGRAÇÃO COM A INFRAESTRUTURA DE PERSISTÊNCIA

1. Para facilitar a execução dos testes, o Spring já habilita o suporte ao uso de bancos de dados em memória, como o H2 ou HSQLDB de forma automática, bastando apenas contar com a presença de um deles no projeto.

Adicione a dependência do H2 no `pom.xml` :

```
<dependencies>
    <!-- dependências anteriores omitidas -->
    <!-- dependência do Green Mail -->

    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

2. No *source folder* de testes (`/src/test/java`), crie a classe `TopicRepositoryTests` no pacote `br.com.alura.forum.repository` e anote-a com `@DataJpaTest` para carregar o contexto de teste de persistência do Spring Test:

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class TopicRepositoryTests {

}
```

Note que também devemos acrescentar a anotação `@RunWith(SpringRunner.class)` para rodar os testes com o JUnit.

3. Adicione na classe `TopicRepositoryTests` o método de teste `shouldReturnOpenTopicsByCategory()`, que verifica a implementação do método `findOpenTopicsByCategory()`, cuja *query* foi escrita durante este treinamento.

```

@RunWith(SpringRunner.class)
@DataJpaTest
public class TopicRepositoryTests {

    @Autowired
    private TestEntityManager testEntityManager;

    @Autowired
    private TopicRepository topicRepository;

    @Test
    public void shouldReturnOpenTopicsByCategory() {

        TopicRepositoryTestsSetup testSetup = new TopicRepositoryTestsSetup(testEntityManager);
        testSetup.openTopicsByCategorySetup();

        List<OpenTopicsByCategory> openTopics = this.topicRepository.findOpenTopicsByCategory();

        assertThat(openTopics).isNotEmpty();
        assertThat(openTopics).hasSize(2);

        assertThat(openTopics.get(0).getCategoryName())
            .isEqualTo("Programação");
        assertThat(openTopics.get(0).getTopicCount()).isEqualTo(2);

        assertThat(openTopics.get(1).getCategoryName())
            .isEqualTo("Front-end");
        assertThat(openTopics.get(1).getTopicCount()).isEqualTo(1);
    }
}

```

Não esqueça de injetar `TestEntityManager` e `TopicRepository`.

O código ainda não compila. No pacote `br.com.alura.forum.repository.setup`, crie a classe `TopicRepositoryTestsSetup`, que isola a complexidade do setup da base de dados para os cenários de teste.

```

public class TopicRepositoryTestsSetup {

    private TestEntityManager testEntityManager;

    public TopicRepositoryTestsSetup(TestEntityManager testEntityManager) {
        this.testEntityManager = testEntityManager;
    }

    public void openTopicsByCategorySetup() {
        Category programacao = this.testEntityManager
            .persist(new Category("Programação"));
        Category front = this.testEntityManager
            .persist(new Category("Front-end"));

        Category javaWeb = this.testEntityManager
            .persist(new Category("Java Web", programacao));
        Category javaScript = this.testEntityManager
            .persist(new Category("JavaScript", front));

        Course fj27 = this.testEntityManager
            .persist(new Course("Spring Framework", javaWeb));
        Course fj21 = this.testEntityManager
            .persist(new Course("Servlet API e MVC", javaWeb));
    }
}

```

```

        Course js46 = this.testEntityManager
            .persist(new Course("React", javaScript));

        Topic springTopic = new Topic("Tópico Spring", "Conteúdo do tópico", null, fj27);
        Topic servletTopic = new Topic("Tópico Servlet", "Conteúdo do tópico", null, fj21);
        Topic reactTopic = new Topic("Tópico React", "Conteúdo do tópico", null, js46);

        this.testEntityManager.persist(springTopic);
        this.testEntityManager.persist(servletTopic);
        this.testEntityManager.persist(reactTopic);
    }
}

```

4. Rode a classe de teste novamente e veja o teste passar.

5. Para que seja possível testar a camada de persistência em um ambiente mais próximo do de produção, iremos configurar nossos testes para rodar também com o MySQL. Para isso, adicione a anotação
`@AutoConfigureTestDatabase(replace=AutoConfigureTestDatabase.Replace.NONE)` sobre a classe de testes, para que o Spring Test não substitua as configurações do DataSource padrão pela integração automática com o H2.

```

@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace= AutoConfigureTestDatabase.Replace.NONE)
public class TopicRepositoryTests {

    // código interno omitido
}

```

Rode novamente os testes e veja o que acontece com a query que lista a quantidade de tópicos que permanecem abertos no fim de cada dia. Provavelmente, tivemos um erro no `assert`.

6. Para que seja possível rodar os testes em um ambiente de acesso a dados independente, crie o arquivo `application-test.properties` apontando para uma base de testes.

```

# data source
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/fj27_spring_test?createDatabaseIfNotExist=true&
useSSL=false
spring.datasource.username=root
spring.datasource.password=caelum

# jpa properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL55Dialect
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.format_sql=true

```

7. Na classe de testes, ative o perfil `test` para a execução neste ambiente.

```

@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace= AutoConfigureTestDatabase.Replace.NONE)
@ActiveProfiles("test")
public class TopicRepositoryTests {

    // código interno omitido
}

```

}

8. Rode novamente os testes e veja que agora todos passam.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

12.5 TESTANDO NOSSOS CONTROLLERS

Agora que já aprendemos a testar a camada de persistência da nossa aplicação, vamos aprender a testar nossos *controllers* com o Spring Tests. Assim como para a camada de persistência, o Spring Tests possui anotações e classes para facilitar os testes da camada web.

Vamos testar o *endpoint* "/api/topics/{topicId}/answers", mapeado em nosso `AnswerController`. Devemos testar se a requisição processa a resposta esperada.

```
@RunWith(SpringRunner.class)
@SpringBootTest
@Transactional
@ActiveProfiles("test")
public class AnswerControllerTests {

    @Test
    public void shouldProcessSuccessfullyNewAnswerRequest() {
    }
}
```

Note que usaremos a anotação `ActiveProfiles` para considerar o *profile* `test` e `@Transactional` para executar *roll-back* no banco, integrando nosso teste com a camada de persistência (uma vez que essa já esteja testada).

O Spring Tests provê a classe `MockMvc`, representando o ponto de entrada da aplicação, e será usada para simular o processamento de uma requisição. Para que o Spring consiga injetar a classe `MockMvc`, devemos utilizar a anotação `@AutoConfigureMockMvc` na classe `AnswerControllerTests`:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
```

```

@AutoConfigureMockMvc
@Transactional
@ActiveProfiles("test")
public class AnswerControllerTests {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void shouldProcessSuccessfullyNewAnswerRequest() throws Exception {
        this.mockMvc.perform(request);
    }
}

```

O método `perform()` de `MockMvc` executa uma requisição e retorna o tipo `ResultActions`, que permite encadear outras ações como declarações de expectativas de resultados (bastante similar ao `assertThat`). Mas, antes, vamos construir nossa requisição para o *endpoint* `"/api/topics/{topicId}/answers"`. A variável `request` que nosso `mockMvc` espera receber é do tipo `RequestBuilder` - uma interface para construir requisições. Usaremos a implementação `MockHttpServletRequestBuilder` para essa construção. Como faremos uma requisição `post`, já que estamos testando o *endpoint* para criação de uma nova resposta, importaremos o método estático `post()` de `MockMvcRequestBuilders`:

```

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
@Transactional
@ActiveProfiles("test")
public class AnswerControllerTests {

    @Autowired
    private MockMvc mockMvc;

    private static final String ENDPOINT = "/api/topics/{topicId}/answers";

    @Test
    public void shouldProcessSuccessfullyNewAnswerRequest() throws Exception {
        URI uri = new UriTemplate(ENDPOINT).expand("1");
        MockHttpServletRequestBuilder request = post(uri);

        this.mockMvc.perform(request);
    }
}

```

Como o método `post()` recebe uma `URI`, foi necessário criar uma instância da mesma com a ajuda da classe `UriTemplate`, que possui o método `expand()` para definir valores para *path variables* (como o `topicId` neste caso). Depois de executada a requisição, precisamos fazer alguma verificação. Como dito anteriormente, o método `perform()` retorna um `ResultActions` para verificar a resposta:

```
ResultActions response = this.mockMvc.perform(request)
```

A classe `ResultActions` possui o método `andExpect()`, que recebe vários *matchers* para fazer a verificação. Podemos verificar, por exemplo, o *status* da resposta gerada. Como o *endpoint* para processar uma nova resposta exige um usuário logado, neste momento o *status code* esperado é o 401 (*unauthorized*):

```
@Test  
public void shouldProcessSuccessfullyNewAnswerRequest() throws Exception {  
  
    URI uri = new UriTemplate(ENDPOINT).expand("1");  
    MockHttpServletRequestBuilder request = post(uri);  
  
    ResultActions response = this.mockMvc.perform(request);  
    response.andExpect(status().isUnauthorized());  
}
```

Ao rodar o teste, vemos que ele passa:

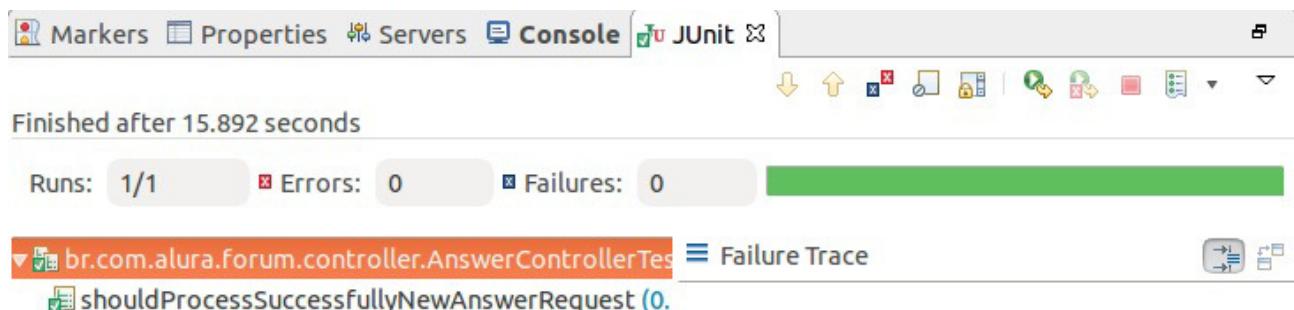


Figura 12.6: teste passa

Mas o cenário do nosso teste deve ser de sucesso, ou seja, nossa requisição deveria retornar o *status* 201 já que deve criar uma nova resposta para um tópico específico. Precisamos montar um cenário simulando uma requisição feita por um usuário logado. Vamos criar um *setup* inicial (como fizemos com o teste do *repository*). Podemos isolar este *setup* em um método de outra classe - mas, como outros cenários de testes podem reaproveitar o mesmo usuário logado e requisição, vamos colocar em um método dentro de nossa própria classe de testes:

```
@RunWith(SpringRunner.class)  
@SpringBootTest  
@AutoConfigureMockMvc  
@Transactional  
@ActiveProfiles("test")  
public class AnswerControllerTests {  
  
    // atributos e métodos omitidos  
  
    public void setUp() {  
        // setup logic  
    }  
}
```

Precisamos, antes de tudo, de um usuário logado. Vamos começar criando um usuário:

```
public void setUp() {
```

```

String rawPassword = "123456";
User user = new User("Aluno da Alura", "aluno@gmail.com",
    new BCryptPasswordEncoder().encode(rawPassword));
User persistedUser = this.userRepository.save(user);
}

```

Vamos precisar também de um tópico persistido e seu `id` para executar com sucesso nossa requisição. Criaremos um atributo chamado `topicId` no escopo da classe para inicializá-lo no `setUp()` e utilizá-lo no nosso método de testes:

```

public void setUp() {
    String rawPassword = "123456";
    User user = new User("Aluno da Alura", "aluno@gmail.com",
        new BCryptPasswordEncoder().encode(rawPassword));
    User persistedUser = this.userRepository.save(user);

    Topic topic = new Topic("Descrição do Tópico", "Conteúdo do Tópico",
        persistedUser, null);
    Topic persistedTopic = this.topicRepository.save(topic);
    this.topicId = persistedTopic.getId();
}

```

Agora será preciso logar o usuário. Um usuário logado deve ter um *token* de autenticação. Vamos precisar do `AuthenticationManager` para autenticar o usuário e do `TokenManager` para gerar o *token*, como visto no capítulo 8. Também criaremos um atributo `jwt` no escopo da classe para utilizá-lo nos métodos de testes após a execução do `setUp()`:

```

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
@Transactional
@ActiveProfiles("test")
public class AnswerControllerTests {

    // outros atributos e métodos omitidos

    @Autowired
    private AuthenticationManager authManager;

    @Autowired
    private TokenManager tokenManager;

    private Long topicId;
    private String jwt;

    public setUp() {
        Topic topic = new Topic("Descrição do Tópico", "Conteúdo do Tópico",
            persistedUser, null);
        Topic persistedTopic = this.topicRepository.save(topic);
        this.topicId = persistedTopic.getId();

        Authentication authentication = authManager.authenticate(
            new UsernamePasswordAuthenticationToken(user.getEmail(), rawPassword));
        this.jwt = this.tokenManager.generateToken(authentication);
    }
}

```

Para que este método seja executado antes de qualquer método de testes, utilizaremos a anotação

@Before do JUnit, que vai garantir este comportamento:

```
@Before
public void setUp() {
    Topic topic = new Topic("Descrição do Tópico", "Conteúdo do Tópico",
                           persistedUser, null);
    Topic persistedTopic = this.topicRepository.save(topic);
    this.topicId = persistedTopic.getId();

