Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

JACQUES VIEIRA DOS SANTOS TREVISAN

APLICATIVO PARA AUXILIAR PESSOAS NA REALIZAÇÃO DE ATIVIDADES FÍSICAS: UM ESTUDO DO DESENVOLVIMENTO COM TECNOLOGIAS HÍBRIDAS E WEBSERVICES

Assis/SP
2021

Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

JACQUES VIEIRA DOS SANTOS TREVISAN

APLICATIVO PARA AUXILIAR PESSOAS NA REALIZAÇÃO DE ATIVIDADES FÍSICAS: UM ESTUDO DO DESENVOLVIMENTO COM TECNOLOGIAS HÍBRIDAS E WEBSERVICES

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Bacharelado em Ciência da Computação do Instituto Municipal de Ensino Superior de Assis – IMESA e a Fundação Educacional do Município de Assis – FEMA, como requisito parcial à obtenção do Certificado de Conclusão.

Orientando: Jacques Vieira Dos Santos Trevisan
Orientador: Dr. Almir Rogério Camolesi

Assis/SP
2021

FICHA CATALOGRÁFICA

T814a TREVISAN, Jacques Vieira dos Santos
Aplicativo para auxiliar pessoas na realização de atividades físicas: um estudo do desenvolvimento com tecnologias híbridas e webservices / Jacques Vieira dos Santos Trevisan. – Assis, 2021.

61p.

Trabalho de conclusão do curso (Ciência da Computação). – Fundação Educacional do Município de Assis-FEMA

Orientador: Dr. Almir Rogério Camolesi

1.Mobile 2.Aplicativo-webservices

CDD005.133

APLICATIVO PARA AUXILIAR PESSOAS NA REALIZAÇÃO DE ATIVIDADES FÍSICAS: UM ESTUDO DO DESENVOLVIMENTO COM TECNOLOGIAS HÍBRIDAS E WEBSERVICES

JACQUES VIEIRA DOS SANTOS TREVISAN

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Bacharelado em Ciência da Computação do Instituto Municipal de Ensino Superior de Assis – IMESA e a Fundação Educacional do Município de Assis – FEMA, como requisito do Curso de Graduação, avaliado pela seguinte comissão examinadora:

Orientador: _____

Dr. Almir Rogério Camolesi

Examinador: _____

Me. Guilherme de Cleva Farto

Assis/SP
2021

DEDICATÓRIA

Dedico este trabalho a meu pai Vicente Trevisan Filho e minha mãe Silvia Regina Vieira Dos Santos que investiram na minha educação, e a meus irmãos que influenciaram positivamente na escolha da minha formação e carreira.

RESUMO

Com a evolução dos *smartphones* na vida pessoal da população mundial e o aumento na popularidade de soluções móveis, é importante a consideração de atender os dois sistemas operacionais em destaque do mercado, *Android* e *iOS*. Para isso surgiu a necessidade de frameworks híbridas, que prometem com um único código fonte traduzir e compilar para múltiplas plataformas. Neste trabalho foram estudadas as tecnologias para desenvolvimento móvel que possibilitam o desenvolvimento híbrido assim como APIs REST para a implementação de um servidor para a aplicação, responsável pelo controle dos dados. Com objetivo final de implementar um aplicativo móvel utilizando a *framework* híbrida *Ionic* que consome um *web service* em *Java* com *Spring Boot.* O aplicativo é focado em *fitness,* sua proposta é de ser uma plataforma para *personal trainers* elaborarem, personalizarem e distribuírem aulas, que podem ser acessadas pelos alunos interessados em específicos tipos de exercícios que melhor servirem a seu perfil. Dois aspectos de destaque do aplicativo são os exemplos com fotos ou vídeos, que podem serem cadastrados pelo instrutor, em cada atividade e o sistema de exercícios modulares, que podem ser aplicados com diferentes parâmetros e atender diferentes perfis de aluno. Este trabalho teve como produto final um MVP da proposta apresentada assim como considerações positivas e negativas encontradas durante o processo de desenvolvimento ao trabalhar com tais tecnologias móveis.

