UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DIRETORIA DE PESQUISA E PÓS-GRADUAÇÃO

DEPARTAMENTO ACADÊMICO DE ELETRÔNICA

CURSO DE ESPECIALIZAÇÃO EM REDES DE COMPUTADORES E TELEINFORMÁTICA

GUSTAVO WEBER KUHN

APLICATIVO ANDROID PARA MENSAGENS INSTANTÂNEAS UTILIZANDO MICROSERVIÇOS REST

MONOGRAFIA DE ESPECIALIZAÇÃO

CURITIBA

2018

GUSTAVO WEBER KUHN

APLICATIVO ANDROID PARA MENSAGENS INSTANTÂNEAS UTILIZANDO MICROSERVIÇOS REST

Monografia de Especialização, apresentada ao Curso de Especialização em Redes de Computadores e Teleinformática, do Departamento Acadêmico de Eletrônica – DAELN, da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Especialista.

Orientador: Prof. M.Sc. Danillo Leal Belmonte

CURITIBA

2018

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Curitiba

Diretoria de Pesquisa e Pós-Graduação
Departamento Acadêmico de Eletrônica
Curso de Especialização em Redes de Computadores e Teleinformática

TERMO DE APROVAÇÃO

APLICATIVO ANDROID PARA MENSAGENS INSTANTÂNEAS UTILIZANDO MICROSERVIÇOS REST

por

GUSTAVO WEBER KUHN

Esta monografia foi apresentada em 10 de Julho de 2018 como requisito parcial para a obtenção do título de Especialista em Redes de Computadores e Teleinformática. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Prof. M.Sc. Danillo Leal Belmonte
Orientador

_____
Prof. Dr. Kleber Kendy Horikawa Nabas
Membro titular

_____
Prof. M.Sc. Omero Francisco Bertol
Membro titular

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

Dedico este trabalho à minha família e namorada, que me apoiaram nos momentos de ausência e de incertezas durante este trabalho.

AGRADECIMENTOS

Agradeço a tudo e todos que de alguma forma tornaram possível realizar com sucesso essa fase de minha vida.

A Deus, por tudo que proporcionou ao longo da vida, pelas pessoas que me fizeram conhecer, pelos sonhos que me fizeram ter e pelo conhecimento que me permitiu adquirir.

À família por todo o incentivo, carinho e conselhos dados, desde à infância até este momento.

Aos amigos pelo companheirismo, apoio, inúmeras horas de estudo e crescimento e triunfo compartilhados ao longo do curso.

À UTFPR e aos professores pela base técnica e científica, tão necessárias para o desenvolvimento do curso e deste trabalho.

Ao Prof. Danillo Leal Belmonte pela paciência e pelo auxílio nos momentos mais cruciais.

Por fim, à banca examinadora pela atenção e apontamentos responsáveis por engrandecer o estudo realizado.

I don’t need a hard disk in my computer if I can get to the server faster… carrying out this non-connected computers is byzantine by comparison.

(**Steve Jobs, 1997**)

RESUMO

KUHN, Gustavo Weber. **Aplicativo Android para mensagens instantâneas utilizando microserviços REST**. 2018. 43 f. Monografia de Especialização em Redes de Computadores e Teleinformática, Departamento Acadêmico de Eletrônica, Universidade Tecnológica Federal do Paraná. Curitiba, 2018.

Com o aumento dos dispositivos móveis e serviços distribuídos, é de se esperar que as aplicações existentes, as quais atuam de forma monolítica, sejam migradas aos poucos à nuvem e distribuídas em diversos microserviços, cada um responsável por uma pequena parcela do funcionamento do programa. Utilizados em provedores inovadores de computação distribuída como Amazon e Netflix, os microserviços tornam as aplicações mais escaláveis e de fácil manutenção. No presente trabalho foi separado cada domínio de um software de mensageria instantânea em serviços construídos com o Spring Framework para Java, cada etapa dessa nova abordagem é tratada de forma individual sendo elas: Definição da arquitetura, separação de tarefas para cada serviço e desenvolvimento da aplicação para Android. A comunicação da aplicação com os serviços externos é feita através do protocolo HTTP usando REST.

Palavras-chave: Micro serviços. REST. Nuvem. Android. Spring Framework.

ABSTRACT

KUHN, Gustavo Weber. **Instant messenger for android using REST microservices as backend**. 2018. 43 f. Monografia de Especialização em Redes de Computadores e Teleinformática, Departamento Acadêmico de Eletrônica, Universidade Tecnológica Federal do Paraná. Curitiba, 2018.

With the increase of mobile devices and distributed services, it is expected that the existent applications - those that works in a monolithic manner - migrate to "the cloud" and become distributed between micro services, each one responsible for making the program work in a small portion. Frequently used in innovative providers of distributed computing, such as Amazon and Netflix, the micro services architecture make applications more scalable and ease their maintenance. In this paper, the different services of an instant messaging software are built with Spring Framework for Java each stage of this new approach was analyzed separately. Such stages are: Definition of architecture tasks separation for each service; and developing an Android app. The communication from the app with external services is made using HTTP protocol and REST.

Keywords: Micro services. REST. Cloud. Android. Spring Framework.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

API	*Application Programming Interface*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transport Protocol*
IoC	*Inversion of Control* (ou Inversão de Controle)
SQL	*Structured Query Language*
URL	*Uniform Resource Locator*
XML	*eXtensible Markup Language*
XMPP	*eXtensible Messaging and Presence Protocol*

LISTA DE ACRÔNIMOS

AIM	*AOL Instant Messaging*
DAO	*Data Access Object*
ICQ	*Instant Messaging Client*
JSON	*JavaScript Object Notation*
POJO	*Plain Old Java Object*
REST	*REpresentational State Transfer*
SOAP	*Simple Object Access Protocol*

SUMÁRIO

1 INTRODUÇÃO

1.1 TEMA

À medida que instituições acadêmicas e laboratórios de pesquisa se tornaram os primeiros locais de uso do computador no início da década de 1970, os programadores começaram a desenvolver meios para se comunicar com os outros através de um sistema de mensagens de texto. Este novo sistema de mensagens permitiu que as pessoas conversassem com outros usuários do mesmo computador ou uma máquina conectada em uma rede local dentro da universidade (HOYOS, 2016).