    Authentication authentication = authManager.authenticate(
        new UsernamePasswordAuthenticationToken(user.getEmail(), rawPassword));
    this.jwt = this.tokenManager.generateToken(authentication);
}
```

Com o `setUp()` pronto, podemos construir a requisição com um usuário logado e um tópico persistido. Além disso, precisamos enviar uma resposta válida, que, segundo nossa regra de negócio, deve ter no mínimo 5 caracteres. Vamos criar uma instância de `NewAnswerInputDto` com um conteúdo válido e enviá-lo no corpo da requisição - também devemos informar que o formato será em `JSON`. Podemos utilizar os métodos `content()` e `contentType()`, também de `MockMvcRequestBuilders`, de maneira encadeada:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
@Transactional
@ActiveProfiles("test")
public class AnswerControllerTests {

    // atributos omitidos

    // método setUp() omitido

    @Test
    public void shouldProcessSuccessfullyNewAnswerRequest() throws Exception {
        URI uri = new UriTemplate(ENDPOINT).expand(this.topicId);

        NewAnswerInputDto inputDto = new NewAnswerInputDto();
        inputDto.setContent("Não consigo subir servidor");

        MockHttpServletRequestBuilder request = post(uri)
            .contentType(MediaType.APPLICATION_JSON)
            .content(new ObjectMapper().writeValueAsString(inputDto));
    }
}
```

Note que utilizamos uma instância de `ObjectMapper` da biblioteca `jackson` para converter o objeto `inputDto` em uma String `JSON` pelo método `writeValueAsString()`. Agora precisamos definir o usuário logado na requisição. Como aprendemos no capítulo de segurança, o `token` deve ser enviado pelo cabeçalho da requisição. A classe `MockMvcRequestBuilders` também possui o método `header()`, que usaremos para enviar o `token`:

```
@Test
public void shouldProcessSuccessfullyNewAnswerRequest() throws Exception {
```

```

URI uri = new UriTemplate(ENDPOINT).expand(this.topicId);

NewAnswerInputDto inputDto = new NewAnswerInputDto();
inputDto.setContent("Não consigo subir servidor");

MockHttpServletRequestBuilder request = post(uri)
    .header("Authorization", "Bearer " + this.jwt)
    .contentType(MediaType.APPLICATION_JSON)
    .content(new ObjectMapper().writeValueAsString(inputDto));
}

}

```

Por fim, vamos processar a requisição e definir que esperamos o *status code* 201 (*created*) e o conteúdo `inputDto.getContent()`. Usaremos o método `andExpect()` com a ajuda dos métodos `status()` e `content()`, do import estático de `org.springframework.test.web.servlet.result.MockMvcResultMatchers`:

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
@Transactional
@ActiveProfiles("test")
public class AnswerControllerTests {

    // atributos omitidos

    // método setUp() omitido

    @Test
    public void shouldProcessSuccessfullyNewAnswerRequest() throws Exception {
        URI uri = new UriTemplate(ENDPOINT).expand(this.topicId);

        NewAnswerInputDto inputDto = new NewAnswerInputDto();
        inputDto.setContent("Não consigo subir servidor");

        MockHttpServletRequestBuilder request = post(uri)
            .contentType(MediaType.APPLICATION_JSON)
            .header("Authorization", "Bearer " + this.jwt)
            .content(new ObjectMapper().writeValueAsString(inputDto));

        this.mockMvc.perform(request)
            .andExpect(status().isCreated())
            .andExpect(content()
                .string(containsString(inputDto.getContent())));
    }
}

```

Agora, vemos o teste passar:

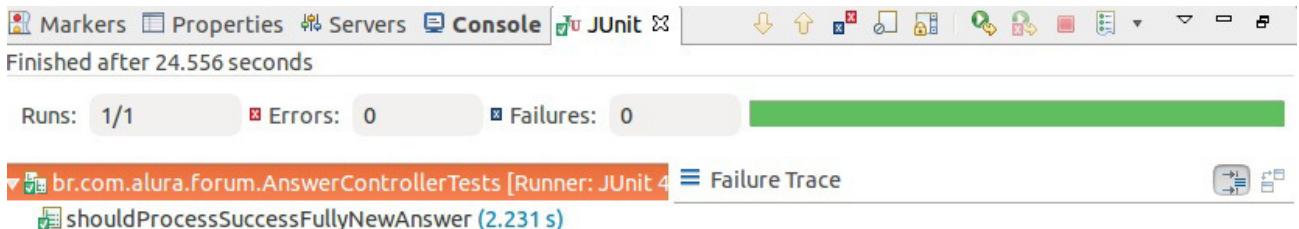


Figura 12.7: teste passa

Também podemos criar um outro cenário em que o teste deve falhar: enviar um conteúdo com menos de 5 caracteres, por exemplo. Podemos aproveitar o mesmo `setUp()` e esperarmos que o *status code* agora seja 400, ou seja, *bad request*:

```
@Test
public void shouldRejectNewAnswerRequest() throws Exception {
    URI uri = new UriTemplate(ENDPOINT).expand(this.topicId);

    NewAnswerInputDto inputDto = new NewAnswerInputDto();
    inputDto.setContent("bad");

    MockHttpServletRequestBuilder request = post(uri)
        .contentType(MediaType.APPLICATION_JSON)
        .header("Authorization", "Bearer " + this.jwt)
        .content(new ObjectMapper().writeValueAsString(inputDto));

    this.mockMvc.perform(request)
        .andExpect(status().isBadRequest());
}
```

Ao rodar os teste novamente, vemos que este também passa:



Figura 12.8: teste passa

12.6 EXERCÍCIO: TESTANDO A CAMADA WEB DA APLICAÇÃO

- Para que seja possível testar a execução da chamada para os *endpoints*, também vamos contar com o suporte do Spring Test. Ainda no *source folder* de testes, no pacote `br.com.alura.forum.controller`, crie a classe `AnswerControllerTests` com o teste do *endpoint* de inserção de novas respostas a um tópico.

```
@RunWith(SpringRunner.class)
```

```

@SpringBootTest
@AutoConfigureMockMvc
@Transactional
@ActiveProfiles("test")
public class AnswerControllerTests {

    private static final String ENDPOINT = "/api/topics/{topicId}/answers";

    private Long topicId;
    private String jwt;

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void shouldProcessSuccessfullyNewAnswerRequest() throws Exception {
        URI uri = new UriTemplate(ENDPOINT).expand(this.topicId);

        NewAnswerInputDto inputDto = new NewAnswerInputDto();
        inputDto.setContent("Não consigo subir servidor");

        MockHttpServletRequestBuilder request = post(uri)
            .contentType(MediaType.APPLICATION_JSON)
            .header("Authorization", "Bearer " + this.jwt)
            .content(new ObjectMapper().writeValueAsString(inputDto));

        this.mockMvc.perform(request)
            .andExpect(status().isCreated())
            .andExpect(content()
                .string(containsString(inputDto.getContent())));
    }
}

```

Obs: Os métodos: `post()` , `status()` , `content()` e `containsString()` são dos seguintes imports estáticos:

```

import static org.hamcrest.CoreMatchers.containsString;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

```

2. Caso você tenha executado o teste com a implementação acima, percebeu que ainda é impossível fazer com que ele passe.

Para que o request seja executado corretamente, precisamos prover um setup mínimo de dados sobre um tópico ao qual a resposta será adicionada, e um usuário, que em nosso cenário será o autor tanto do tópico como da nova resposta que estamos processando.

Adicione o método `setup()` contendo com a configuração para o teste.

```

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
@Transactional
@ActiveProfiles("test")
public class AnswerControllerTests {

    // atributos anteriores omitidos

    @Autowired

```

```

private TopicRepository topicRepository;

@Autowired
private UserRepository userRepository;

@Autowired
private TokenManager tokenManager;

@Autowired
private AuthenticationManager authManager;

@Before
public void setup() throws RuntimeException {

    String rawPassword = "123456";
    User user = new User("Aluno da Alura", "aluno@gmail.com",
        new BCryptPasswordEncoder().encode(rawPassword));
    User persistedUser = this.userRepository.save(user);

    Topic topic = new Topic("Descrição do Tópico", "Conteúdo do Tópico",
        persistedUser, null);
    Topic persistedTopic = this.topicRepository.save(topic);
    this.topicId = persistedTopic.getId();

    Authentication authentication = authManager.authenticate(
        new UsernamePasswordAuthenticationToken(user.getEmail(), rawPassword));
    this.jwt = this.tokenManager.generateToken(authentication);
}

// código do teste omitido
}

```

Perceba a necessidade de contar com a injeção das dependências necessárias ao *setup*.

Adicione também a assinatura do método `User save(User user)`; à interface `UserRepository`, que em um cenário real no Alura Fórum já existiria.

Crie também, na classe `User`, um construtor que recebe como parâmetro o nome, o email e a senha do usuário:

```

public User(String name, String email, String password) {
    this.name = name;
    this.email = email;
    this.password = password;
}

```

3. Rode a classe de testes e veja como tudo corre bem.

4. Adicione um novo cenário de teste onde é adicionada uma resposta inválida para avaliar como a aplicação se comporta.

```

@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
@Transactional
@ActiveProfiles("test")
public class AnswerControllerTests {

    // atributos omitidos
}

```

```

// setup omitido

// primeiro teste omitido

@Test
public void shouldRejectNewAnswerRequest() throws Exception {
    URI uri = new UriTemplate(ENDPOINT).expand(this.topicId);

    NewAnswerInputDto inputDto = new NewAnswerInputDto();
    inputDto.setContent("bad");

    MockHttpServletRequestBuilder request = post(uri)
        .contentType(MediaType.APPLICATION_JSON)
        .header("Authorization", "Bearer " + this.jwt)
        .content(new ObjectMapper().writeValueAsString(inputDto));

    this.mockMvc.perform(request)
        .andExpect(status().isBadRequest());
}
}

```

5. Rode novamente a classe de testes para garantir que também funciona como esperado.

MONITORANDO A API COM SPRING BOOT

13.1 SPRING ACTUATOR

O termo *actuator* é utilizado na manufatura para referenciar um dispositivo mecânico usado para mover e controlar alguma coisa. O módulo Spring Boot Actuator inclui várias funcionalidades para ajudar a monitorar e administrar a aplicação depois de colocada em produção. O monitoramento é feito através de *endpoints* que são habilitados após adicionarmos a dependência do Actuator no arquivo `pom.xml` :

```
<dependencies>
    <!-- dependência do h2database -->

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

Os *endpoints* do Actuator permitem monitorar e interagir com a aplicação e precisam do Spring MVC para funcionar através do protocolo HTTP. Por padrão, alguns *endpoints* já estão habilitados no Actuator. Para usá-los, basta acessar o Actuator pelo endereço `http://localhost:8080/actuator` :

```
localhost:8080/actuator

JSON Raw Data Headers
Save Copy Collapse All Expand All

{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "health-component": {
      "href": "http://localhost:8080/actuator/health/{component}",
      "templated": true
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    "health-component-instance": {
      "href": "http://localhost:8080/actuator/health/{component}/{instance}",
      "templated": true
    },
    "info": {
      "href": "http://localhost:8080/actuator/info",
      "templated": false
    }
  }
}
```

Figura 13.1: Tela Actuator

Por exemplo, o endpoint `/health` provê informações básicas sobre a saúde da aplicação. Podemos acessá-lo clicando no endereço `http://localhost:8080/actuator/health`:

```
localhost:8080/actuator/health

JSON Raw Data Headers
Save Copy Collapse All Expand All

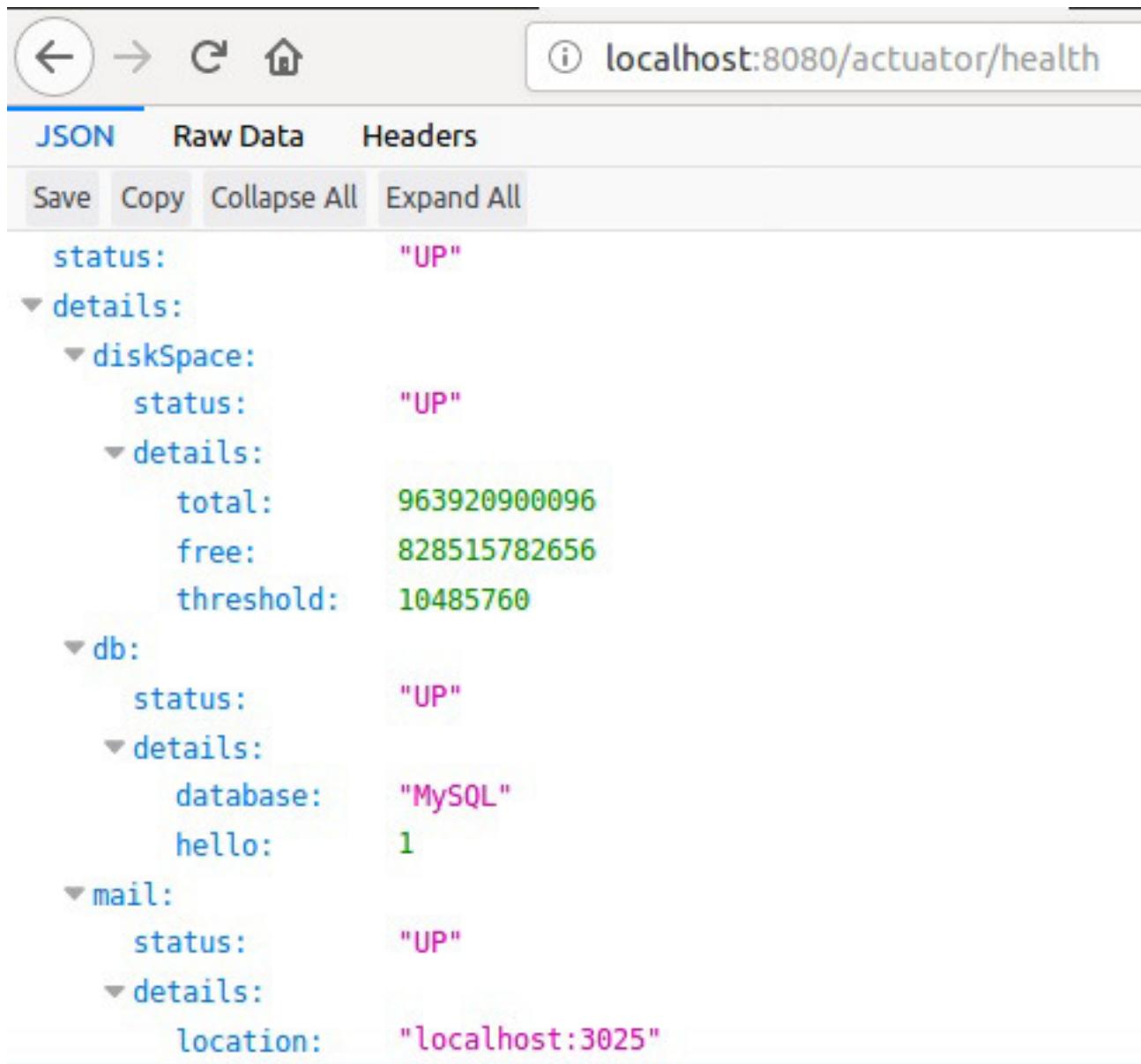
{
  "status": "UP"
}
```

