*Explicando a API*

*Começando deste ponto? Você pode fazer o* [*DOWNLOAD*](https://s3.amazonaws.com/caelum-online-public/android-chat/stages/ichat-alura-final.zip) *do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.*

Esse vídeo é destinado para os curiosos que estão interessados em saber como a API de entrega das mensagens foi implementada. Vamos fazer uma pequena revisão de como a API foi implementada, ou seja, não falaremos muito sobre as tecnologias aqui utilizadas, pois são temas de outros treinamentos daqui da Alura, então não se preocupe se achar o código muito complexo, há treinamentos para te auxiliar, como os treinamentos [**Spring MVC I: Criando aplicações web**](https://cursos.alura.com.br/course/spring-mvc-1-criando-aplicacoes-web), [**Spring Boot: Agilidade no desenvolvimento java com Spring**](https://cursos.alura.com.br/course/springboot) e [**Threads 2: Programação concorrente avançada**](https://cursos.alura.com.br/course/threads-java-2).

## **Inicializando o projeto**

Primeiramente, você deve baixar o zip do código fonte da API [aqui](https://s3.amazonaws.com/caelum-online-public/android-chat/files/ichat-api-src.zip). Extraia o zip e importe-o no Eclipse como um **projeto Maven** (***Existing Maven Projects***), nesse momento ele faz todas as configurações para sermos capaz de rodar o projeto, inclusive o download das suas dependências, então pode ser que essa configuração demore um pouco.

Com o projeto configurado, dentro do diretório **src/main/java**, conseguimos ver os pacotes do projeto, e consequentemente as suas classes. Vamos começar pelo mais simples, pela classe **Message**, que representa a **mensagem**, dentro do pacote **br.com.caelum.ichat.model**. Assim como no cliente Android, a classe também tem um atributo para representar o texto e um para identificar a mensagem:

public class Message {  
  
 private String text;  
 private Integer id;  
  
 public String getText() {  
 return text;  
 }  
  
 public Integer getId() {  
 return id;  
 }  
}

Também temos uma classe do Spring Boot, a **SpringBootStartup**, no pacote **br.com.caelum.ichat.infra**, é essa classe que faz o Tomcat subir. Para isso, basta estender a classe **SpringBootServletInitializer**, que ela faz toda essa "mágica" para nós:

@SpringBootApplication(scanBasePackages="br.com.caelum")  
public class SpringBootStartup extends SpringBootServletInitializer {  
  
 @Override  
 protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {  
 return application.sources(SpringBootStartup.class);  
 }  
  
 public static void main(String[] args) throws Exception {  
 SpringApplication.run(SpringBootStartup.class, args);  
 }  
  
}

Podemos reparar que a classe **SpringBootStartup** possui um método **main**, logo essa classe é **executável**, então vamos executá-la para que o Tomcat seja inicializado. Assim que a classe termina de executar, podemos abrir o projeto na URL <http://localhost:8080/> e ver o nosso projeto funcionando, podemos até enviar algumas mensagens para testar.

## **Recebendo requisições**

Para receber as requisições, temos os famosos ***controllers***, dentro do pacote **br.com.caelum.ichat.controller**. Temos o **IndexController**, que é o que nos leva para a página principal, ou seja, ele recebe uma requisição e nos redireciona para o **index.jsp**, que se encontra no diretório **src/main/resources/META-INF/resources/WEB-INF/views**:

@Controller  
@RequestMapping("/")  
public class IndexController {  
  
 @RequestMapping  
 public String home() {   
 return "index";  
 }  
  
}

Para realizar a entrega das mensagens, temos o **PollingController**, que recebe as requisições, tanto de GET quando de POST, para **/polling**. Então vamos entender um pouco como ele funciona. Quando o Spring instanciar essa classe, a primeira coisa que ele irá fazer é chamar o método **init()**, pois ele está anotado com **@PostConstruct**:

@PostConstruct  
public void init() {  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 while(true) {  
 try {  
 Message message = messages.take();  
 sendToClients(message);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }).start();  
}

Esse método cria e inicializa uma *thread*. Dentro dela fazemos um *loop* infinito, e dentro dessse *loop*, quando a linha **Message message = messages.take();** for executada, a *thread* irá travar, pois o método **take()**, da classe **BlockingQueue**, é bloqueante. E a execução ficará bloqueada até que haja uma mensagem dentro da **BlockingQueue**, mas em que momento que chegará essa mensagem? Quando o usuário fizer uma requisição do tipo POST, enviando uma mensagem. Podemos simular isso com o **curl**:

curl -XPOST -H "content-type: application/json" -d '{"text":"ola","id":1}' http://localhost:8080/polling

Podemos reparar que a mensagem é exibida na página. Quando fazemos ese POST, o **PollingController** recebe-o através do método **doPost**, que pegará a mensagem que está no corpo (o JSON que enviamos) e a adicionará na **BlockingQueue**, consequentemente desbloqueando a *thread*. Após a *thread* ser desbloqueada, o resto do seu código será executado, ou seja, o método **sendToClients** será chamado:

private void sendToClients(Message message) {  
 for (DeferredResult<Message> client : clients) {  
 client.setResult(message);  
 }  
}

Esse método enviará a mensagem para cada um dos clientes (há também uma *queue* de clientes) que estão acessando a API, ou seja, que fizeram um GET para **/polling**. Já esse GET é tratado no método **ouvirMensagem()**:

@ResponseBody  
@RequestMapping(method=RequestMethod.GET)  
public DeferredResult<Message> ouvirMensagem() {  
  
 long timeout = 20 \* 1000L;  
 final DeferredResult<Message> client = new DeferredResult<>(timeout);  
  
 TimeoutCallback timeoutCallback = new TimeoutCallback(client, clients);  
 ClientCallback clientCallback = new ClientCallback(client, clients);  
  
 client.onTimeout(timeoutCallback);  
 client.onCompletion(clientCallback);  
  
 clients.offer(client);  
 return client;  
}

Esse método basicamente adiciona, na *queue* de clientes, o cliente que fizer a requisição. Além disso, temos dois *callbacks* e um **DeferredResult**.

O **DeferredResult** permite com que a resposta da requisição seja enviada a partir de outra *Thread* (ao invés da Thread do container que naturalmente processa as requisições), conseguimos postergar a resposta da requisição para algum outro momento, para alguma outra *thread*. E para gerenciar isso, já que não queremos que o cliente fique preso eternamente, temos um *timeout* para o cliente ficar preso apenas por um período de tempo. Indicamos o valor do *timeout* no construtor do **DeferredResult**.

E quando acontecer o *timeout*? O **TimeoutCallback** será chamado, e ele irá remover o cliente da *queue* de clientes. Quando o cliente obtém a sua resposta, ele também é removido da fila, quem tem essa responsabilidade é o **ClientCallback**.