As aplicações de mensagens instantâneas surgiram antes mesmo da *internet*, na década de 60 os sistemas operacionais multiusuários continham serviços de troca de mensagens entre os usuários das máquinas (VLECK, 2013). Com o desenvolvimento das redes de computadores, os protocolos para mensagens instantâneas evoluíram e os programas on-line foram desenvolvidos de duas maneiras, a primeira com o protocolo *peer-to-peer* fazendo a comunicação direta entre os dois usuários, a outra em um protocolo cliente-servidor obrigando o usuário a se conectar a um servidor (HOYOS, 2016). No final da década de 80 e início da década de 90 a Quantum Link ofereceu um serviço para os computadores Commodore 64 que abria uma sessão de conversa entre os usuários que usavam o equipamento de modo simultâneo utilizando somente uma tela de texto (HOYOS, 2016). No meio da década de 90 a internet recebeu vários softwares com interfaces gráficas entre eles os popularmente conhecidos ICQ e o AIM (*AOL Instant Messaging*) (HOYOS, 2016). Nos anos 2000 foi criado uma aplicação e protocolo de código aberto “open-source software” chamado Jabber também conhecido como XMPP, os servidores XMPP podem atuar com diferentes protocolos de mensagens evitando a necessidade do uso de múltiplos clientes possibilitando que um único aplicativo possa conversar com diferentes tipos de softwares de mensagens. Na década de 2010 as redes sociais iniciaram a disponibilizar o serviço de mensagens em seus *sites* (HOYOS, 2016).

Com a popularização dos *smartphones* e a facilidade de acesso aos dispositivos eletrônicos, os aplicativos de mensagens instantâneas se tornaram mais presentes na vida cotidiana e influenciaram na forma como os usuários interagem

gerando críticas sobre seu papel na sociedade, pois pode diminuir a interação face a face fazendo com que pessoas não desenvolvam suas relações sociais e acabem se isolando em suas casas (TYLER, 2002, p. 196). Por outro lado, permite a conversa entre duas pessoas em qualquer lugar do mundo sem as taxas de ligações internacionais, em ambientes empresarias trabalhadores usam a ferramenta como alternativas a reuniões ou telefonemas que são mais intrusivos e mais interruptivos que uma simples mensagem na tela do computador (BEAN-MELLINGER, 2018).

1.2 ARQUITETURA

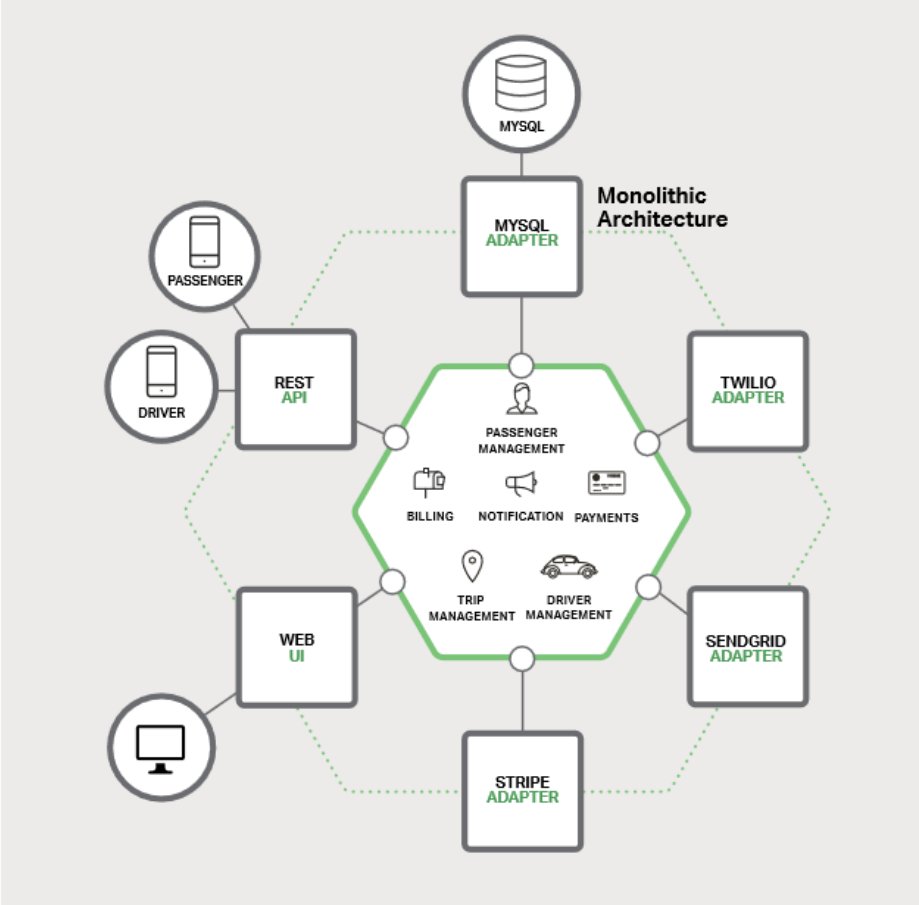
Micros serviços é um estilo arquitetônico que estrutura uma aplicação em uma coleção de elementos vagamente acoplados, que implementam regras empresariais. A arquitetura baseada em microserviços possibilita a entrega contínua de grandes e complexas aplicações (RICHARDSON, 2017).

Ao contrário das aplicações monolíticas onde a maioria ou todas as funcionalidades são agregadas em um único projeto, a arquitetura de microserviços é uma maneira de desenvolver a única aplicação como um conjunto de pequenas aplicações, cada uma rodando seu próprio processo e se comunicando utilizando mecanismos leves, frequentemente o protocolo HTTP. Essas pequenas aplicações são construídas em torno de funções empresariais e são implementadas de forma automatizada. A gestão de cada aplicação é feita de forma minimamente centralizada, possibilitando o desenvolvimento em múltiplas linguagens e diferentes tecnologias de armazenamento de dados (FOWLER; LEWIS, 2014).

Como principais vantagens dos microserviços, podem-se citar: heterogeneidade de tecnologias, poliglotismo de linguagens de programação, resiliência, escalabilidade e facilidade de implementação e desenvolvimento (YVER, 2016).

Um exemplo de aplicação monolítica seria o desenvolvimento de um aplicativo de taxi como UBER ou Cabify, após algumas reuniões iniciais para definir os módulos necessários para a construção da aplicação chega-se a arquitetura apresentada na Figura 1.