Figura 13.2: Tela Actuator Health

Veja que aparece apenas o `status` da aplicação, com o valor `UP`. Isso quer dizer que a aplicação está rodando. Por padrão, o Actuator não mostra os detalhes deste endpoint. Para que possamos ter acesso a mais detalhes sobre a saúde da aplicação, podemos acrescentar a configuração abaixo no arquivo

```
application.properties :  
  
management.endpoint.health.show-details=always
```

Agora, ao acessar novamente o endpoint `http://localhost:8080/actuator/health`, obtemos o seguinte resultado:



The screenshot shows a browser window with the URL `localhost:8080/actuator/health`. The interface has tabs for `JSON`, `Raw Data`, and `Headers`. Below the tabs are buttons for `Save`, `Copy`, `Collapse All`, and `Expand All`. The main content area displays a JSON object representing the application's health status. The JSON structure includes fields for `status` (set to "UP"), `details` (which further branches into `diskSpace`, `db`, `mail`, and `hello` components), and specific metrics like total disk space (963920900096), free disk space (828515782656), and MySQL database hello count (1). The JSON is color-coded with blue for keys and pink for values.

```
status: "UP"  
details:  
  diskSpace:  
    status: "UP"  
    details:  
      total: 963920900096  
      free: 828515782656  
      threshold: 10485760  
  db:  
    status: "UP"  
    details:  
      database: "MySQL"  
      hello: 1  
  mail:  
    status: "UP"  
    details:  
      location: "localhost:3025"
```

Figura 13.3: Tela Actuator Health Details

Além do `status` da aplicação, podemos checar informações sobre o espaço em disco utilizado, o banco de dados e o servidor de email. Estas informações são ligadas a infraestrutura da aplicação.

Caso você queira acessar especificamente apenas uma destas informações, você pode utilizar o endpoint `http://localhost:8080/actuator/health/{component}` substituindo `{component}` por

um dos componentes listados. Por exemplo, `http://localhost:8080/actuator/health/db`.

The screenshot shows a browser window with the URL `localhost:8080/actuator/health/db` in the address bar. The page displays a JSON response. The `status` field is set to "UP". The `details` section contains two entries: `database` set to "MySQL" and `hello` set to 1.

```
status: "UP"
▼ details:
  database: "MySQL"
  hello: 1
```

Figura 13.4: Tela Actuator Health DB

Também é possível customizar o `endpoint /health` e adicionar mais indicadores a ele. Basta criar um classe gerenciada pelo Spring e implementar a interface `HealthIndicator`.

Outro `endpoint` habilitado por padrão é o `/info`, mas ao acessá-lo não vemos nada aparecer. Podemos configurar para aparecer alguma coisa. A ideia do `/info` é mostrar algumas informações sobre a aplicação, como nome, versão, etc... Já possuímos todas estas informações no arquivo `pom.xml` e vamos reaproveitá-las para mostrar neste `endpoint` do Actuator. Para isso, basta adicionar as propriedades abaixo no arquivo `application.properties`:

```
#actuator properties
management.endpoint.health.show-details=always
info.app.name=@project.name@
info.app.description=@project.description@
info.app.version=@project.version@
info.app.encoding=@project.build.sourceEncoding@
info.app.java.version=@java.version@
```

Ao rodar novamente a aplicação e acessar o `endpoint http://localhost:8080/actuator/info`, obtemos o resultado:

```
localhost:8080/actuator/info

JSON Raw Data Headers
Save Copy Collapse All Expand All

▼ app:
  name: "Alura Forum"
  ▼ description: "Projeto base do curso FJ-27 Spring Framework da Caelum"
  version: "0.0.1-SNAPSHOT"
  encoding: "UTF-8"
  ▼ java:
    version: "1.8.0_201"
```

Figura 13.5: Tela Actuator Info Details

Para saber mais

Caso sua aplicação esteja integrada com outra ferramenta de monitoramento que implemente a especificação JMX (Java Management Extensions), o Actuator disponibiliza essa informação via `@JmxEndpoint` além do protocolo HTTP. Para mais informações acesse a [documentação](#).

Já conhece os cursos online Alura?



A Alura oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex aluno da Caelum tem 15% de desconto neste link!

[Conheça os cursos online Alura.](#)

13.2 OUTROS ENDPOINTS

O Actuator possui mais *endpoints* que não estão habilitados por padrão. Para habilitá-los, acrescentaremos a seguinte propriedade no arquivo `application.properties`:

```
management.endpoints.web.exposure.include=*
```

Agora, quando acessamos `http://localhost:8080/actuator`, obtemos:

The screenshot shows a browser window with the URL `localhost:8080/actuator`. The page has a header with back, forward, refresh, and home icons. Below the header, there are tabs for `JSON`, `Raw Data`, and `Headers`. Under the `JSON` tab, there are buttons for `Save`, `Copy`, `Collapse All`, and `Expand All`. The main content is a JSON object representing the available endpoints:

```
{  
  "_links": {  
    "self": {  
      "href": "http://localhost:8080/actuator"  
    },  
    "auditevents": {  
      "href": "http://localhost:8080/actuator/auditevents"  
    },  
    "beans": {  
      "href": "http://localhost:8080/actuator/beans"  
    },  
    "caches": {  
      "href": "http://localhost:8080/actuator/caches"  
    },  
    "caches-cache": {  
      "href": "http://localhost:8080/actuator/caches/{cache}"  
    },  
    "health": {  
      "href": "http://localhost:8080/actuator/health"  
    },  
    "health-component": {  
      "href": "http://localhost:8080/actuator/health/{component}"  
    },  
    "health-component-instance": {  
      "href": "http://localhost:8080/actuator/health/{component}/{instance}"  
    },  
    "conditions": {  
      "href": "http://localhost:8080/actuator/conditions"  
    },  
    "configprops": {  
      "href": "http://localhost:8080/actuator/configprops"  
    },  
    "env-toMatch": {  
      "href": "http://localhost:8080/actuator/env/toMatch"  
    }  
  }  
}
```

Figura 13.6: Tela Actuator All Links

O Actuator possui muitos *endpoints* definidos e é possível acessar muitas informações e métricas da aplicação. Por exemplo, `/httptrace` exibe informações de rastreio HTTP (com as últimas 100

requisições feitas para a aplicação). O endpoint `/mappings` exibe uma lista ordenada de todos os `@RequestMappings`, e `/metrics/{id}` exibe uma lista de ids que representam outros endpoints para métricas da aplicação, como informações sobre a JVM ou sobre o Tomcat, por exemplo. Para adicionar as métricas do `Hibernate`, inclua a propriedade abaixo no arquivo `application.properties`:

```
spring.jpa.properties.hibernate.generate_statistics=true
```

Rode novamente a aplicação, acesse o endpoint `/metrics` e veja que agora são disponibilizadas as métricas do `Hibernate` que inclui, por exemplo, o número de `queries` e transações realizadas e a quantidade de entidades carregadas.

Para saber mais

O Spring Boot Actuator também inclui um endpoint para derrubar a aplicação, o `/shutdown`. Ele possui uma configuração específica por ser muito perigoso - visto que derruba a aplicação através de uma simples requisição. Para habilitar este endpoint, basta incluir a seguinte configuração no arquivo `application.properties`:

```
management.endpoint.shutdown.enabled=true
```

Como o endpoint apenas aceita requisições POST, abra o terminal e digite o seguinte comando:

```
curl -X POST localhost:8080/actuator/shutdown
```

Em caso de sucesso, você verá a seguinte mensagem de saída:

```
{"message": "Shutting down, bye..."}  
}
```

13.3 CRIANDO UM NOVO ENDPOINT

Além dos endpoints que vêm por padrão com o Actuator, podemos construir um endpoint com informações customizadas. Vamos construir um endpoint que mostra todas as dúvidas abertas por categoria. Já temos essa lógica pronta na aplicação, basta configurarmos o endpoint. Para isso, o Actuator possui a anotação `@Endpoint` para identificar um endpoint que provê informações durante a execução da aplicação.

No pacote, `br.com.alura.forum.actuator.endpoints`, criaremos a classe `UnansweredTopicsActuatorEndpoint` com essa anotação, além da anotação `@Component`:

```
@Endpoint  
@Component  
public class UnansweredTopicsActuatorEndpoint {  
}
```

Em seguida injetamos o `TopicRepository` para chamar o método `findOpenTopicsByCategory()` para trazer todos os tópicos abertos por categoria:

```
@Endpoint  
@Component  
public class UnansweredTopicsActuatorEndpoint {  
  
    @Autowired  
    private TopicRepository topicRepository;  
  
    public List<OpenTopicsByCategory> execute() {  
        return topicRepository.findOpenTopicsByCategory();  
    }  
}
```

Como este *endpoint* é apenas de leitura e será executado via método HTTP GET, devemos anotar o método com `@ReadOperation`. O Actuator também permite operações de escrita e remoção (via métodos HTTP POST e DELETE) - basta usar as anotações `@WriteOperation` e `@DeleteOpration`, respectivamente.

Para definir um *endpoint*, a anotação `@Endpoint` possui um parâmetro `id` que é um identificador para um *endpoint* do Actuator. Eles podem conter apenas letras minúsculas, números e os caracteres ponto(".") e hífen("-"). Vamos definir o nosso como `open-topics-by-category`:

```
@Endpoint(id="open-topics-by-category")  
@Component  
public class UnansweredTopicsActuatorEndpoint {  
  
    @Autowired  
    private TopicRepository topicRepository;  
  
    @ReadOperation  
    public List<OpenTopicsByCategory> execute() {  
        return topicRepository.findOpenTopicsByCategory();  
    }  
}
```

Por fim, como o Actuator expõe informações sensíveis da aplicação, não queremos que elas fiquem públicas. Vamos configurar para que apenas usuários administradores possam ter acesso a elas. Podemos criar um contexto específico de segurança para isso ou aproveitar o que já fizemos anteriormente para acessar o relatório de dúvidas abertas.

Para utilizar o *entry point* de segurança configurado a partir da classe `AdminSecurityConfiguration`, basta que o acesso ao Actuator seja modificado para `http://localhost:8080/admin/actuator`. Para isso, acrescente a seguinte configuração no arquivo `application.properties`:

```
management.endpoints.web.base-path=/admin/actuator
```

Rode novamente a aplicação e tente acessar `http://localhost:8080/admin/actuator`. Perceba

que agora a autenticação com email e senha para o acesso é exigida.

13.4 PARA SABER MAIS: SPRING BOOT ADMIN

Spring Boot Admin é um projeto que provê uma interface para aplicações com Spring Boot. Para funcionar, é preciso adicionar as dependências:

```
<dependencies>
    <!-- outras dependências -->

    <dependency>
        <groupId>de.codecentric</groupId>
        <artifactId>spring-boot-admin-server</artifactId>
        <version>2.1.4</version>
    </dependency>

    <dependency>
        <groupId>de.codecentric</groupId>
        <artifactId>spring-boot-admin-server-ui</artifactId>
        <version>2.1.4</version>
    </dependency>

    <dependency>
        <groupId>de.codecentric</groupId>
        <artifactId>spring-boot-admin-starter-client</artifactId>
        <version>2.1.4</version>
    </dependency>
</dependencies>
```

E adicionar as seguintes propriedades no `application.properties`:

```
spring.boot.admin.url=http://localhost:8080/
spring.boot.admin.client.url=http://localhost:8080/
management.security.enabled=false
```

A configuração acima desabilita a segurança para o Spring Boot Admin e optamos por fazer isso apenas para testar seu funcionamento. Para mais informações de customização do Spring Boot Admin, acesse o [site da ferramenta](#).