Palavras-chave: Mobile; Desenvolvimento Híbrido; Web Service; Fitness.

ABSTRACT

With the evolution of smartphones in the personal lives of the world population and the increase in the popularity of mobile solutions, it is important to consider serving the two leading operating systems in the market, Android and iOS. For this to happen, the need for hybrid frameworks emerged, which promise to translate and compile for multiple platforms with a single source code. In this work, technologies for mobile development that enable hybrid development as well as REST APIs for the implementation of a server for the application, responsible for data control, were studied. With the ultimate goal of implementing a mobile application using the hybrid Ionic framework that consumes a Java web service with Spring Boot. The application is focused on fitness, its proposal is to be a platform for personal trainers to prepare, customize and distribute classes, which can be accessed by students interested in specific types of exercises that best suit their profile. Two outstanding aspects of the application are the examples with photos or videos, which can be added by the instructor in each activity and the modular exercise system, which can be applied with different parameters and meet different student profiles. This work had as final product an MVP of the presented proposal as well as positive and negative considerations found during the development process when working with such mobile technologies.

Keywords: Mobile; Hybrid Development; Web service; Fitness.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
HTML	Hypertext Markup Language
CSS	Cascading Style Sheet
SQL	Structured Query Language
IDE	Integrated Development Environment
APK	Android Application Pack
JPA	Java Persistence API
MVC	Model View Controller
WEB	World Wide Web
DTO	Data Transfer Object
CLI	Command Line Interface
JVM	Java Virtual Machine
COVID	Corona Virus Disease
SGBD	Sistema de Gerenciamento de Banco de Dados
JDK	Java Development Kit
MVP	Minimal Viable Product

SUMÁRIO

1. INTRODUÇÃO

Na última década, pode-se notar o aumento pelo interesse na área de tecnologias móveis, refletindo na oportunidade de grandes empresas migrarem para o uso de aplicativos, assim como também pelas pequenas empresas. Os fatores que tornam isto possível são principalmente a popularidade dos smartphones e a ideia de um pequeno aparelho conter todas as soluções necessárias no dia a dia.

Tradicionalmente, há uma concorrência nítida na questão dos principais sistemas operacionais de *smartphones*. Começando pelo iOS lançado pela Apple em 2007, seguido do Android em 2008 pela Google e finalmente o Windows Phone em 2010 pela Microsoft. Neste trabalho serão levados em conta apenas o *Android* e *iOS* devido a descontinuidade do Windows Phone em 2015. Atualmente o Android lidera operando em aproximadamente 72.8% dos *smartphones* globalmente, iOS em aproximadamente 26.6% e outros sistemas menos relevantes em menos de 1% (KOAY, 2020).

Para o desenvolvimento de aplicativos móveis, no momento de planejamento, deve ser levada em consideração esta divisão do mercado e se decidir se a abordagem será nativa ou híbrida, ao escolher nativa o desenvolvimento torna-se mais complexo, pois para atender aos dois mercados haveriam dois projetos cada um na linguagem nativa do sistema correspondente. Não só nas tecnologias de *software*, mas até recentemente a linguagem *Swift*, própria da *Apple*, para aplicativos *iOS*, *macOS* e outros produtos da empresa, só poderia ser compilada no hardware de um computador *Mac*.

Para a abordagem híbrida, um único código fonte poderia ser usado na compilação para ambas as plataformas móveis, e dependendo do *framework* utilizado, uma versão *web* também seria possível, tornando o projeto no geral mais barato (SILVA e SANTOS, 2014).

Neste trabalho será explorado o *Ionic*, um *framework* de desenvolvimento híbrido, e WebServices para a criação de um aplicativo de *personal trainer*, tal aplicativo tem o objetivo de facilitar a entrega de aulas personalizadas de um instrutor a seus alunos.