Figura 1. Arquitetura da aplicação monolítica



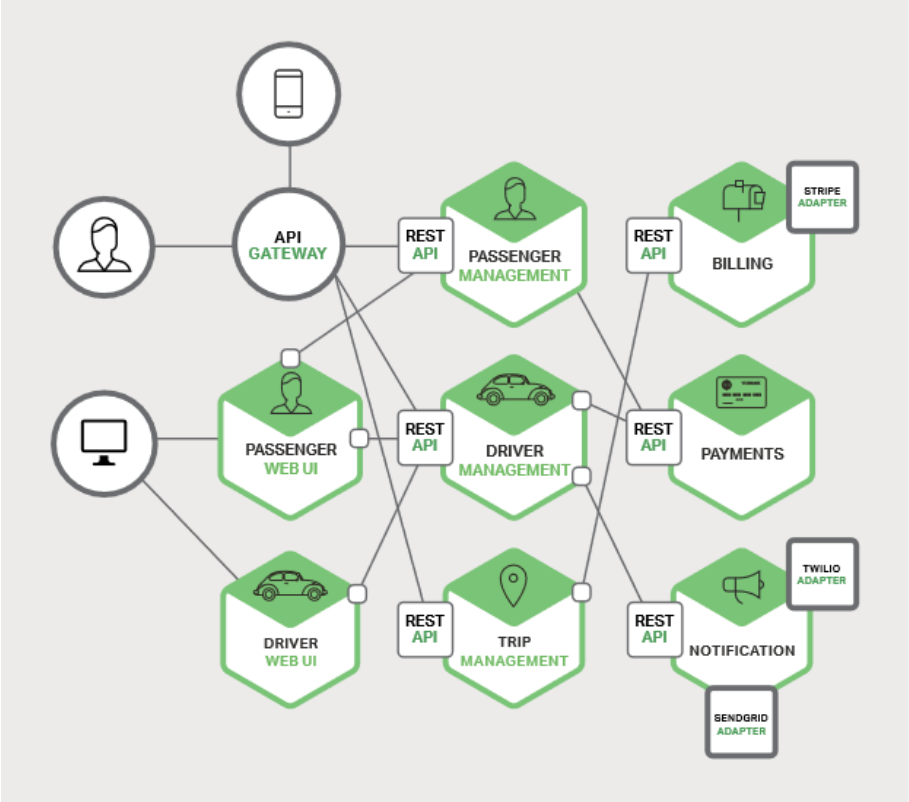
Fonte: Richardson e Smith (2016, p. 2).

Nas primeiras etapas do desenvolvimento a aplicação é construída de forma modular e de fácil entendimento para seus programadores, como na figura acima é mostrado uma aplicação central que trata todas as regras de negócios e inclui uma série de adaptadores para conexões com aplicações no mundo exterior, como banco de dados, *webservices* para as aplicações dos usuários, *websites* e outros. Após o crescimento dessa aplicação regras de negócio se tornam mais complexas, novas utilidades são incluídas e o número de programadores aumenta tornando o projeto grande o suficiente ao ponto que poucos desenvolvedores e arquitetos conhecem todas as suas funcionalidades e toda a construção do código. Aplicações desse tipo implicam em uma série de dificuldades de manutenção, o tempo de inicialização de alguns programas podem durar vários minutos tornando a fase de testes muito dispendiosa, a relação de dependências é complexa fazendo com que uma pequena mudança possa impactar em vários outros processos, a mudança ou atualização de *frameworks* requer a reescrita de milhares de linhas de código e o processo de integração contínua se torna impraticável pois várias implementações são feitas

diariamente e um tempo longo para inicialização ocasionaria um período longo de indisponibilidade do sistema (RICHARDSON; SMITH, 2016, p. 3).

Algumas organizações como Amazon, eBay e Netflix resolveram os problemas construindo aplicações utilizando o padrão de arquitetura dos microserviços. Ao invés de construir uma grande e monstruosa aplicação monolítica, a ideia é dividir o aplicativo em um conjunto de serviços menores e interconectados, no caso da aplicação de taxi pode-se dividir os microserviços em aplicações especializadas em gestão de viagens, pagamento, gestão de motoristas e gestão de passageiros, cada aplicação tendo seu próprio conjunto de conectores, banco de dados e API's. Nesse novo modelo os serviços se comunicam através de API's, como exemplo o serviço de gestão de usuários deve enviar uma requisição ao serviço de notificação para que uma mensagem seja enviada ao usuário ou o serviço de gestão de viagens deve enviar uma requisição ao serviço de gestão de motoristas para alocar o motorista mais próximo (RICHARDSON; SMITH, 2016, p. 5). Cada serviço pode ser mantido por uma equipe diferente e codificado com linguagens ou *frameworks* distintos. A Figura 2, apresenta um exemplo do aplicativo de táxis usando o padrão de microserviços.

Figura 2. Arquitetura da aplicação no padrão de microserviços



Fonte: Richardson e Smith (2016, p. 5).

1.3 PROBLEMA

O presente trabalho visa verificar as principais vantagens de se utilizar a arquitetura de microserviços, por meio do desenvolvimento de um aplicativo simples de mensagens instantâneas para Android.

1.4 JUSTIFICATIVA

Os microserviços estão atualmente no foco das atenções de artigos, *blogs*, mídias sociais e conferências sobre desenvolvimento e arquitetura de softwares. O modelo ganhou notoriedade quando grandes empresas de software em nuvem como Amazon e Netflix migraram para essa nova arquitetura.

Frente a isso, esse trabalho busca mostrar o desenvolvimento utilizando esse conceito através de um aplicativo de mensagens instantâneas para Android.

1.5 OBJETIVOS

1.5.1 Objetivo Geral

Desenvolver uma aplicação Android para envio de mensagens instantâneas usando a arquitetura microserviços na aplicação do servidor.

1.5.2 Objetivos Específicos

Para atender ao objetivo geral neste trabalho de conclusão de curso os seguintes objetivos específicos serão abordados:

- Separar as funcionalidades de cada serviço.
- Desenvolver os serviços em um servidor.
- Desenvolver a aplicação para celulares Android.