Com a configuração pronta, acesse `http://localhost:8080/` e clique em `Wallboard`. A tela abaixo deve aparecer:

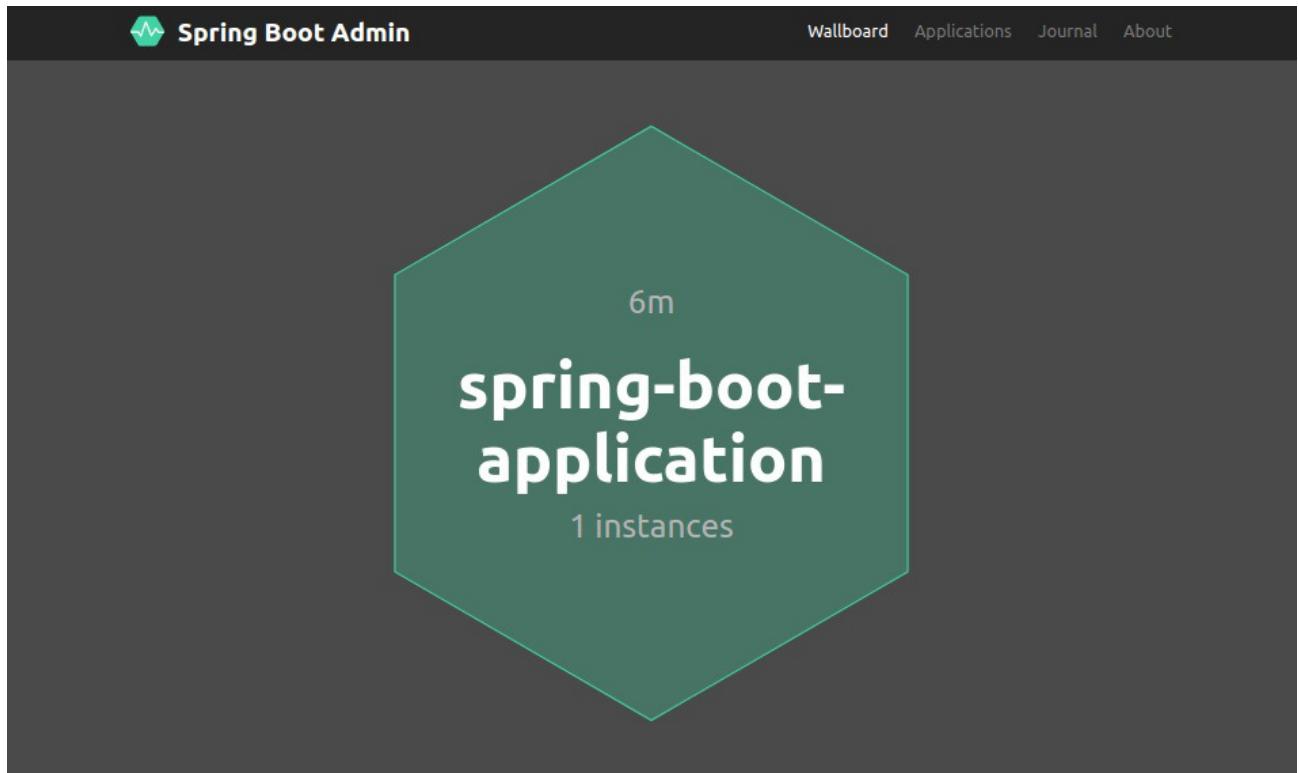


Figura 13.7: Spring Boot Admin Interface

Clique na instância da aplicação e navegue pelos *endpoints* do Actuator com a interface gráfica do Spring Boot Admin:

spring-boot-application
79324201107b

spring-boot-application
Id: 79324201107b

http://caelum-thais-dev:8080/ http://caelum-thais-dev:8080/actuator http://caelum-thais-dev:8080/actuator/health

Info		Health	
app name: Alura Forum description: Projeto base do curso FJ-27 Spring Framework version: 0.0.3-SNAPSHOT encoding: UTF-8 java: version: 1.8.0_201		Instance diskSpace total: 964 GB free: 815 GB threshold: 10.5 MB db database: MySQL hello: 1 mail	
Metadata startup: 2019-04-29T16:08:31.096-03:00			

Figura 13.8: Spring Boot Admin Interface

Saber inglês é muito importante em TI

aluralíngua

cotidianas. Além disso, todas as aulas possuem explicações gramaticais, para você entender completamente o que está aprendendo. Aprender inglês é fundamental para o profissional de tecnologia de sucesso!

Na **Alura Língua** você reforça e aprimora seu inglês! Usando a técnica *Spaced Repetitions* o aprendizado naturalmente **se adapta ao seu conhecimento**. Exercícios e vídeos interativos fazem com que você pratique em situações

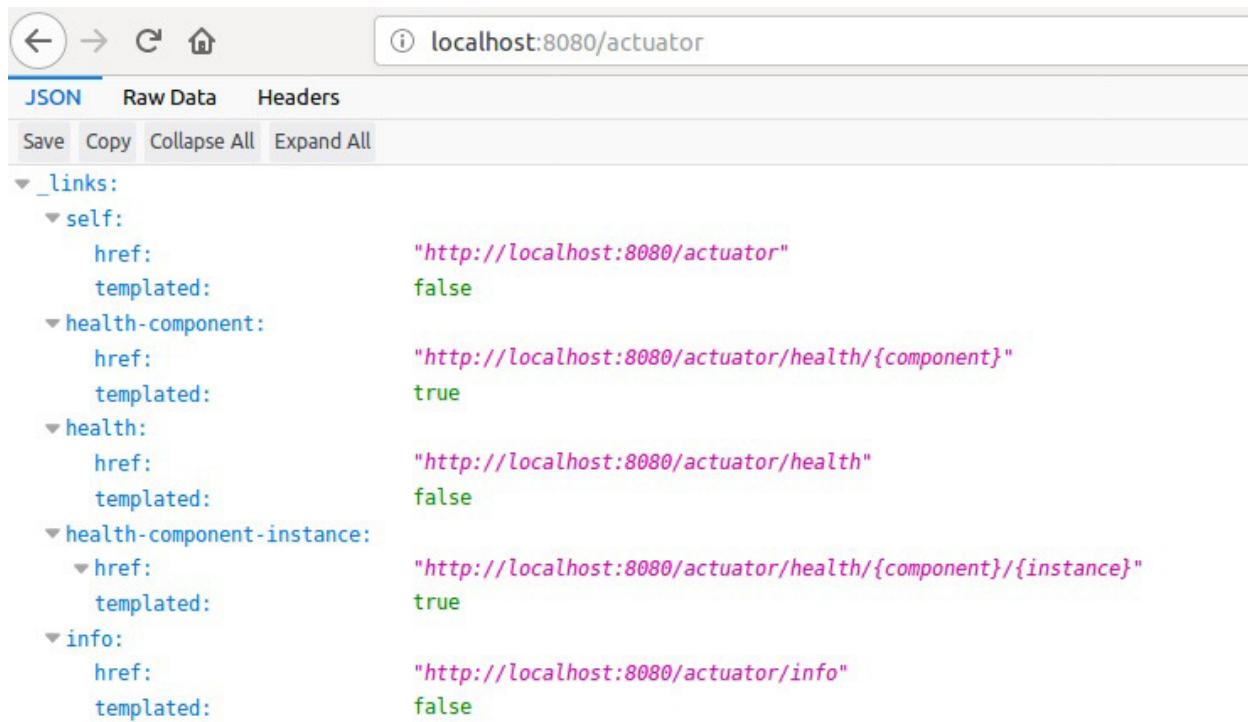
[Pratique seu inglês na Alura Língua.](#)

13.5 EXERCÍCIOS: HABILITANDO O ACTUATOR E CRIANDO NOVO ENDPOINT

1. Vamos habilitar o Spring Boot Actuator, que expõe informações operacionais sobre o aplicativo em execução. Primeiro, vamos incluir as dependências no arquivo `pom.xml` :

```
<dependencies>  
    <!-- dependência do h2database -->  
  
    <dependency>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-actuator</artifactId>  
    </dependency>  
  
</dependencies>
```

2. O Actuator já está habilitado. Rode a aplicação e acesse o endereço `http://localhost:8080/actuator`



The screenshot shows a browser window with the URL `localhost:8080/actuator` in the address bar. The page displays a JSON object representing the available endpoints. The JSON structure includes fields for `_links`, `self`, `health-component`, `health`, `health-component-instance`, and `info`. Each field contains a `href` and a `templated` value.

```
{  
  "_links": {  
    "self": {  
      "href": "http://localhost:8080/actuator",  
      "templated": false  
    },  
    "health-component": {  
      "href": "http://localhost:8080/actuator/health/{component}",  
      "templated": true  
    },  
    "health": {  
      "href": "http://localhost:8080/actuator/health",  
      "templated": false  
    },  
    "health-component-instance": {  
      "href": "http://localhost:8080/actuator/health/{component}/{instance}",  
      "templated": true  
    },  
    "info": {  
      "href": "http://localhost:8080/actuator/info",  
      "templated": false  
    }  
  }  
}
```

Figura 13.9: Tela do Actuator

Note que o Actuator, por padrão, já vem configurado com os endpoints `health` e `info` como públicos. Acesse cada um deles e veja as informações.

3. Vamos configurar para que `/health` exiba mais detalhes. Para isso, acrescente a seguinte configuração no arquivo `application.properties` :

```
#actuator  
management.endpoint.health.show-details=always
```

Teste acessando `http://localhost:8080/actuator/health` e veja que mais detalhes como o

espaço utilizado em disco, o banco de dados utilizado e o servidor de email agora aparecem.

4. O *endpoint* `/info` não mostra nenhuma informação e isso também pode ser personalizado. Acrescente as configurações abaixo para que apresentemos algumas informações contidas no arquivo `pom.xml` como o nome do projeto e sua versão:

```
info.app.name=@project.name@  
info.app.description=@project.description@  
info.app.version=@project.version@  
info.app.encoding=@project.build.sourceEncoding@  
info.app.java.version=@java.version@
```

Teste acessando `http://localhost:8080/actuator/info` e veja que nossas informações agora são apresentadas.

5. Podemos habilitar todos os *endpoints* do Actuator. Para habilitá-los, acrescente a seguinte configuração no arquivo `application.properties`:

```
management.endpoints.web.exposure.include=*
```

Rode novamente a aplicação e acesse `http://localhost:8080/actuator/`. Note que a lista de endpoints aumentou já que todos os *endpoints* foram habilitados. Navegue por alguns e veja algumas informações sobre a aplicação.

```
localhost:8080/actuator

JSON Raw Data Headers
Save Copy Collapse All Expand All

{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "auditevents": {
      "href": "http://localhost:8080/actuator/auditevents",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:8080/actuator/beans",
      "templated": false
    },
    "caches": {
      "href": "http://localhost:8080/actuator/caches",
      "templated": false
    },
    "caches-cache": {
      "href": "http://localhost:8080/actuator/caches/{cache}",
      "templated": true
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    "health-component": {
      "href": "http://localhost:8080/actuator/health/{component}",
      "templated": true
    },
    "health-component-instance": {
      "href": "http://localhost:8080/actuator/health/{component}/{instance}",
      "templated": true
    },
    "conditions": {
      "href": "http://localhost:8080/actuator/conditions",
      "templated": false
    },
    "configprops": {
      "href": "http://localhost:8080/actuator/configprops",
      "templated": false
    },
    "env-toMatch": {
      "href": "http://localhost:8080/actuator/env/toMatch"
    }
  }
}
```

Figura 13.10: Tela do Actuator

Por exemplo, `/httptrace` exibe informações de rastreio HTTP (com as últimas 100 requisições feitas para a aplicação). O endpoint `/mappings` exibe uma lista ordenada de todos os `@RequestMappings`, e `/metrics/{id}` exibe uma lista de ids que representam outros endpoints para métricas da aplicação, como informações sobre a JVM ou sobre o Tomcat, por exemplo.

6. Além dos endpoints que vêm por padrão com Actuator, podemos construir um endpoint com informações customizadas. Vamos construir um endpoint que mostra todas as dúvidas abertas por categoria. Já temos essa lógica pronta na aplicação, basta configurarmos o endpoint. No pacote,

br.com.alura.forum.actuator.endpoints , crie a classe
UnansweredTopicsActuatorEndpoint :

```
@Component
public class UnansweredTopicsActuatorEndpoint {  
}
```

7. Crie o método `execute()` que deve retornar a lista de todas as dúvidas abertas por categoria:

```
@Component
public class UnansweredTopicsActuatorEndpoint {  
  
    @Autowired
    private TopicRepository topicRepository;  
  
    public List<OpenTopicsByCategory> execute() {
        return topicRepository.findOpenTopicsByCategory();
    }
}
```

8. Anote a classe com `@Endpoint` para expor a classe como um novo *endpoint* do Actuator e anote o método `execute()` com `@ReadOperation` para identificar o método como uma operação de leitura.

```
@Endpoint(id="open-topics-by-category")
@Component
public class UnansweredTopicsActuatorEndpoint {  
  
    @Autowired
    private TopicRepository topicRepository;  
  
    @ReadOperation
    public List<OpenTopicsByCategory> execute() {
        return topicRepository.findOpenTopicsByCategory();
    }
}
```

O `id` é usado para mapear o *endpoint*.