1.1. OBJETIVOS

Implementar um aplicativo móvel para iOS e Android utilizando o *framework Ionic* que consome um WebService Java com *Spring Boot*. Este aplicativo consiste numa plataforma para que *personal trainers* publiquem aulas personalizadas a seus alunos. Este aplicativo explora e aplica a computação móvel utilizando uma Arquitetura Orientada a Serviços.

1.2. JUSTIFICATIVA

Um estudo conduzido pela Freeletics com mais de dois mil brasileiros em 2019 apresenta dois principais motivos pela preferência dos aplicativos em relação a frequentar uma academia: o primeiro sendo comodidade, poder praticar os exercícios sem cronogramas, do lugar que preferir, e o segundo sendo o custo da mensalidade das aulas acompanhadas. Além disto, devido à pandemia, várias academias foram fechadas nos períodos de *lockdown* e instrutores e alunos buscavam uma forma de continuar com os exercícios de forma remota.

1.3. MOTIVAÇÃO

A motivação para o desenvolvimento do presente trabalho residiu na constatação do crescimento do interesse de pessoas em obter orientação personalizada para atividades físicas remotas. A grande preocupação dos instrutores de academia em relação aos aplicativos existentes é o risco que o aluno se coloca ao praticar os exercícios sem a devida orientação. Outra motivação é o interesse pessoal por tecnologias móveis.

1.4. PERSPECTIVA DE CONTRIBUIÇÃO

Este trabalho explora *frameworks* móveis e uso de WebServices para desenvolvimento de soluções. O *software* desenvolvido oferece uma plataforma em que profissionais disponibilizam conteúdo para usuários que buscam uma forma de se exercitar de maneira remota, mas ainda com o acompanhamento de um instrutor quando necessário. A plataforma poderá futuramente ser ampliada para não só envolver a área de educação física, mas também da saúde, por exemplo, para fisioterapia.

1.5. METODOLOGIA

Para o desenvolvimento do trabalho foi utilizada a metodologia experimental finalizando com a implementação de um estudo de caso, aplicativo de *personal trainer*. Inicialmente, para atender aos objetivos estabelecidos, será feito o planejamento do projeto, levantando requisitos para o *software* a ser criado, serão utilizados também diagramas de caso de uso, modelagem lógica do banco de dados e prototipação de telas.

Para o desenvolvimento do *software*, utilizou-se das tecnologias: Java com o *framework* Spring Boot para a API REST responsável por processar as requisições HTTP; JavaScript com o *framework* Ionic e Angular para o aplicativo mobile, a decisão do uso do Ionic é sua tecnologia híbrida, pois, o mesmo possibilita com o código fonte em HTML, CSS, JavaScript e TypeScript, gerar uma aplicação para ambas as plataformas iOS e Android (GOMIDE, 2020); por fim para o banco de dados, PostgreSQL.

1.6. RECURSOS NECESSÁRIOS

No contexto de *hardware*, foi necessário um computador com o sistema operacional Windows com capacidade para o desenvolvimento *back-end* e *front-end* do aplicativo e atuar como servidor, onde será hospedado a API e o Banco de Dados. Também será necessário um celular com sistema operacional iOS ou Android para execução do aplicativo.

Para o desenvolvimento, no contexto de *software*, foi utilizado a IDE IntelliJ para a linguagem de programação Java, com a *framework* Spring Boot, e Visual Studio Code para JavaScript, juntamente com a *framework* Ionic e Angular.

1.7. ESTRUTURA DO TRABALHO

● **Capítulo 1** – Introdução: Contextualização, hipótese, objetivos, justificativa, estado da arte e perspectivas de contribuição desta pesquisa.

● **Capítulo 2** – Revisão bibliográfica: Revisão bibliográfica dos recursos de software, conceitos aplicados e uma abordagem detalhada das tecnologias a serem utilizadas.

● **Capítulo 3** – Planejamento do *software* e Implementação: Levantamento de requisitos mínimos para o desenvolvimento do produto minimamente viável. Posteriormente, o desenvolvimento dos componentes do *software*, e a prototipação das telas.