1.6 ESTRUTURA DO TRABALHO

Esta monografia de trabalho de conclusão de curso de especialização está dividida em 5 (cinco) seções ou capítulos:

- Capítulo 1 - Introdução: histórico das aplicações de mensagem instantânea, apresentação da arquitetura dos microserviços.
- Capítulo 2 - Conceitos e softwares utilizados: descrição dos *frameworks* e softwares usados no desenvolvimento do trabalho.
- Capítulo 3 - Desenvolvimento: desenvolvimento da arquitetura e do software do servidor e do aplicativo das mensagens instantâneas.
- Capítulo 4 - Análise dos resultados: resultados do desenvolvimento da aplicação.
- Capítulo 5 - Conclusão: consideração finais e conclusões sobre o desenvolvimento.

2 CONCEITOS E SOFTWARES UTILIZADOS

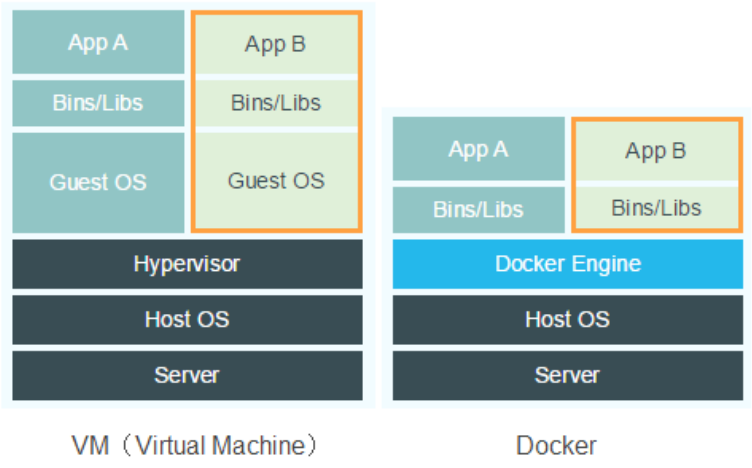
Neste capítulo serão detalhados e analisados os softwares utilizados na elaboração do projeto.

2.1 DOCKER

Docker é um software que serve como uma plataforma de *containers* – sendo estes blocos que isolam sistemas e processos, atuando como se fossem máquinas diferentes.

As máquinas virtuais comuns compartilham um mesmo hardware e tem seus recursos gerenciados através de um software chamado de “hipervisor”, em cima deste hardware compartilhado é instalado um novo sistema operacional, fazendo com que essa máquina seja tratada como uma máquina “visitante”. Os containers trabalham de forma mais leve e versátil, não compartilhando hardware, mas compartilhando recursos do “cerne” do sistema operacional hospedeiro, assim podendo pular etapas de um boot tradicional, tornando a inicialização da aplicação mais rápida. A diferença nos diagramas de virtualização é ilustrada na Figura 3 (VAUGHAN-NICHOLS, 2018).

Figura 3. Comparação diagrama de virtualização VM e Docker



Fonte: Vaughan-Nichols (2017).

O Docker oferece um repositório de imagens de containers pré-montados que são disponibilizados por usuários diversos, como por exemplo, a imagem “flopes/spring-boot-docker” que será utilizada na montagem desse projeto.

Esse sistema de virtualização de baixo custo computacional é útil no contexto

dos microserviços, pois permitem a criação de um ou vários *containers* para cada serviço podendo simular assim um ambiente de computação distribuída em uma única máquina.

2.2 SPRING FRAMEWORK

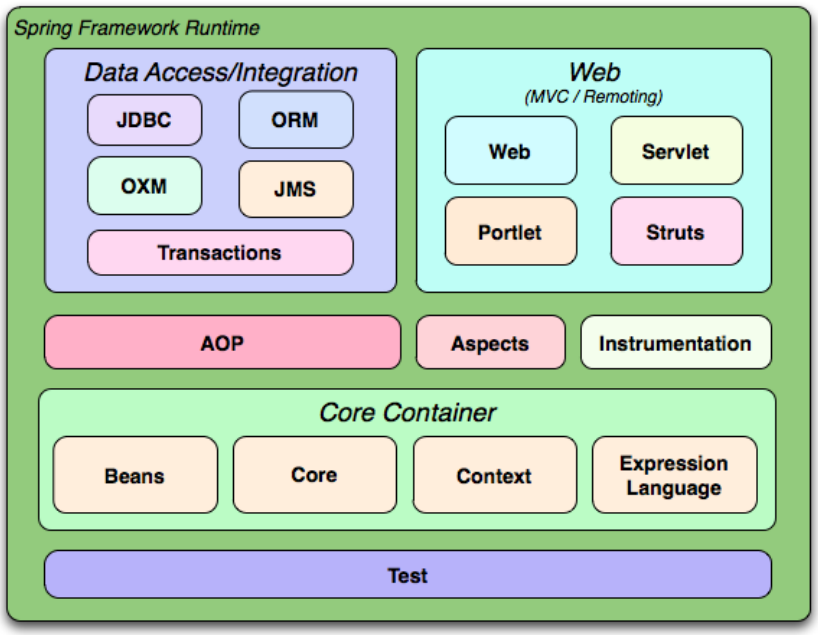
Spring Framework é uma tecnologia que trás algumas melhorias para as arquiteturas existentes do Java como inversão de controle e injeção de dependência.

Inversão de controle (*Inversion of Control*, IoC) é um padrão de sucesso na comunidade de programadores, pois provê baixo acoplamento e alta coesão no código. O IoC trata os objetos Java (*Plain Old Java Object*, POJO) como instâncias reutilizáveis chamadas “beans”, diferentemente da forma clássica onde o programador instancia cada objeto através do comando “new”, a inversão de controle faz com que os “beans” sejam iniciados pelo próprio *framework* fazendo com que exista o gerenciamento completo do ciclo de vida de cada um deles.

A injeção de dependência é uma prática que faz com que a inicialização de um objeto dependa da existência e configuração de um objeto previamente criado (SPRING FRAMEWORK, 2018).

Além de programar as melhores práticas de programação o Spring Framework oferece aos programadores vários módulos como os apresentados na Figura 4.

Figura 4. Estrutura Spring Framework



Fonte: Spring Framework (2018).