9. Acesse novamente `http://localhost:8080/actuator` e veja que agora o *endpoint* criado aparece na lista como `http://localhost:8080/actuator/open-topics-by-category`. Acesse-o e veja o resultado.

```

▼ _links:
  ▼ self:
    href: "http://localhost:8080/actuator"
    templated: false
  ▼ open-topics-by-category:
    ▼ href:
      href: "http://localhost:8080/actuator/open-topics-by-category"
      templated: false
  ▼ auditevents:
    href: "http://localhost:8080/actuator/auditevents"
    templated: false
  ▼ beans:
    href: "http://localhost:8080/actuator/beans"
    templated: false
  ▼ caches:

```

Figura 13.11: Tela do Actuator

10. Como o Actuator expõe informações sensíveis da aplicação, não queremos que elas fiquem públicas. Vamos configurar para que apenas usuários administradores possam ter acesso a elas. Podemos criar um contexto específico de segurança para isso ou aproveitar o que já fizemos anteriormente para acessar o relatório de dúvidas abertas.

Para utilizar o *entry point* de segurança configurado a partir da classe `AdminSecurityConfiguration`, basta que o acesso ao Actuator seja modificado para `http://localhost:8080/admin/actuator`. Para isso, acrescente a seguinte configuração no arquivo `application.properties`:

```
management.endpoints.web.base-path=/admin/actuator
```

Rode novamente a aplicação e tente acessar `http://localhost:8080/admin/actuator`. Perceba que agora a autenticação com email e senha para o acesso é exigida.

11. Acrescente todas as configurações que fizemos até aqui no arquivo `application.properties` no arquivo `application-prod.properties`:

```
#actuator
management.endpoint.health.show-details=always
info.app.name=@project.name@
info.app.description=@project.description@
info.app.version=@project.version@
info.app.encoding=@project.build.sourceEncoding@
info.app.java.version=@java.version@
management.endpoints.web.exposure.include=*
management.endpoints.web.base-path=/admin/actuator
```

APÊNDICE: MELHORANDO A PERFORMANCE COM CACHE

Sempre que entramos no Fórum da Alura, nos é apresentado a lista de todos os tópicos. Além disso, é mostrado o *dashboard* com as informações e estatísticas dos tópicos por categoria. O leitor mais atento já deve ter percebido que esse *dashboard* não muda muito dentro de um determinado intervalo de tempo. E o fato de não mudar nos leva a um questionamento: Será que realmente precisamos ficar executando a lógica do método do *controller* em todas as vezes que um usuário acessar esse recurso?

14.1 COMEÇANDO COM SPRING CACHE

Um dos pontos onde podemos ganhar em tempo de execução dentro de uma aplicação é quando não necessitamos realizar a mesma lógica muitas vezes - por exemplo, evitar que um método do *controller*, cujo retorno não muda muito, seja executado diversas vezes.

Essa estratégia, em que guardamos um resultado e o reproveitamos, é comumente conhecida como **cache** e o Spring também fornece ajuda nessa área. Será preciso adicionar as dependências do Spring Cache no arquivo `pom.xml`:

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

O módulo Spring Cache fornece anotações que facilitam "cachear" dados. Primeiro, precisamos habilitar essas anotações através da anotação `@EnableCaching` na classe principal:

```
@EnableCaching
// outros annotations omitidos
@SpringBootApplication
public class AluraForumApplication {

    // método main omitido
}
```

Vamos, então, guardar os dados de retorno do método para a aplicação cliente montar o *dashboard*. Quando o cliente acessa o endpoint `/api/topics/dashboard`, o Spring MVC executa o método `getDashboardInfo()` da classe `TopicController` que acessa o banco e carrega as informações do *dashboard* (se você analisar o código novamente, notará uma quantidade significativa de *queries* para

executar essa lógica). As informações do *dashboard* raramente mudam a cada acesso. As informações estatísticas até mudam com mais frequência, mas não é uma informação sensível a ponto de impedir a utilização do fórum.

```
@RestController
@RequestMapping("/api/topics")
public class TopicController {

    // atributos e métodos omitidos

    @GetMapping(value = "/dashboard", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<TopicDashboardItemOutputDto> getDashboardInfo() {

        CategoriesAndTheirStatisticsData categoriesStatisticsData =
            this.dashboardDataProcessingService.execute();
        return TopicDashboardItemOutputDto.listFromCategories(categoriesStatisticsData);
    }
}
```

A anotação `@Cacheable` é justamente a anotação que podemos usar para marcar os valores de retorno do método que serão armazenados no *cache*. Apenas colocar a anotação `@Cacheable` em cima do método faz com que o Spring saiba que, uma vez que o código foi executado, ele deve guardar o retorno e reutilizá-lo para todas as próximas invocações do método `getDashboardInfo()`, ou seja, para as próximas requisições para `/api/topics/dashboard`:

```
@Cacheable
@GetMapping(value = "/dashboard", produces = MediaType.APPLICATION_JSON_VALUE)
public List<TopicDashboardItemOutputDto> getDashboardInfo() {

    // código omitido
}
```

Mas quando reiniciamos a aplicação, recebemos a seguinte exceção:

```
*****
APPLICATION FAILED TO START
*****

Description:
A component required a bean named 'cacheManager' that could not be found.

Action:
Consider defining a bean named 'cacheManager' in your configuration.
```

Figura 14.1: exception-cache-manager

O erro diz que um componente `CacheManager` é obrigatório. Para o *cache* funcionar, precisamos de uma implementação de `CacheManager` para armazenar e recuperar os dados que foram "cacheados" já que não há uma implementação padrão definida. Ou seja, após habilitar as operações de *cache*, é necessário registrar um `CacheManager`.

`CacheManager` é uma *interface* do Spring e sua implementação mais simples é a

`ConcurrentMapCacheManager` que constrói instâncias de `ConcurrentMapCache` para cada solicitação de `cache` realizada e possui a função de armazenar e resgatar dados "cacheados".

Vamos, então, avisar o Spring para usar a implementação de `ConcurrentMapCacheManager`:

```
@EnableCaching
@SpringBootApplication
public class AluraForumApplication {

    // método main omitido

    @Bean
    public CacheManager cacheManager() {
        return new ConcurrentMapCacheManager();
    }
}
```

Agora, ao rodar a aplicação, recebemos um novo erro:

```
java.lang.IllegalStateException: No cache could be resolved for 'Builder[public java.util.List br.com.alura.forum.controller.TopicController.getDashboardInfo()] caches=[] | key=' | keyGene
at org.springframework.cache.interceptor.CacheAspectSupport.getCache(CacheAspectSupport.java:255) ~[spring-context-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at org.springframework.cache.interceptor.CacheAspectSupportsCacheOperationContext.<init>(CacheAspectSupport.java:707) ~[spring-context-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at org.springframework.cache.interceptor.CacheAspectSupport.getOperationContext(CacheAspectSupport.java:265) ~[spring-context-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at org.springframework.cache.interceptor.CacheAspectSupportsCacheOperationContexts.<init>(CacheAspectSupport.java:598) ~[spring-context-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at org.springframework.cache.interceptor.CacheAspectSupport.execute(CacheAspectSupport.java:345) ~[spring-context-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at org.springframework.cache.interceptor.CacheInterceptor.invoke(CacheInterceptor.java:61) ~[spring-context-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:186) ~[spring-aop-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at org.springframework.aop.framework.CglibAopProxy$DynamicAdvisedInterceptor.intercept(CglibAopProxy.java:688) ~[spring-aop-5.1.4.RELEASE.jar:5.1.4.RELEASE]
at br.com.alura.forum.controller.TopicController$$EnhancerBySpringCGLBs$9c9cb7dd3.getDashboardInfo(<generated>) ~[classes/:na]
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0_201]
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) ~[na:1.8.0_201]
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ~[na:1.8.0_201]
```

Figura 14.2: cacheName-exception

O erro diz que não conseguiu resolver o `cache` para o retorno do método `getDashboardInfo()`. Isso acontece porque a anotação precisa receber pelo menos um valor (do tipo `String`), que serve como uma identificação para recuperação dos dados no futuro por uma implementação de `CacheResolver`. Um `CacheResolver` também é uma interface do Spring que resolve instâncias de `cache` baseadas da configuração do `CacheManager` e nos nomes retornados pelo método `getCacheNames()` de um `BasicOperation` - interface que deve ser implementada por todas as operações de `cache` e que retorna os nomes de `caches` associados a uma operação específica.

A grosso modo, o Spring exige que definamos nomes que podem agrupar um ou mais objetos que queiramos colocar em `cache` - isso também é conhecido como **região de cache**.

Devemos, portanto, definir um nome para o parâmetro `value` da anotação `@Cacheable` para criar um região de cache com este nome:

```
@Cacheable(value="dashboardData")
@GetMapping(value = "/dashboard", produces = MediaType.APPLICATION_JSON_VALUE)
public List<TopicDashboardItemOutputDto> getDashboardInfo() {

    // código omitido
}
```

A primeira vez que este método for invocado, os dados serão armazenados com o nome `dashboardData` e recuperados através deste mesmo nome em chamadas futuras. Esse comportamento

pode ser notado ao fazer uma primeira requisição para `api/topics/dashboard`. Note que as *queries* utilizadas para trazer os dados do *dashboard* são executadas e mostradas no *console*. Ao executar novamente a requisição para este recurso, nenhuma *query* é executada já que os dados são resgatados do *cache* `dashboardData` que foram armazenados na primeira requisição.

Portanto, métodos com execuções caras (que exigem bastante processamento como muitas idas ao banco de dados) podem ser executados apenas uma vez e seu resultado reusado quando necessário, sem necessidade de executar o método novamente.

Aprenda se divertindo na Alura Start!



Você conhece alguém que tem potencial para tecnologia e programação, mas que nunca escreveu uma linha de código? Pode ser um filho, sobrinho, amigo ou parente distante. Na **Alura Start** ela vai poder criar games, apps, sites e muito mais! É o **começo da jornada com programação e a porta de entrada para uma possível carreira de sucesso**. Ela vai estudar em seu próprio ritmo e com a melhor didática. A qualidade da conceituada Alura, agora para Starters.

[Conheça os cursos online da Alura Start!](#)

14.2 CACHEEVICT

Outro método que pode ter seus valores "cacheados" é o `getTopicDetails()` da classe `TopicController` que retorna os dados de um tópico específico. Podemos usar a mesma estratégia que usamos para os dados do *dashboard*:

```
@RestController
@RequestMapping("/api/topics")
public class TopicController {

    // outros métodos e atributos omitidos

    @Cacheable(value="topicDetails")
    @GetMapping(value = "/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
    public TopicOutputDto getTopicDetails(@PathVariable Long id) {
        // código interno omitido
    }
}
```

Acontece que quando um usuário do fórum da Alura responde um tópico, este deve ser atualizado imediatamente para não afetar a usabilidade da aplicação. Este não é o problema para a maioria dos

tópicos (já que a maioria são muito antigos e já foram solucionados) mas é para os tópicos mais recentes. Como utilizar o *cache* nesses casos?

O Spring Cache provê uma outra anotação para remover os dados "cacheados" caso uma atualização ocorra, a `@CacheEvict`. No caso dos detalhes de um tópico, queremos que a cada nova resposta criada para um tópico específico, o *cache* seja removido da memória. Para garantir este comportamento, devemos usar a anotação no método `answerTopic()` da classe `AnswerController` e usar a mesma chave do cache que queremos remover, ou seja, `topicDetails`:

```
@Controller
@RequestMapping("/api/topics/{topicId}/answers")
public class AnswerController {

    @CacheEvict(value="topicDetails")
    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_V
ALUE)
    public ResponseEntity<AnswerOutputDto> answerTopic(@PathVariable Long topicId,
        @Valid @RequestBody NewAnswerInputDto newAnswerDto,
        @AuthenticationPrincipal User loggedUser,
        UriComponentsBuilder uriBuilder) {
        // código omitido
    }
}
```

Mas, ao testar, vemos que os dados são "cacheados" mas não são atualizados após a criação de uma nova resposta. Ou seja, os dados dessa região de cache não foram removidos. Isso acontece já que as anotações `@Cacheable` e `@CacheEvict` definem, por padrão, uma atributo `key` como segue:

- caso o método não possua nenhuma parâmetro, a `key` é definida com uma `SimpleKey.EMPTY` - um array de `Object` vazio.
- caso o método tenha apenas um parâmetro, a `key` é definida com o valor deste parâmetro.
- caso o método possua mais de um parâmetro, a `key` é definida com uma `SimpleKey` contendo todos os parâmetros em um array de `Object`.

No caso do método `getTopicDetails()`, esse valor é o parâmetro `id` que representa o `id` do tópico. Já o método `answerTopic()` recebe três parâmetros, o `id` do tópico (`topicId`), o usuário logado (`loggedUser`) e o componente para construção de URIs (`uriBuilder`). O valor da chave é concatenado com o valor do parâmetro `value`, ou seja, com o `cacheName`. Já o `getDashboardInfo()`, por não possuir parâmetro, possui uma `SimpleKey` vazia como chave. Portanto, as chaves de cada região de cache são definidas como segue:

- `getTopicDetails() - topicDetails::id`
- `answerTopic() - topicDetails::SimpleKey [topicId, br.com.alura.forum.controller.dto.input.NewAnswerInputDto@33cbb6cc, br.co`

```
m.alura.forum.model.User@3f7a8aa0,org.springframework.web.servlet.support.ServletUriComponentsBuilder@1c9dcd61]
• getDashboardInfo() - dashboardData::SimpleKey []
```

Portanto, no momento de criação do *cache* do tópico de `id=15`, o `cacheName` criado será `topicDetails::15` e quando uma nova resposta é criada para o tópico de `id=15`, o `cacheName` que tentará ser deletado será algo parecido com `topicDetails::SimpleKey [15,br.com.alura.forum.controller.dto.input.NewAnswerInputDto@33ccb6cc,br.com.alura.forum.model.User@3f7a8aa0,org.springframework.web.servlet.support.ServletUriComponentsBuilder@1c9dcd61]`. Então, o cache `topicDetails::15` não será deletado, já que as chaves não batem.