● **Capítulo 4** – Processo de desenvolvimento da aplicação móvel e API.

● **Capítulo 5** – Resultados obtidos e recursos implementados na aplicação móvel.

● **Capítulo 6** – Conclusão: Discussão dos resultados, e considerações finais deste trabalho.

● **Referências** – Fontes de artigos e publicações referenciadas durante o trabalho.

2. REVISÃO BIBLIOGRÁFICA

Nesta seção serão revisados os conceitos de desenvolvimento móvel nas plataformas Android e iOS, as opões de *frameworks* para o desenvolvimento híbrido e tecnologias usadas para a codificação dos Web Services.

2.1. DESENVOLVIMENTO MÓVEL

A Figura 1 apresenta a divisão de mercado para sistemas operacionais móveis no último ano, é notável a proporção em que as duas mais presentes dominam o mercado atual, juntos, *Android* e *iOS* formam mais de 99% de cobertura do mercado móvel, nenhum dos dois devem ser desprezados no planejamento e publicação de um aplicativo, pois suas parcelas são significativas.

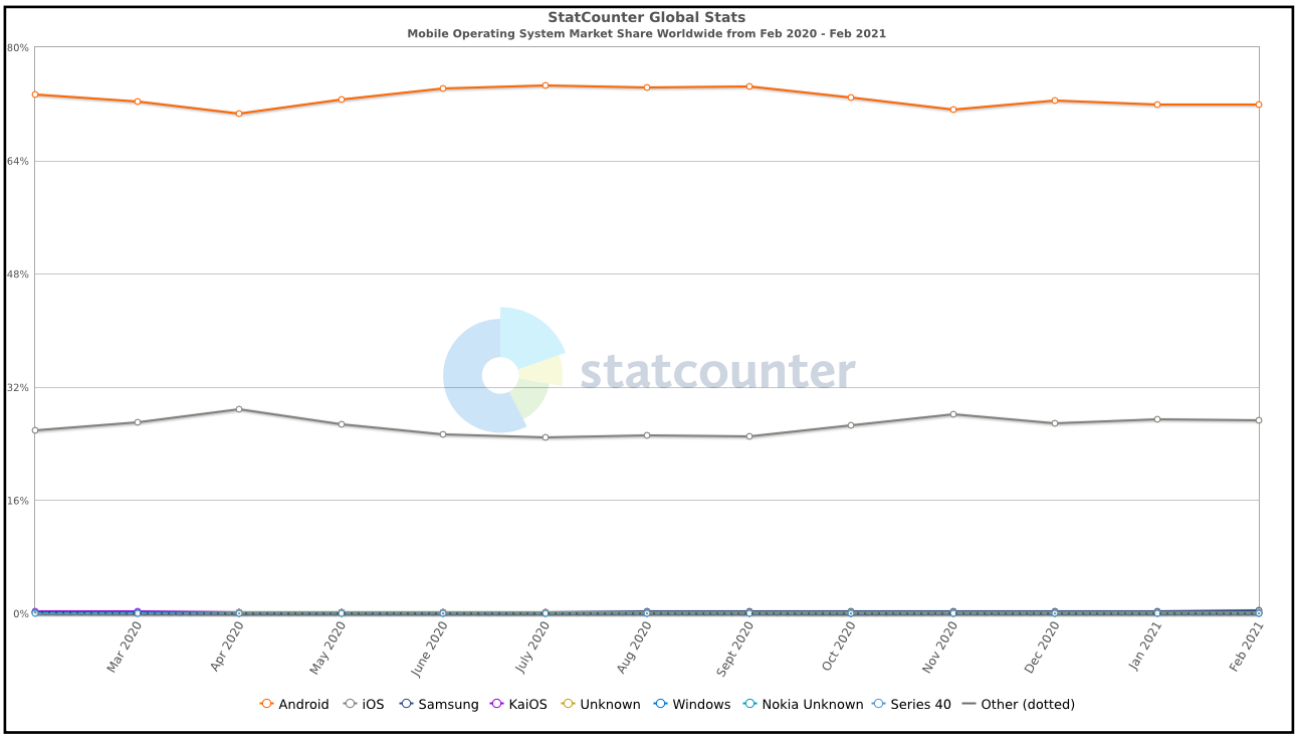

Figura 1: Divisão de mercado dos sistemas operacionais de *smartphones*.