2.3 MYSQL

O MySQL é um sistema de gerenciamento de banco de dados criado pela ORACLE. Para adicionar, acessar e processar dados armazenados em um banco de dados, é necessário um software de gerenciamento de banco de dados, como o MySQL Server. Para realizar operações de consulta e transações nos bancos o MySQL utiliza a linguagem SQL (*Structured Query Language*), comum entre os vários bancos de dados relacionais do Mercado (ORACLE, 2018).

A linguagem SQL consiste em cinco elementos principais: as cláusulas, os identificadores, as expressões, as *queries* e os atributos ou predicados. As cláusulas são as ações que são requisitadas ao banco de dados, as principais são: UPDATE (atualização de um dado), SELECT (seleção de um dado), INSERT (Inserção de um dado) e DELETE (exclusão de um dado). Os identificadores servem para apontar a tabela e/ou o esquema utilizado, é precedido da cláusula FROM. As expressões são usadas para alterar o valor das variáveis de acordo com algum critério específico como por exemplo: “UPDATE table_1 SET field1 = field1 + 1”. As *queries* selecionam quais os campos devem ser retornados por uma seleção do banco e os atributos ou predicados são as condições que limitam o retorno ou os parâmetros da operação, ou seja, operam como filtros precedidos da cláusula WHERE (ORACLE, 2018).

2.4 REST

REST (*Representational State Transfer*) é um estilo arquitetônico para projetar sistemas distribuídos. É um conjunto de restrições, como ser sem estado, ter uma relação cliente / servidor e uma interface uniforme. REST não está estritamente relacionado ao HTTP, mas é mais comumente associado a ele.

Os princípios do REST são:

- Recursos: os recursos são acessados por URIs facilmente compreensíveis.
- Representação: os objetos são representados usualmente em formatos ou XML.
- Mensagens: mensagens usam os métodos HTTP explicitamente (por exemplo, GET, POST, PUT e DELETE).

- Sem estado: uma requisição não altera o estado do objeto e não interfere nas requisições posteriores.

Os métodos HTTP são usados para mapear as operações de criação, remoção, atualização e leitura em requisições HTTP, a seguir será apresentado uma explicação dos principais métodos:

a) GET: entrega de dados. O método é usado para a entrega de um recurso, não importa a quantidade de requisições que é feita os resultados devem ser os mesmos. Por exemplo, para a requisição do recurso com identificador 1: *GET /addresses/1*

b) POST: criação de dados. O método é usado para criação e, ás vezes, para atualização do dado. A requisição deve ser feita com o objeto no corpo da chamada em formato ou XML. Por exemplo, para criação de um recurso: *POST /addresses*

c) PUT: atualização ou criação de dados. O método é usado para atualização ou criação do dado, sua diferença em relação ao POST é o fato que o PUT não altera seu resultado esperado independentemente da quantidade de vezes em que é requisitado, ou seja, é um método idempotente. Assim como o POST, o objeto deve estar no corpo da requisição em formato XML. Por exemplo, para criação ou atualização do elemento com identificador 1: *PUT /addresses/1*

d) DELETE: remoção de dados. O método é usado para a exclusão de um dado ou objeto da base. Por exemplo: para remover o elemento com identificador 1: *DELETE /addresses/1*

Os resultados das requisições são dados através dos códigos de retorno do protocolo HTTP. A Tabela 1 mostra a relação dos códigos do protocolo.

Tabela 1. Relação dos códigos do protocolo HTTP

CÓDIGO	SIGNIFICADO
1XX	Informação
2XX	Sucesso
3XX	Redirecionamento
4XX	Erro no cliente
5XX	Erro no servidor

Fonte: Spring Framework (2018). Disponível em: <https://spring.io/understanding/REST>. Acesso em: 24 jan. 2018.

A definição do formato em que o recurso será retornado ou enviado para o servidor é dada pelos elementos do cabeçalho da requisição HTTP chamados *Content-Type* e *Accept*. No caso da aplicação cliente necessitar que o recurso seja retornado em JSON o cabeçalho deverá conter o campo ”Accept: application/json”, se o cliente irá enviar o objeto no mesmo formato, o cabeçalho deverá conter: “Content-Type: application/json” (REST, 2018).

3 DESENVOLVIMENTO

Os procedimentos adotados para o desenvolvimento do projeto foram: definir as tarefas atribuídas a cada serviço, construção dos serviços, construção do aplicativo consumidor, testes de comportamento e desempenho. Neste capítulo serão detalhados os passos do desenvolvimento deste trabalho.

3.1 DEFINIÇÃO DAS TAREFAS ATRIBUÍDAS PARA CADA SERVIÇO

A arquitetura baseada em microserviços tem como lema a frase “faça uma coisa e a faça bem-feita”, essa frase descreve a arquitetura da aplicação e demonstra que pequenos serviços devem ser especializados em fazer apenas uma tarefa dentro da aplicação. Para delimitar quais serão as atividades de cada serviço é necessário escolher as tarefas que cada serviço irá efetuar, cada serviço poderá ser desenvolvido por uma equipe individual com a linguagem de programação que achar necessária, apenas devendo respeitar regras básicas como qualidade de código e qualidade dos testes feitos. O escopo de cada atividade da aplicação deve seguir a máxima granularidade possível, ou seja, sempre limitar ao máximo o seu funcionamento a fim de facilitar a escalabilidade e manutenção do serviço.

Para a aplicação de mensagens instantâneas é necessário construir pelo menos dois serviços, um para fazer escrita e leitura de mensagens e outro para administração de usuários, cada serviço funciona de forma independente e apenas a aplicação final é responsável por fazer o relacionamento entre eles. Os serviços foram construídos utilizando o Spring Framework para Java e hospedados em um servidor Linux usando o software Docker, o banco de dados é um MYSQL também hospedado no servidor. Cada serviço possui sua tabela no banco de dados e seu respectivo *webservice* e porta no servidor, abaixo é representando a estrutura dos microserviços no servidor.

O micro serviço de recebimento e envio das mensagens possui uma API REST para comunicação com o aplicativo Android desenvolvido. O Spring Framework será responsável pela criação dos endereços de envio e recebimento de mensagens, assim como o cadastro de clientes. O banco de dados MYSQL é usado para

armazenamento das mensagens para que possam ser entregues para o usuário final caso este esteja indisponível no momento em que a mensagem é enviada.