Para garantir que o `cacheName` seja o mesmo para os dois, devemos definir a mesma chave para cada um deles. Queremos que seja usado o parâmetro `id` para o método `getTopicDetails()` e o `topicId` para o método `answerTopic()`. Através da *expression language* do Spring (SpEL), podemos facilmente definir isso:

```
@Cacheable(value="topicDetails", key="#id")
@CacheEvict(value="topicDetails", key="#topicId")
```

O caractere `#` da linguagem de expressão é usado para nomes de argumentos: `#argument_name`. Dessa maneira conseguimos definir o valor da `key` dinamicamente pelo nome do parâmetro. Também é possível passar pela ordem em que eles aparecem usando `#p0`, `#p1`, ... (`parameters`) ou `#a0`, `#a1`, ... (`arguments`).

Agora, quando uma nova resposta é criada em um tópico específico, sua região de *cache* é removida e será atualizada em uma nova chamada para o método `getTopicDetails()` - e obtemos o comportamento desejado. Neste caso, não haveria necessidade de definir uma chave para o método `getTopicDetails()` já que ele recebe apenas o `id`, mas é importante garantir já que esse método pode vir a mudar no futuro e o Spring sugere essa abordagem em sua documentação oficial.

Agora, ao fazer um requisição para `/api/topics/15`, criamos o *cache* com o nome `topicDetails::15`. Ao criar uma nova resposta para este tópico, o cache `topicDetails::15` é removido e será atualizado na próxima chamada de `/api/topics/15`.

PARA SABER MAIS

Caso você queira limpar a região inteira do cache como a `topicDetails` do exemplo que utilizamos anteriormente, basta fazer:

```
@CacheEvict(value="topicDetails", allEntries=true)
```

Neste cenário o Spring vai ignorar qualquer chave especificada e vai limpar a região inteira ao invés de apenas uma entrada.

14.3 EXERCÍCIO: COMEÇANDO COM SPRING CACHE

1. Anote o método `getDashboardInfo()` da classe `TopicController` com `@Cacheable` para guardarmos seu retorno no cache:

```
@RestController
@RequestMapping("/api/topics")
public class TopicController {

    // outros métodos e atributos omitidos

    @Cacheable("dashboardData")
    @GetMapping(value = "/dashboard", produces = MediaType.APPLICATION_JSON_VALUE)
    public List<TopicDashboardItemOutputDto> getDashboardInfo() {

        CategoriesAndTheirStatisticsData categoriesStatisticsData =
            this.dashboardDataProcessingService.execute();
        return TopicDashboardItemOutputDto.listFromCategories(categoriesStatisticsData);
    }

    // outros métodos omitidos
}
```

2. Agora precisamos habilitar o cache no Spring anotando a classe `AluraForumApplication` com `@EnableCaching`:

```
@EnableCaching
// outras annotations omitidas
@SpringBootApplication
public class AluraForumApplication {

    // método main omitido
}
```

Adicione também o método `cacheManager()`, que provê o *bean* do `CacheManager` padrão que será usado para gerenciar os dados armazenados em cache.

```
@EnableCaching
// outras annotations omitidas
@SpringBootApplication
public class AluraForumApplication {
```

```

// método main omitido

@Bean
public CacheManager cacheManager() {
    return new ConcurrentMapCacheManager();
}

}

```

- Rode a aplicação e faça uma chamada para o *endpoint* do Dashboard. Perceba que numa primeira chamada tudo ocorre normalmente, o *controller* chama a classe de serviço que completa o trabalho buscando as informações no banco de dados (observe os logs do Hibernate no console) e montando os objetos que representam os dados estatísticos.

Agora, faça uma segunda chamada. Perceba como a performance foi melhorada. A aplicação se utilizou dos dados armazenados no cache para evitar todo o processamento desnecessário. Novamente, observe os logs do Hibernate, e perceba que nada foi executado no banco.

- Agora que o cache está funcionando, vamos implementar também o cache para o *endpoint* `/api/topics/{id}`, que traz as informações detalhadas de um tópico específico.

Anote com `@Cacheable` o método `getTopicDetails()` da classe `TopicController`.

```

@RestController
@RequestMapping("/api/topics")
public class TopicController {

    // outros métodos e atributos omitidos

    @Cacheable(value="topicDetails", key="#id")
    @GetMapping(value = "/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
    public TopicOutputDto getTopicDetails(@PathVariable Long id) {

        // código interno omitido
    }
}

```

A propriedade `key` define a chave a ser usada para o cache com base nas informações do método. Utilizamos o Spring Expression Language (SpEL), para inferir a chave. No nosso caso, usamos o parâmetro `id` para compor a chave.

- Rode novamente a aplicação, e dessa vez faça uma chamada ao *endpoint* de detalhes de um tópico. Perceba que em novas chamadas também consumimos as informações presentes no cache.
- Adicione agora uma nova resposta ao tópico que você usou pra consultar os detalhes. Em seguida, acesse novamente os detalhes do tópico e veja o que é apresentado. Perceba que a nova resposta não está presente nas informações detalhadas, dado que o cache não foi atualizado.

Adicione a anotação `@CacheEvict` ao método `answerTopic()` da classe `AnswerController`, com o mesmo *cache name*, `topicDetails`.

```
@Controller
```

```

@RequestMapping("/api/topics/{topicId}/answers")
public class AnswerController {

    @CacheEvict(value="topicDetails", key="#topicId")
    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE, produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<AnswerOutputDto> answerTopic(@PathVariable Long topicId,
        @Valid @RequestBody NewAnswerInputDto newAnswerDto,
        @AuthenticationPrincipal User loggedUser,
        UriComponentsBuilder uriBuilder) {

        // código omitido

    }
}

```

Quando uma nova resposta for inserida a um tópico, a chave exata referente ao tópico será removida da região de cache, fazendo com que na próxima chamada aos detalhes, o processamento ocorra normalmente, atualizando o cache ao final.

7. Teste novamente a adição de uma nova resposta a um tópico e perceba que, desta vez, ao consultar os detalhes do tópicos as informações estão atualizadas.

14.4 REDIS

O Spring já oferece uma integração automática, bastando adicionar o `spring-boot-starter-data-redis` como dependência no arquivo `pom.xml`.

O Redis é um armazenamento de chave-valor de estruturas de dados em memória e é um acrônimo para *Remote Dictionary Server* (Servidor de Dicionário Remoto). Dentre as vantagens de utilizá-lo para *cache* é que todos seus dados residem na memória principal do servidor (diferente da maioria dos sistemas de banco de dados que armazenam dados em disco) e aceita armazenamento de qualquer tipo de dado. Isso evita perda de tempo com buscas e cria um *cache* com um bom desempenho diminuindo a latência de acesso.

Além disso, o Redis suporta operações com múltiplas chaves, transações e implementação de expiração de dados de maneira fácil. Para mais informações sugerimos acessar a [documentação oficial](#) da ferramenta. Para a instalação, siga este [link](#).

Seus livros de tecnologia parecem do século passado?



Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.

Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.](#)

14.5 CONFIGURANDO O REDIS NO PROJETO

Primeiro, vamos remover a implementação atual que estamos utilizando de `CacheManager` já que o Spring Boot auto configura um `RedisCacheManager` bastando apenas a declaração do `starter` `spring-boot-starter-data-redis` no arquivo `pom.xml` :

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

Depois de instalado, abra o terminal e digite o comando `redis-server` para iniciar o servidor do Redis. Outros comandos úteis estão disponíveis [neste link](#). Por padrão, o Redis roda na porta 6379. Caso você tenha configurado de maneira diferente (como rodar o Redis em uma porta diferente), será preciso avisar isso ao Spring Boot no arquivo `application.properties` :

```
# cache properties
spring.cache.type=redis
spring.redis.host=localhost
spring.redis.port=6379
spring.cache.redis.cache-null-values=false
# time to live in milliseconds (8hs)
spring.cache.redis.time-to-live=28800000
```

Essas são algumas propriedade que podemos customizar do Redis, caso necessário. Caso você receba uma exceção de conexão recusada, revise se as propriedades foram definidas corretamente.

Com o Redis rodando e configurado, ao fazer uma requisição para `/api/topics/dashboard` recebemos o seguinte erro:

```

java.io.NotSerializableException: br.com.alura.forum.controller.dto.output.TopicDashboardItemOutputDto
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184) ~[na:1.8.0_181]
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348) ~[na:1.8.0_181]
    at java.util.ArrayList.writeObject(ArrayList.java:766) ~[na:1.8.0_181]
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0_181]
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) ~[na:1.8.0_181]
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ~[na:1.8.0_181]
    at java.lang.reflect.Method.invoke(Method.java:498) ~[na:1.8.0_181]
    at java.io.ObjectStreamClass.invokeWriteObject(ObjectStreamClass.java:1140) ~[na:1.8.0_181]
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1496) ~[na:1.8.0_181]
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1432) ~[na:1.8.0_181]
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1178) ~[na:1.8.0_181]
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348) ~[na:1.8.0_181]
    at org.springframework.core.serializer.DefaultSerializer.serialize(DefaultSerializer.java:46) ~[spring-core-5.1
    at org.springframework.core.serializer.support.SerializationConverter.convert(SerializationConverter.java:63) ~[spring-core-5.1

```

Figura 14.3: serialize-error

O erro diz que a classe `TopicDashboardItemOutputDto` não pode ser serializada. Isso acontece já que o Redis, antes de persistir o objeto no *cache*, precisa serializar o objeto para um *array* de *bytes*. Vamos então fazer com que a classe `TopicDashboardItemOutputDto` implemente a interface `Serializable`:

```

public class TopicDashboardItemOutputDto implements Serializable {

    // código omitido
}

```

Agora, ao rodar a aplicação e fazer uma requisição para `/api/topics/dashboard` vemos que o *cache* para os items do *dashboard* continua funcionando, mas agora com o Redis. Vamos fazer o mesmo para o `topicDetails`. Precisamos garantir que os objetos de retorno dos métodos `getTopicDetails()` e `answerTopic()` implementem a interface `Serializable`:

```

public class TopicOutputDto implements Serializable {
    // código omitido
}

public class AnswerOutputDto implements Serializable {
    // código omitido
}

```

Antes de testar, abra o terminal e digite o comando `redis-cli monitor` que abre o console de monitoramento do Redis. Ao fazer uma requisição para `/api/topics/1` no navegador, vemos a seguinte saída no terminal:

```
1556121605.350878 [0 127.0.0.1:57092] "GET" "topicDetails::1"
1556121605.392480 [0 127.0.0.1:57092] "SET" "topicDetails::1" "\xac\xed\x00\x05s
r\x007br.com.alura.forum.controller.dto.output.TopicOutputDto0\|xa1\|x11\x96\x9f%
0\x02\x00\x0cI\x00\x11numberOfResponsesL\x00\|aanswerst\x00\x10Ljava/util/List;L\
\x00\x0ccategoryNameL\x00\x12Ljava/lang/String;L\x00\|acontentq\x00~\|x00\x02L\x00\
ncourseNameq\x00~\|x00\x02L\x00\x0fcreationInstantt\x00\x13Ljava/time/Instant;L\x
00\x02idt\x00\x10Ljava/lang/Long;L\x00\|nlastUpdateq\x00~\|x00\x03L\x00\|townerName
q\x00~\|x00\x02L\x00\x10shortDescriptionq\x00~\|x00\x02L\x00\x06statust\x003Lbr/co
m/alura/forum/model/topic/domain/TopicStatus;L\x00\x0fsubcategoryNameq\x00~\|x00\
x02xp\x00\x00\x00sr\x00\x13java.util.ArrayList\x81\xd2\x1d\x99\xc7a\x9d\x03
\x00\x01I\x00\x04sizepx\x00\x00\x00w\x04\x00\x00\x00\x00xt\x00\rPrograma\xc3
\xc7\xc3\xc3opt\x00\x1dJava e Orienta\xc3\xc7\xc3\xc3o a Objetosr\x00\rjava.tim
e.Ser\x95]\x84\xba\x1b\|H\xb2\x0c\x00\x00xpw\r\x02\x00\x00\x00\x00[RD\xb5\x00\x0
0\x00\x00xsrx\x00\x0ejava.lang.Long;\x8b\xe4\x90\xcc\x8f#\xfdf\x02\x00\x01J\x00\x0
5valuexr\x00\x10java.lang.Number\x86\xac\x95\x1d\x0b\x94\xe0\x8b\x02\x00\x00xp\x
00\x00\x00\x00\x00\x00\x01sq\x00~\|x00\x0bw\r\x02\x00\x00\x00\x00[RD\xb5\x00\
\x00\x00\x00xt\x00\x0cRafael Rollot\x00\x17Como mapear uma Servlet~r\x001br.com.a
lura.forum.model.topic.domain.TopicStatus\x00\x00\x00\x00\x00\x00\x00\x12\x0
0\x00xr\x00\x0ejava.lang.Enum\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x12\x00\x00xpt\x00
\nNOT_SOLVEDt\x00\x04Java"
```

Repare que o Redis tentou acessar o cache via `GET topicDetails::1`, como não encontrou, ele persistiu os detalhes do tópico de `id=1` através do comando `SET topicDetails::1`. Ao adicionar uma resposta ao tópico de `id=1`, a saída será:

```
\x00xr\x00\x0ejava.lang.Enum\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x12\x00\x00xpt\x00
\nNOT_SOLVEDt\x00\x04Java"
1556121852.501557 [0 127.0.0.1:57092] "DEL" "topicDetails::1"
```

Veja que o comando executado foi `DEL topicDetails::1`, ou seja, o *cache* foi limpo já que o método para adição de uma nova resposta utiliza a anotação `@CacheEvict` para garantir esse comportamento. E quando acessamos novamente `/api/topics/1` um novo cache `topicDetails::1` é criado com o objeto atualizado com a nova resposta.