Fonte: StatCounter, 2021

2.1.1. Android

Lançado pela Google, o Android é um sistema operacional *Open Source* para uma gama de aparelhos móveis, com uma arquitetura principalmente baseada no kernel do Linux (Android Developers, 2021). No desenvolvimento Android é utilizado um SDK que oferece ferramentas para a codificação na linguagem Java. A Figura 2 é um diagrama da arquitetura do SO exemplificando o papel do Java no sistema como a interface entre o *runtime* e os aplicativos para o usuário final.

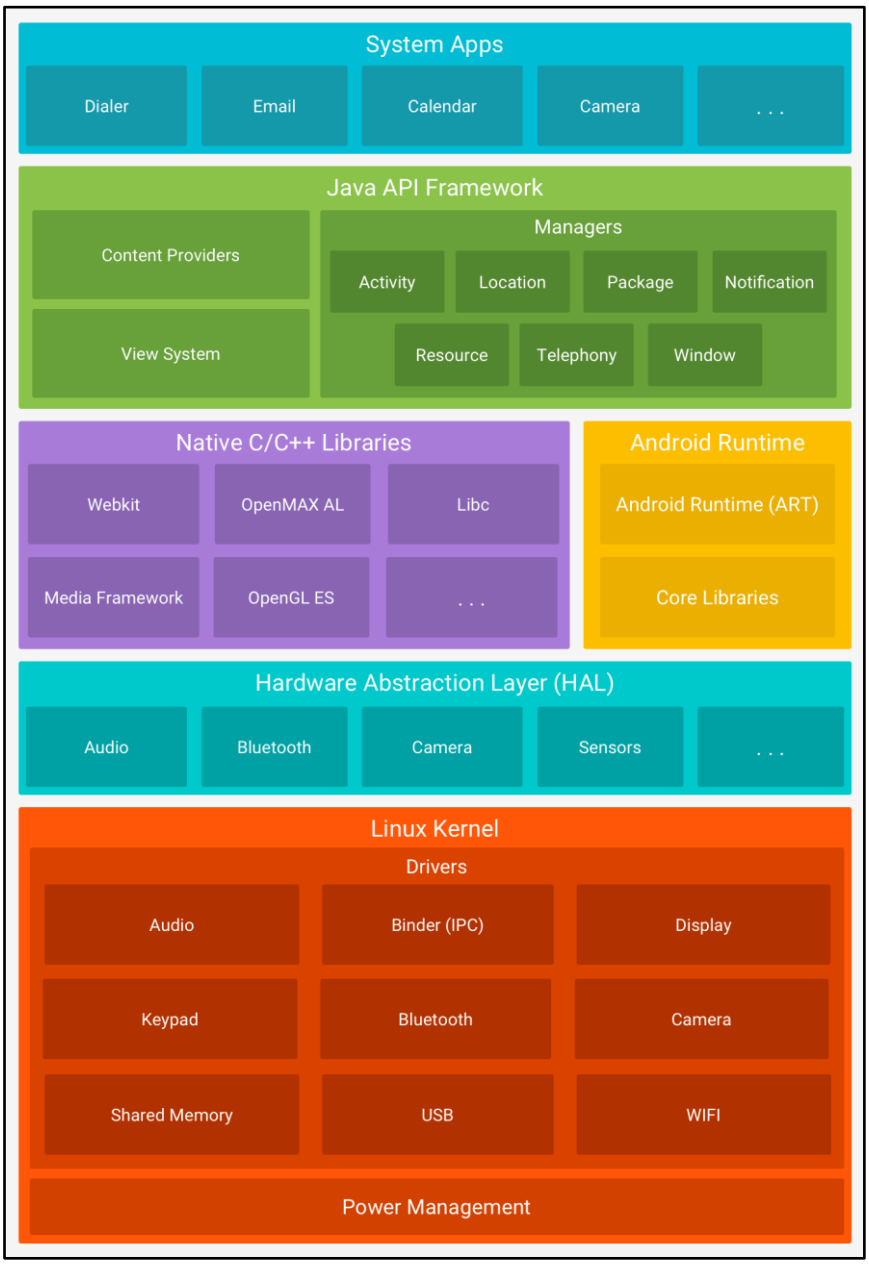

Figura 2: *Stack* de *softwares* Android.

Fonte: Android Developers, 2021

É recomendado, pela documentação, o uso da IDE *Android Studio* que também disponibiliza um gerenciador de dispositivos virtuais, assim é possível emular o aplicativo sem exportá-lo. O código final em Java então é compilado para um arquivo APK que pode ser instalado em um aparelho Android.

2.1.2. iOS

A linguagem de programação da *Apple* é o *Swift* que oferece binários para *macOS* e *Linux* que podem compilar código para *iOS*, *macOS*, *watchOS*, *tvOS* e *Linux*. Em setembro de 2020 foi anunciado um *port* oficial do *Swift* para *Windows*, tornando mais acessível a criação de aplicativos nativos pela plataforma suportada (ABDULRASOOL, 2020).

2.1.3. Tecnologias Para Desenvolvimento Móvel

Embora a programação de aplicativos nativos possa vir a ser mais performáticos por não dependerem de bibliotecas externas, pode ser caro para um projeto a manutenção de dois códigos. No conceito de desenvolvimento híbrido é feito o uso de *frameworks* que aproveitam o mesmo código para gerar o produto final independente do sistema, por meio de tradução ou como *Progressive Web Apps (BIØRN-HANSEN, 2017)*, um navegador embutido exclusivo para a aplicação, também conhecido como *WebView*. Entre as *frameworks* mais populares, que aplicam o conceito de programação híbrida estão *Ionic* lançado em 2013, *React Native* em 2015 e por fim *Flutter* em 2017.