3.1.1 Serviço de Envio de Mensagens

O Serviço encarregado do envio das mensagens é uma aplicação feita usando o SpringBoot. Essa aplicação é dividida em três partes, a primeira parte é responsável pela conexão e comunicação com o banco de dados, essa parte é chamada DAO (Apêndice C), a segunda parte é onde está o modelo das mensagens, chamada de Model (Apêndice D), e a terceira parte é a chama Controller (Apêndice B), essa parte é onde são mapeados os links que farão acesso aos recursos, essa controller é baseada no REST.

Existem três tipos de mensagem na aplicação, a primeira é a mensagem enviada pelo aplicativo com as informações do número do destinatário, número do remetente, códigos de identificação da mensagem e a mensagem. A segunda é uma requisição com o número do usuário, quando o servidor recebe essa requisição irá retornar todas as mensagens enviadas para aquele usuário específico. Um exemplo é a requisição do usuário número 002 para o servidor, será feita uma requisição do aplicativo do celular para o link /message/user/2 do servidor e será retornado em formato (Apêndice A) uma lista com as mensagens destinadas a esse usuário. Após o recebimento da mensagem o cliente Android irá receber a mensagem e retornar uma mensagem ao servidor para o link /message/confirm em formato (Apêndice A).

3.1.2 Serviço de Cadastro de Usuários

Para o cadastro dos usuários foi criado um outro serviço onde é recebido o cadastro e criado o usuário no banco de dados da aplicação. Para o envio dos dados do usuário é feito uma requisição REST para cadastro do usuário. O corpo da mensagem enviada via método POST é apresentado no (Apêndice E) e seus modelos e controladores no Apêndice F e Apêndice G, respectivamente. Por exemplo para que seja consultado os dados do usuário número 1 é enviado uma requisição para o *link* “/user/1”.

3.1.3 Aplicativo Android

Para confecção do aplicativo android foram utilizadas diversas tarefas em paralelo, as principais são as tarefas que fazem a coleta das informações nos serviços do servidor, o código é responsável por fazer uma requisição REST ao servidor e guardar os resultados internamente utilizando um banco de dados interno da aplicação.

Como no Apêndice G é possível ver que o código faz uma requisição e quando recebe a mensangem com o código 200 é feita a inserção da nova mensagem ou novo usuário no banco de dados do aparelho.

4 ANÁLISE DOS RESULTADOS

4.1 TROCA DE MENSAGENS

Como resultado dos desenvolvimentos é apresentado ao usuário, as telas: a) Sala de troca de mensagens; b) Sala de conversas do usuário; e c) Escolha de usuário para nova conversa.

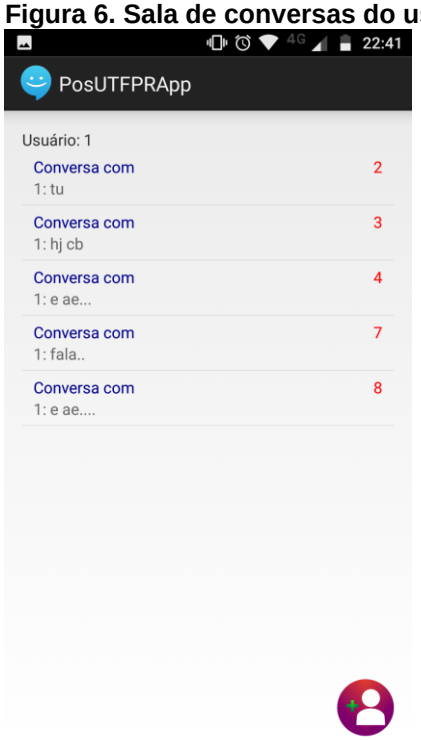
Cada *smartphone* possui o aplicativo separadamente e com usuários diferentes. Assim através da sala de trocas de mensagens (Figura 5) é possível a troca de mensagens.

Figura 5. Sala de troca de mensagens

Fonte: Autoria própria.

Para seleção de salas de mensagens é apresentado ao usuário a tela apresentada na Figura 6.

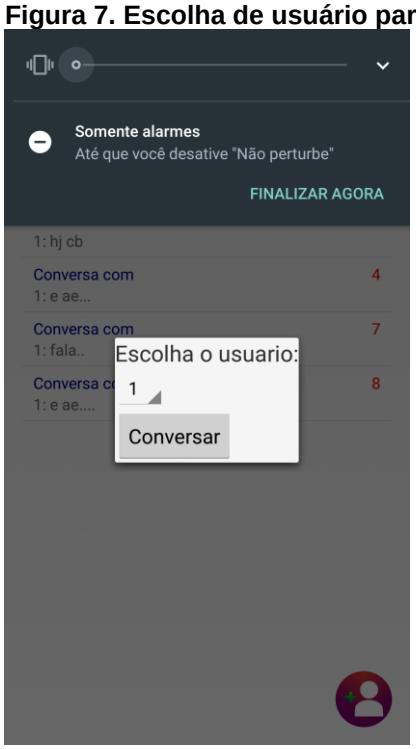
Figura 6. Sala de conversas do usuário



Fonte: Autoria própria.

Para seleção de um novo usuário é apresentado ao usuário uma tela para a escolha como demonstra a Figura 7.

Figura 7. Escolha de usuário para nova conversa



Fonte: Autoria própria.

5 CONCLUSÃO

A arquitetura baseada em microserviços se mostra interessante pelo fato de que cada serviço da aplicação pode ser escalável individualmente. Como exemplo o serviço de inscrição dos usuários pode ser escalável com um fator menor do que o serviço de troca de mensagens.

A vantagem do uso de comunicação usando serviços REST permite que uma mesma aplicação possa utilizar múltiplos serviços apenas mudando o apontamento das requisições para endereços diferentes.

A aplicação android é composta pelas telas de trocas de mensagens e inscrição de usuário, cada uma das quais são associadas a um serviço individual.

Contudo, conclui-se que utilizando a arquitetura de microserviços é possível que uma aplicação seja construída de forma organizada e independente, permitindo que várias equipes diferentes possam trabalhar juntar sem dependências de tecnologia. A manutenção do software fica mais simplificada visto que cada serviço tem suas funcionalidade bem limitadas e assim tem seu código reduzido.