Para saber todas as chaves criadas no Redis, basta acessar o console do Redis pelo comando `redis-cli` e em seguida digitar `KEYS *`:

```
127.0.0.1:6379> KEYS *
1) "dashboardData2::SimpleKey []"
2) "topicDetails::1"
127.0.0.1:6379>
```

EXERCÍCIOS: INTEGRANDO SPRING CACHE E REDIS

1. Adicione a dependência do Spring Cache no arquivo `pom.xml`:

```

<dependencies>

    <!-- dependência do cache -->

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-redis</artifactId>
    </dependency>

<dependencies>

```

2. Remova a implementação do CacheManager usada para testar a abstração de cache do Spring.

```

@EnableCaching
// outras annotations omitidas
@SpringBootApplication
public class AluraForumApplication {

    // método main omitido

    // REMOVER ESTE BEAN
    //@Bean
    //public CacheManager cacheManager() {
    //    return new ConcurrentMapCacheManager();
    //}
}

```

PS: Segundo a própria documentação do Spring, não é sugerido o uso de um ConcurrentMapCacheManager em ambientes de produção.

3. Com a remoção do cache manager simples, o Spring automaticamente vai prover a implementação de CacheManager da configuração do Redis, o RedisCacheManager .

Sendo assim rode a aplicação, e teste a funcionalidade do dashboard. Veja o que acontece!

```

java.net.ConnectException: Connection refused
at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method) ~[na:1.8.0_201]
at sun.nio.ch.SocketChannelImpl.finishConnect(SocketChannelImpl.java:717) ~[na:1.8.0_201]
at io.netty.channel.socket.nio.NioSocketChannel.doFinishConnect(NioSocketChannel.java:327) ~[netty-transport-4.1.34.Final.jar:4.1.34.Final]
at io.netty.channel.nio.AbstractNioChannel$AbstractNioUnsafe.finishConnect(AbstractNioChannel.java:340) ~[netty-transport-4.1.34.Final.jar:4.1.34.Final]
at io.netty.channel.nio.NioEventLoop.processSelectedKey(NioEventLoop.java:665) ~[netty-transport-4.1.34.Final.jar:4.1.34.Final]
at io.netty.channel.nio.NioEventLoop.processSelectedKeysOptimized(NioEventLoop.java:612) ~[netty-transport-4.1.34.Final.jar:4.1.34.Final]
at io.netty.channel.nio.NioEventLoop.processSelectedKeys(NioEventLoop.java:529) ~[netty-transport-4.1.34.Final.jar:4.1.34.Final]
at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:491) ~[netty-transport-4.1.34.Final.jar:4.1.34.Final]
at io.netty.util.concurrent.SingleThreadEventExecutor$5.run(SingleThreadEventExecutor.java:905) ~[netty-common-4.1.34.Final.jar:4.1.34.Final]
at io.netty.util.concurrent.FastThreadLocalRunnable.run(FastThreadLocalRunnable.java:30) ~[netty-common-4.1.34.Final.jar:4.1.34.Final]
at java.lang.Thread.run(Thread.java:748) [na:1.8.0_201]

```

Figura 14.7: Error Redis Connection

Se o erro acima não aparecer, siga para o próximo exercício. Caso o erro acima apareça, algo não saiu como esperado e não foi possível se conectar com o Redis . Por padrão, a configuração de cache usará uma instância Redis acessível através do localhost na porta 6379.

Opcionalmente o apontamento para o serviço do Redis é configurável a partir do arquivo application.properties , onde é possível ainda definir configurações para um ajuste mais fino

sobre a gestão do *cache*, como no exemplo abaixo:

```
# cache properties
spring.cache.type=redis
spring.redis.host=localhost
spring.redis.port=6379
spring.cache.redis.cache-null-values=false
# time to live in milliseconds (8hs)
spring.cache.redis.time-to-live=28800000
```

4. No terminal, execute o comando `redis-server /etc/redis/redis.conf` para iniciar o servidor do Redis. Execute novamente o acesso ao *endpoint* do dashboard.

Veja que agora recebemos outro erro:

```
java.io.NotSerializableException: br.com.alura.forum.controller.dto.output.TopicDashboardItemOutputDto
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184) ~[na:1.8.0_181]
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348) ~[na:1.8.0_181]
    at java.util.ArrayList.writeObject(ArrayList.java:766) ~[na:1.8.0_181]
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0_181]
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) ~[na:1.8.0_181]
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ~[na:1.8.0_181]
    at java.lang.reflect.Method.invoke(Method.java:498) ~[na:1.8.0_181]
    at java.io.ObjectStreamClass.invokeWriteObject(ObjectStreamClass.java:1140) ~[na:1.8.0_181]
    at java.io.ObjectOutputStream.writeSerialData(ObjectOutputStream.java:1496) ~[na:1.8.0_181]
    at java.io.ObjectOutputStream.writeOrdinaryObject(ObjectOutputStream.java:1432) ~[na:1.8.0_181]
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1178) ~[na:1.8.0_181]
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348) ~[na:1.8.0_181]
    at org.springframework.core.serializer.DefaultSerializer.serialize(DefaultSerializer.java:46) ~[spring-core-5.1.10.RELEASE.jar:5.1.10.RELEASE]
    at org.springframework.core.serializer.support.SerializingConverter.convert(SerializingConverter.java:63) ~[spring-core-5.1.10.RELEASE.jar:5.1.10.RELEASE]
```

Figura 14.8: Error Redis Serialization

O Redis não consegue *serializar* os objetos que serão inseridos no cache.

5. Faça com que as classes `TopicDashboardItemOutputDto` , `TopicOutputDto` e `AnswerOutputDto` implementem a interface `Serializable` , do pacote `java.io` :

```
public class TopicDashboardItemOutputDto implements Serializable {
    // código omitido
}

public class TopicOutputDto implements Serializable {
    // código omitido
}

public class AnswerOutputDto implements Serializable {
    // código omitido
}
```

Aproveite para gerar o *serialization id* nessas classes. Use o recurso do Eclipse para isso: *CTRL + 1 > Add default serial version ID*.

6. Teste novamente o acesso ao *endpoint* do dashboard e veja que tudo funciona. Acesse mais de uma vez e verifique quantas vezes o código do *controller* está sendo executado.

Opcionalmente, para acompanhar a execução diretamente no Redis, execute o comando `redis-cli monitor` no terminal e faça novamente uma requisição para o *endpoint* do dashboard. Assim será possível acompanhar o fluxo de acesso ao cache.

7. Acesse o detalhe de um tópico e veja que o cache continua funcionando.

Acrescente uma resposta ao tópico e, em seguida, acesse novamente os detalhes deste tópico e veja que o cache é removido e atualizado. Para acompanhar a execução no Redis lembre-se que é possível utilizar o comando `redis-cli monitor` no terminal.

8. (Opcional) Construa um *endpoint* para limpar o cache da aplicação sabendo que a classe `CacheManager` do Spring possui um método chamado `getCacheNames()`, que retorna os nomes de todos os caches conhecidos pelo manager na aplicação.

Faça a busca de cada cache pelo seu nome e utilize o método `clear()` para limpá-los.

```
@RestController
@RequestMapping("/admin/cache")
public class CacheController {

    @Autowired
    private CacheManager manager;

    @GetMapping("/clear")
    public String clearCache(){
        manager.getCacheNames().stream()
            .forEach(name -> manager.getCache(name).clear());
        return "cache cleaned";
    }
}
```

APÊNDICE: MAIS SEGURANÇA COM SPRING SECURITY ACL

15.1 EXERCÍCIO: MARCANDO RESPOSTAS COMO SOLUÇÃO

- Nossa *app* deve ser capaz de, na página de detalhes de um tópico, marcar uma resposta como a solução para o mesmo. Portanto, na classe `AnswerController`, adicione o método `markAsSolution()`, com o devido mapeamento de *request*.

```
@Controller
@RequestMapping("/api/topics/{topicId}/answers")
public class AnswerController {

    // atributos omitidos

    // método answerTopic() omitido

    @PostMapping("/{answerId}/solution")
    public ResponseEntity<?> markAsSolution(@PathVariable Long topicId,
                                              @PathVariable Long answerId) {

        return ResponseEntity.ok().build();
    }
}
```

- Agora que já temos a base necessária, podemos prosseguir com nosso desenvolvimento. Com o id da resposta marcada disponível, altere a implementação para carregar a resposta do banco de dados, e marcá-la como solução para o tópico.

```
@Controller
@RequestMapping("/api/topics/{topicId}/answers")
public class AnswerController {

    // atributos omitidos

    // método answerTopic() omitido

    @PostMapping("/{answerId}/solution")
    public ResponseEntity<?> markAsSolution(@PathVariable Long topicId,
                                              @PathVariable Long answerId) {

        Answer answer = this.answerRepository.findById(answerId);
        answer.markAsSolution();

        return ResponseEntity.ok().build();
    }
}
```

3. Para que as alterações na resposta sejam propagadas ao banco de dados e mantidas para a aplicação, anote o método `markAsSolution()` com `@Transactional`.

```
@Transactional  
@PostMapping("/{answerId}/solution")  
public ResponseEntity<?> markAsSolution(@PathVariable Long topicId,  
                                         @PathVariable Long answerId) {  
  
    // implementação omitida ...  
  
}
```

OBS: Com `@Transactional` sobre a action do controller, o contexto transacional será iniciado ao receber o request e fechado ao final do processamento do mesmo. Assim, ao terminar o processamento com sucesso, o estado dos objetos gerenciados pela JPA será refletido no banco de dados. Caso o processamento não ocorra como o esperado qualquer alteração não é efetivada.

4. Utilize a classe `UriComponentsBuilder` para construir o endereço do recurso que deve ser enviado ao cliente na resposta.

```
@Controller  
@RequestMapping("/api/topics/{topicId}/answers")  
public class AnswerController {  
  
    // atributos omitidos  
  
    // método answerTopic() omitido  
  
    @Transactional  
    @PostMapping("/{answerId}/solution")  
    public ResponseEntity<?> markAsSolution(@PathVariable Long topicId,  
                                         @PathVariable Long answerId, UriComponentsBuilder uriBuilder) {  
  
        Answer answer = this.answerRepository.findById(answerId);  
        answer.markAsSolution();  
  
        URI path = uriBuilder  
            .path("/api/topics/{topicId}/solution")  
            .buildAndExpand(topicId)  
            .toUri();  
  
        return ResponseEntity.created(path)  
            .body(new AnswerOutputDto(answer));  
    }  
}
```

5. A funcionalidade de marcação de solução deve ser acionada essencialmente, ou pelo **usuário que registrou o tópico**, ou um **usuário admin** qualquer.

Sendo assim, altere a implementação para verificar se o usuário logado tem direitos sobre a execução dessa ação sobre este tópico específico.

```
@Transactional  
@PostMapping("/{answerId}/solution")  
public ResponseEntity<?> markAsSolution(@PathVariable Long topicId,  
                                         @PathVariable Long answerId, UriComponentsBuilder uriBuilder,
```

```

    @AuthenticationPrincipal User loggedUser) {

    Topic topic = this.topicRepository.findById(topicId);
    if(loggedUser.isOwnerOf(topic) || loggedUser.isAdmin()) {

        Answer answer = this.answerRepository.findById(answerId);
        answer.markAsSolution();

        URI path = uriBuilder
            .path("/api/topics/{topicId}/solution")
            .buildAndExpand(topicId)
            .toUri();

        return ResponseEntity.created(path)
            .body(new AnswerOutputDto(answer));
    }

    return ResponseEntity.status(HttpStatus.FORBIDDEN)
        .body("Você não tem direito a acessar este recurso!");
}

```

Adicione também os métodos `isOwnerOf(topic)` e `isAdmin()` à classe `User`.

```

@Entity
public class User implements UserDetails {

    // atributos e métodos anteriores omitidos

    public boolean isOwnerOf(Topic topic) {
        return this.equals(topic.getOwner());
    }

    public boolean isAdmin() {
        return this.authorities.stream()
            .filter(role -> role.equals(Role.ROLE_ADMIN))
            .findFirst().isPresent();
    }
}

```

6. Agora, para que seja possível a aplicação cliente receber as informações atualizadas - com a resposta identificada como solução - ao acessar os detalhes do tópico, adicione a anotação para evitar o cache que mantém as informações sobre o referido tópico.

```

@Transactional
@CacheEvict(value = "topicDetails", key = "#topicId")
@PostMapping("/{answerId}/solution")
public ResponseEntity<?> markAsSolution(@PathVariable Long topicId,
    @PathVariable Long answerId, UriComponentsBuilder uriBuilder,
    @AuthenticationPrincipal User loggedUser) {

    // implementação omitida
}

```

7. Rode novamente a aplicação e teste a marcação de uma resposta como solução para um tópico.

Veja que tudo funciona como deveria. Caso o usuário logado seja o *owner* do tópico ou qualquer *admin* a solução é registrada, caso seja qualquer outro usuário, o acesso ao recurso é negado.

8. (Opcional) Para que seja possível testar a marcação de solução a partir da aplicação cliente, no terminal, acesse a pasta do projeto da aplicação cliente `Desktop/fj27-alura-forum-client/`, e execute o comando `git checkout 13SolucionandoFEchandoTopicosACL`, para atualizar seu estado.