REFERÊNCIAS

BEAN-MELLINGER, Barbara. **Advantages & disadvantages of instant messaging in business**. Small Business - Chron.com, 29 jun. 2018. Disponível em: <http://smallbusiness.chron.com/advantages-disadvantages-instant-messaging-business-21284.html>. Acesso em: 03 jul. 2018.

FOWLER, Martin; LEWIS, James. **Microservices: a definition of this new architectural term**. Copyright© martinfowler.com, publicado em: 25 mar. 2014. Disponível em: <https://martinfowler.com/articles/microservices.html >. Acesso em: 02 jul. 2018.

HOYOS, Brandon de. **IRC, ICQ, AIM and More: a history of instant messaging**. Lifewire, publicado em: 18 out. 2016. Disponível em: <https://www.lifewire.com/im-a-brief-history-1949611>. Acesso em: 02 jul. 2018.

ORACLE. **MySQL 5.7 Reference Manual**. Oracle Corporation, copyright© 2018. Disponível em: <MySQL: https://dev.mysql.com/doc/refman/5.7/en/what-is-mysql.html>. Acesso em: 02 jul. 2018.

REST. **Understanding REST**. Pivotal Software, copyright© 2018. Disponível em: <https://spring.io/understanding/REST>. Acesso em: 02 jul. 2018.

RICHARDSON, Chris. **Microservice Architecture: what are microservices?** Chris Richardson, copyright© 2017. Disponível em: <http://microservices.io/>. Acesso em: 03 jul. 2018.

RICHARDSON, Chris; SMITH, Floyd. **Microservices from design to deployment**. Editora: NGINX Inc, 18 mai. 2016. Disponível em: <https://www.nginx.com/blog/microservices-from-design-to-deployment-ebook-nginx/>. Acesso em: 03 jul. 2018.

SPRING FRAMEWORK. **Spring Framework Documentation**. Pivotal Software, copyright© 2018. Disponível em: <https://docs.spring.io/spring/docs/2.0.x/reference/beans.html>. Acesso em: 02 jul. 2018.

TYLER, Tom R. **Is the internet changing social life? It seems the more things change, the more they stay the same**. Journal of Social Issues, v. 58, n.1, 2002, p. 195-205. Disponível em: <https://spssi.onlinelibrary.wiley.com/doi/abs/10.1111/1540-4560.00256>. Acesso em: 03 jul. 2018.

VAUGHAN-NICHOLS, Steven J. **What is Docker and why is it so darn popular?** Copyright© ZDNET, publicado em: 21 mar. 2018. Disponível em: <http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/>. Acesso em: 02 jul. 2018.

VLECK, Tom Van. **The history of electronic mail**. Copyright© Tom Van Vleck, publicado em: 01 fev. 2001 e atualizado em: 08 ago. 2013. Disponível em: <http://www.multicians.org/thvv/mail-history.html>. Acesso em: 02 jul. 2018.

YVER, Sreek. **Key benefits of Webservices**. Copyright© IBM Community, publicado em: 21 nov. 2016. Disponível em: <https://www.ibm.com/developerworks/community/blogs/sreek/entry/micro-3?lang=en>. Acesso em: 02 jul. 2018.

APÊNDICE A - MENSAGEM DE ENVIO PARA O SERVIDOR

Mensagem de envio:


```
{
    "messages": [
        {
            "messageid": 113,
            "from": 3,
            "to": 1,
            "senttime": 1508527596000,
            "message": "TESTE",
            "received": false,
            "userMessageId": 24
        }
    ],
    "status": "Messages available"
}
```


Confirmação:


```
{
    "ConfirmMessage": [
        {
            "messageid": 113,
            "received": true
        },
            {
            "messageid": 114,
            "received": true
        },
            {
            "messageid": 115,
            "received": true
        },
            {
            "messageid": 116,
            "received": true
        }
    ],
}
```


Fonte: Autoria própria. Disponível em: <https://github.com/adrarkius/posutfprproject>. Acesso em: 03 jul. 2018.

APÊNDICE B - CONTROLADORA DE TROCA DE MENSAGENS

```
package io.gkuhn.messagebroker.controller;

...

@Controller
@RequestMapping(path="/message")
public class MessageEventController {

@Autowired
private MessageEventRepository messageRepository;

@RequestMapping(path="/user/{userId}", produces="application/json",
method=RequestMethod.GET)
      public @ResponseBody ResponseMessageEvent
getByUser(@PathVariable(value="userId", required = true)int id) {
            List<MessageEvent> responseMessages = new ArrayList<>();
            ResponseMessageEvent response = new ResponseMessageEvent("No message
found", responseMessages);
responseMessages.addAll(messageRepository.findByToAndReceived(id, false));

                  if(responseMessages.size() > 0) {
                        response.setStatus("Messages available");
                        response.setMessages(responseMessages);
                  }

                  return response;
       }

@RequestMapping(value="/", consumes="application/json",
method=RequestMethod.POST)
public @ResponseBody ResponseEntity<ResponseMessageEvent>
sendMessage(@RequestBody List<MessageEvent> messages) {
messages.forEach(t-> t.setMessageid(0));
       messages.forEach(t-> t.setReceived(false));
       messages.forEach(t-> t.setSenttime(new Date()));
       List<MessageEvent> messageListReturn = messageRepository.save(messages);
ResponseMessageEvent response = new ResponseMessageEvent("message sent",
messageListReturn);

            return (new ResponseEntity<ResponseMessageEvent>(response,
HttpStatus.CREATED));
 }

@RequestMapping(value="/confirm", consumes="application/json",
method=RequestMethod.POST)
public @ResponseBody List<Integer> confirmMessage(@RequestBody
List<ConfirmMessageEvent> messageConfirmList) {
List<Integer> resultList = new ArrayList<>();
if(messageConfirmList != null) {
       for(ConfirmMessageEvent messageConfirmElement : messageConfirmList) {
       MessageEvent message =
messageRepository.findOne(messageConfirmElement.getId());
       if(message != null) {
            message.setReceived(true);
            messageRepository.save(message);
            resultList.add(message.getMessageid());
```

```
        }
}
                    }

                    return resultList;
                }
}
```

Fonte: Autoria própria. Disponível em: <https://github.com/adrarkius/posutfprproject>. Acesso em: 03 jul. 2018.

APÊNDICE C - DAO DO SERVIÇO DE TROCA DE MENSAGENS

```
package io.gkuhn.messagebroker.dao;

import io.gkuhn.messagebroker.model.MessageEvent;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface MessageEventRepository extends JpaRepository<MessageEvent,
Integer>{

	List<MessageEvent> findByToAndReceived(int id, boolean received);

}
```

Fonte: Autoria própria. Disponível em: <https://github.com/adrarkius/posutfprproject>. Acesso em: 03 jul. 2018.

APÊNDICE D - MODELOS DO SERVIÇO DE TROCA DE MENSAGENS

ConfirmMessageEvent.java:

```
package io.gkuhn.messagebroker.model;

public class ConfirmMessageEvent {

	private int id;
	private boolean received;

	public ConfirmMessageEvent() {

	}

	public ConfirmMessageEvent(int id, boolean received) {
		this.id = id;
		this.received = received;
	}

	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public boolean isReceived() {
		return received;
	}
	public void setReceived(boolean received) {
		this.received = received;
	}


}
```

MessageEvent.java:

```
package io.gkuhn.messagebroker.model;

import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity(name="messageevent")
public class MessageEvent {

	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	@Column(name="`messageid`")
	private int messageid;

	@Column(name="`from`")
```

```
	private int from;

	@Column(name="`to`")
	private int to;

	@Column(name="`senttime`")
	private Date senttime;

	@Column(name="`message`")
	private String message;

	@Column(name="`received`")
	private boolean received;

	@Column(name="`userMessageId`")
	private Integer userMessageId;



	public MessageEvent(int from, int to, Date senttime, String message,
boolean received, Integer userMessageId) {
		this.from = from;
		this.to = to;
		this.senttime = senttime;
		this.message = message;
		this.received = received;
		this.userMessageId = userMessageId;
	}
```

ResponseMessageEvent.java:

```
package io.gkuhn.messagebroker.model;

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

public class ResponseMessageEvent {

	List <MessageEvent> messages = new ArrayList<>();
	private String status;

	public ResponseMessageEvent(String status, List<MessageEvent>
messages) {
		super();
		this.status = status;
		this.messages = messages;
	}

	public String getStatus() {
		return status;
	}

	public void setStatus(String status) {
		this.status = status;
	}

	public List<MessageEvent> getMessages() {
		return messages;
	}
```

```
	public void setMessages(List<MessageEvent> messages) {
		this.messages = messages;
	}
}
```

Fonte: Autoria própria. . Disponível em: <https://github.com/adrarkius/posutfprproject>. Acesso em: 03 jul. 2018.

APÊNDICE E - MENSAGEM DE INSCRIÇÃO DO USUÁRIO

```
{
    "userid": 1,
    "name": "Gustavo",
    "familyname": null,
    "birthdate": null
}
```

Fonte: Autoria própria. Disponível em: <https://github.com/adrarkius/posutfprproject>. Acesso em: 03 jul. 2018.

APÊNDICE F - CONTROLADORA E MODELOS DO SERVIÇO DE INSCRIÇÃO DE USUÁRIOS

UserController.java:

```
package io.gkuhn.userbroker.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

import io.gkuhn.userbroker.dao.UserRepository;
import io.gkuhn.userbroker.model.User;

@Controller
@RequestMapping(path="/user")
public class UserController {

	@Autowired
	private UserRepository userRepository;

	@RequestMapping(path="/{userId}", produces="application/json",
method=RequestMethod.GET)
	public @ResponseBody User getByUser(@PathVariable(value="userId",
required = true)int id) {
		User response = userRepository.findByUserid(id);
		return response;
	}

	@RequestMapping(value="/", consumes="application/json",
method=RequestMethod.POST)
	public @ResponseBody ResponseEntity<User> sendUser(@RequestBody User
user) {
		User userReturn = userRepository.save(user);
		return (new ResponseEntity<User>(userReturn,
HttpStatus.CREATED));
	}


}
```

User.java:

```
package io.gkuhn.userbroker.model;

import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity(name="users")
public class User {

	@Id
	@GeneratedValue(strategy=GenerationType.AUTO)
	@Column(name="`userid`")
	private int userid;

	@Column(name="`name`")
	private String name;

	@Column(name="`familyname`")
	private String familyname;

	@Column(name="`birthdate`")
	private Date birthdate;


	public User(String name, String familyname, Date birthdate) {
		this.name = name;
		this.familyname = familyname;
		this.birthdate = birthdate;
	}

	public User() { }

	public int getUserid() {
		return userid;
	}

	public void setUserid(int userid) {
		this.userid = userid;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getFamilyname() {
		return familyname;
	}

	public void setFamilyname(String familyname) {
		this.familyname = familyname;
```

```
	}

	public Date getBirthdate() {
		return birthdate;
	}

	public void setBirthdate(Date birthdate) {
		this.birthdate = birthdate;
	}
}
```

UserRepository.java:

```
package io.gkuhn.userbroker.dao;


import io.gkuhn.userbroker.model.User;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends JpaRepository<User, Integer>{

	User findByUserid(int id);

}
```

Fonte: Autoria própria. Disponível em: <https://github.com/adrarkius/posutfprproject>. Acesso em: 03 jul. 2018.

APÊNDICE G - CLASSE DE ENVIO E RECEBIMENTO DOS DADOS NO APLICATIVO ANDROID

```
  HttpClient httpClient = new DefaultHttpClient(httpParameters);
  HttpGet httpGet = new HttpGet(URL);
  try {
      HttpResponse response = httpClient.execute(httpGet);
      StatusLine statusLine = response.getStatusLine();
      int statusCode = statusLine.getStatusCode();
      if (statusCode == 200) {
          HttpEntity entity = response.getEntity();
          InputStream inputStream = entity.getContent();
          BufferedReader reader = new BufferedReader(
                  new InputStreamReader(inputStream));
          String line;
          while ((line = reader.readLine()) != null) {
              stringBuilder.append(line);
          }
          inputStream.close();

          result = stringBuilder.toString();
      } else {

          Log.d("JSON", "Failed to download file");
      }

  } catch (Exception e) {
      Log.d("readJSONFeed", e.getLocalizedMessage());
      Log.d("readJsonFeed", result);

  }
  return result;
}
```

Fonte: Autoria própria. Disponível em: <https://github.com/adrarkius/posutfprproject>. Acesso em: 03 jul. 2018.