Com o *client* e a *API* rodando com os novos recursos, teste novamente o funcionamento da app cliente recarregando a página (`http://localhost:3000/`) e veja que como tudo funciona.

Agora é a melhor hora de respirar mais tecnologia!



Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

[Conheça a Alura Cursos Online.](#)

15.2 EXERCÍCIO: FECHANDO TÓPICOS E AUMENTANDO A SEGURANÇA SOBRE AS INFORMAÇÕES

1. Assim como podemos marcar soluções aos tópicos, devemos também ser capazes de fechar tópicos indefinidamente. Tanto o usuário dono do tópico pode retirá-lo, como um usuário administrador atuando na moderação pode fazê-lo.

Na classe `TopicController`, adicione o método `closeTopic()`, implementando a funcionalidade de fechamento de tópicos.

```
@RestController
@RequestMapping("/api/topics")
public class TopicController {

    // atributos omitidos

    // métodos anteriores omitidos

    @Transactional
    @PostMapping(value = "/{id}/close")
    @CacheEvict(value = "topicDetails", key = "#id")
    public ResponseEntity<Void> closeTopic(@PathVariable Long id,
        UriComponentsBuilder uriBuilder) {

        Topic topic = this.topicRepository.findById(id);
        topic.close();
    }
}
```

```

        URI path = uriBuilder.path("/api/topics/{id}/closed")
            .buildAndExpand(topic.getId())
            .toUri();

        return ResponseEntity.noContent().location(path).build();
    }
}

```

2. Agora que já temos a funcionalidade implementada, lembre-se que somente o *owner* de um determinado tópico ou usuários *admin* podem utilizá-la.

Aqui, poderíamos novamente recorrer aos *if* s para validar se o usuário logado tem direitos sobre este tópico. Mas, para evitar tantas repetições em cenários parecidos com este e adicionar um nível a mais de proteção, utilizaremos o projeto Spring Security ACL.

Primeiro, adicione as dependências necessárias para a utilização do projeto.

```

<dependencies>
    <!-- dependências anteriores omitidas -->

    <!-- dependencia do spring-boot-starter-actuator -->

    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-acl</artifactId>
    </dependency>

    <dependency>
        <groupId>net.sf.ehcache</groupId>
        <artifactId>ehcache</artifactId>
    </dependency>
</dependencies>

```

3. Com as dependências já disponíveis, no pacote `br.com.alura.forum.configuration`, adicione a classe de configuração `AclConfiguration`. Ela é fundamental para que sejam registrados todos os *beans* necessários para utilizar o ACL do Spring Security.

```

@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true)
public class AclConfiguration {

    @Autowired
    private DataSource dataSource;

    @Bean
    public MethodSecurityExpressionHandler defaultMethodSecurityExpressionHandler(
        AclService aclService) {

        DefaultMethodSecurityExpressionHandler expressionHandler
            = new DefaultMethodSecurityExpressionHandler();

        AclPermissionEvaluator permissionEvaluator
            = new AclPermissionEvaluator(aclService);

        expressionHandler.setPermissionEvaluator(permissionEvaluator);
        return expressionHandler;
    }
}

```

```

@Bean
public MutableAclService aclService(LookupStrategy lookupStrategy,
    AclCache aclCache) {

    JdbcMutableAclService aclService = new JdbcMutableAclService(
        this.dataSource, lookupStrategy, aclCache
    );
    aclService.setClassIdentityQuery("SELECT @@IDENTITY");
    aclService.setSidIdentityQuery("SELECT @@IDENTITY");
    return aclService;
}

@Bean
public AclAuthorizationStrategy aclAuthorizationStrategy() {
    return new AclAuthorizationStrategyImpl(
        new SimpleGrantedAuthority("ROLE_ADMIN")
    );
}

@Bean
public PermissionGrantingStrategy permissionGrantingStrategy() {
    return new DefaultPermissionGrantingStrategy(
        new ConsoleAuditLogger()
    );
}

@Bean
public EhCacheBasedAclCache aclCache(EhCacheFactoryBean ehCacheFactoryBean,
    PermissionGrantingStrategy permissionGrantingStrategy,
    AclAuthorizationStrategy aclAuthorizationStrategy) {

    Ehcache ehCache = ehCacheFactoryBean.getObject();

    return new EhCacheBasedAclCache(ehCache,
        permissionGrantingStrategy, aclAuthorizationStrategy);
}

@Bean
public EhCacheFactoryBean aclEhCacheFactoryBean(
    EhCacheManagerFactoryBean aclCacheManagerFactory) {

    CacheManager aclCacheManager = aclCacheManagerFactory.getObject();

    EhCacheFactoryBean ehCacheFactoryBean = new EhCacheFactoryBean();
    ehCacheFactoryBean.setCacheManager(aclCacheManager);
    ehCacheFactoryBean.setCacheName("aclCache");
    return ehCacheFactoryBean;
}

@Bean
public EhCacheManagerFactoryBean aclCacheManager() {
    return new EhCacheManagerFactoryBean();
}

@Bean
public LookupStrategy lookupStrategy(AclCache aclCache,
    AclAuthorizationStrategy aclAuthorizationStrategy) {

    ConsoleAuditLogger auditLogger = new ConsoleAuditLogger();

    return new BasicLookupStrategy(this.dataSource, aclCache,
        aclAuthorizationStrategy, auditLogger);
}

```

```
}
```

4. Com os *beans* devidamente configurados, podemos então inserir novas entradas nas Listas de Controle de Acesso (ACL) de objetos de domínio. Podemos, por exemplo, ao inserir um tópico, registrar que apenas o *owner* e os *admins* podem executar tarefas de administração sobre ele.

No método `createTopic()`, de `TopicController`, altere a implementação para adicionar a chamada ao serviço que registra as entradas na ACL.

```
@RestController
@RequestMapping("/api/topics")
public class TopicController {

    // dependências anteriores omitidas

    @Autowired
    private TopicAclPermissionsRecorderService aclPermissionsRecorder;

    // outros métodos omitidos

    @Transactional
    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
                  produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<TopicOutputDto> createTopic(
        @RequestBody @Valid NewTopicInputDto newTopicDto,
        @AuthenticationPrincipal User loggedUser,
        UriComponentsBuilder uriBuilder) {

        Topic topic = newTopicDto.build(loggedUser, this.courseRepository);
        this.topicRepository.save(topic);

        // adicionando permissões na ACL
        aclPermissionsRecorder.addPermissions(loggedUser, topic,
                                              BasePermission.ADMINISTRATION);

        URI path = uriBuilder.path("/api/topics/{id}")
            .buildAndExpand(topic.getId()).toUri();

        return ResponseEntity.created(path)
            .body(new TopicOutputDto(topic));
    }

    // demais métodos omitidos
}
```

5. Nossa código ainda não funciona, dado que não escrevemos a classe de serviço que pretendemos utilizar.

No pacote `br.com.alura.forum.service.infra`, crie a classe `TopicAclPermissionsRecorderService`, implementando o registro das permissões.

```
@Service
public class TopicAclPermissionsRecorderService {

    @Autowired
    private MutableAclService aclService;

    public void addPermissions(UserDetails principal,
```

```

        Topic topic, Permission... permissions) {

    ObjectIdentity identity = new ObjectIdentityImpl(topic);
    MutableAcl acl = aclService.createAcl(identity);

    PrincipalSid principalSid = new PrincipalSid(principal.getUsername());

    for (Permission permission : permissions) {
        acl.insertAce(acl.getEntries().size(), permission,
                     principalSid, true);
        enterPermissionForAdmins(acl, permission);
    }
    aclService.updateAcl(acl);
}

private void enterPermissionForAdmins(MutableAcl acl, Permission permission) {
    GrantedAuthoritySid adminsSid = new GrantedAuthoritySid(Role.ROLE_ADMIN);
    acl.insertAce(acl.getEntries().size(), permission, adminsSid, true);
}
}

```

6. Agora que temos nosso código compilando normalmente, podemos seguir com os ajustes necessários às implementações de fechamento e solução de tópicos.

De volta ao método `closeTopic()`, da classe `TopicController`, adicione a anotação `@PreAuthorize`, do Spring Security, para inferir à necessidade de autorizar o acesso com base nas entradas de permissões na ACL do tópico.

```

@Transactional
@PostMapping(value = "/{id}/close")
@CacheEvict(value = "topicDetails", key = "#id")
@PreAuthorize(
    "hasPermission(#id, 'br.com.alura.forum.model.topic.domain.Topic', 'ADMINISTRATION')"
)
public ResponseEntity<Void> closeTopic(@PathVariable Long id,
                                         UriComponentsBuilder uriBuilder) {

    // implementação omitida
}

```

PS: Perceba que não foi necessário adicionar nenhuma verificação (if) mais complexa para autorizar o acesso à funcionalidade no contexto de um tópico específico. O ACL já verifica se o acesso é possível em função da lista de controle de acesso (ACL) do tópico.

7. Com o suporte do Spring Secutiry ACL, a implementação do método `markAsSolution()`, da classe `AnswerController`, também pode ser refatorada.

Altera a implementação para usar a autorização pela ACL, ao invés de escrever toda a verificação no controller.

```

@Transactional
@CacheEvict(value = "topicDetails", key = "#topicId")
@PostMapping("/{answerId}/solution")
@PreAuthorize(
    "hasPermission(#topicId, 'br.com.alura.forum.model.topic.domain.Topic', 'ADMINISTRATION')"
)

```

```

)
public ResponseEntity<?> markAsSolution(@PathVariable Long topicId,
    @PathVariable Long answerId, UriComponentsBuilder uriBuilder,
    @AuthenticationPrincipal User loggedUser) {

    // nenhum if é mais necessário

    Answer answer = this.answerRepository.findById(answerId);
    answer.markAsSolution();

    URI path = uriBuilder
        .path("/api/topics/{topicId}/solution")
        .buildAndExpand(topicId)
        .toUri();

    return ResponseEntity.created(path)
        .body(new AnswerOutputDto(answer));
}

```

PS: Em casos onde a autorização não é concedida, o Spring Security ACL lança uma AccessDeniedException , e nossa resposta já será gerada de forma adequada ao cliente.

8. (Opcional) Para que seja possível testar as novas funcionalidades a partir da aplicação cliente, no terminal, acesse a pasta do projeto da aplicação cliente Desktop/fj27-alura-forum-client/ , e execute o comando git checkout 13SolucionandoEFechandoTopicosACL , para atualizar seu estado.

Com o *client* e a *API* rodando com os novos recursos, teste novamente o funcionamento da app cliente recarregando a página (<http://localhost:3000/>) e veja que como tudo funciona.

15.3 PARA SABER MAIS: SPRING SECURITY ACL SQL SCHEMA

Para que seja possível o gerenciamento da Lista de Controle de Acesso (ACL) dos seus objetos de domínio por parte do Spring Security ACL, é necessário que sejam geradas algumas tabelas específicas na base de dados da aplicação.

O schema necessário é definido na própria documentação do projeto, e o .jar do Spring Security ACL já traz os arquivos .sql com a definição das tabelas para cada banco de dados suportado.

Exemplo: `createAclSchemaMySQL.sql`

```

-- ACL Schema SQL for MySQL 5.5+ / MariaDB equivalent

-- drop table acl_entry;
-- drop table acl_object_identity;
-- drop table acl_class;
-- drop table acl_sid;

CREATE TABLE acl_sid (

```

```

    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    principal BOOLEAN NOT NULL,
    sid VARCHAR(100) NOT NULL,
    UNIQUE KEY unique_acl_sid (sid, principal)
) ENGINE=InnoDB;

CREATE TABLE acl_class (
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    class VARCHAR(100) NOT NULL,
    UNIQUE KEY uk_acl_class (class)
) ENGINE=InnoDB;

CREATE TABLE acl_object_identity (
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    object_id_class BIGINT UNSIGNED NOT NULL,
    object_id_identity VARCHAR(36) NOT NULL,
    parent_object BIGINT UNSIGNED,
    owner_sid BIGINT UNSIGNED,
    entries_inheriting BOOLEAN NOT NULL,
    UNIQUE KEY uk_acl_object_identity (object_id_class, object_id_identity),
    CONSTRAINT fk_acl_object_identity_parent FOREIGN KEY (parent_object) REFERENCES acl_object_id
entity (id),
    CONSTRAINT fk_acl_object_identity_class FOREIGN KEY (object_id_class) REFERENCES acl_class (i
d),
    CONSTRAINT fk_acl_object_identity_owner FOREIGN KEY (owner_sid) REFERENCES acl_sid (id)
) ENGINE=InnoDB;

CREATE TABLE acl_entry (
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    acl_object_identity BIGINT UNSIGNED NOT NULL,
    ace_order INTEGER NOT NULL,
    sid BIGINT UNSIGNED NOT NULL,
    mask INTEGER UNSIGNED NOT NULL,
    granting BOOLEAN NOT NULL,
    audit_success BOOLEAN NOT NULL,
    audit_failure BOOLEAN NOT NULL,
    UNIQUE KEY unique_acl_entry (acl_object_identity, ace_order),
    CONSTRAINT fk_acl_entry_object FOREIGN KEY (acl_object_identity) REFERENCES acl_object_identi
ty (id),
    CONSTRAINT fk_acl_entry_acl FOREIGN KEY (sid) REFERENCES acl_sid (id)
) ENGINE=InnoDB;

```

Certifique-se de ter a estrutura necessária em sua base de dados para poder contar com o suporte do projeto.