Flutter sendo a mais recente e tendo ganhado popularidade recentemente é uma *framework* para a linguagem *Dart*, é orientado a *widgets* e contém diversos componentes que podem seguir um estilo único ou usar o padrão do sistema escolhido na hora da exportação (NEVES e JUNIOR, 2020). Ou seja, o mesmo app pode ter a configuração *Material Design* (padrão *Android*) e *Cupertino* (padrão iOS) ou manter uma para ambas versões final.

React Native utiliza o código HTML, CSS e JavaScript para traduzir os componentes web para componentes nativos de cada tipo de sistema como explicado no livro Learning React Native (EISENMAN, 2018, p. 17). Originado da framework para web ReactJS, ambos criados pela empresa Facebook.

Por fim *Ionic*, que é muito comparado com *React Native* por ser codificado em documentos *HTML, CSS e JavaScript* (IONIC, 2020). Um diferencial *Ionic* é que ao invés de gerar código nativo, ele gera apenas um *WebView* que envolve toda a aplicação, porém seus componentes e elementos seguem um layout de UI responsivo com estilo nativo. Até a versão 3 do *Ionic* apenas era possível utilizá-lo em projetos *Angular*. A partir da versão 4, não há dependências de *framework* base, ou seja, o *Ionic* pode ser usado com *JavaScript* independentemente ou como parte de projetos *Vue*, *React* e, mantendo-se retrocompatível, *Angular*. No momento de *build* é possível apenas executar o projeto normalmente no navegador ou gerar sua versão para as plataformas móveis com *Cordova* ou *Capacitor*, ambos são *frameworks* que fazem a comunicação do *JavaScript* com as funcionalidades nativas da plataforma escolhida. A diferença principal destas opções é que o *Cordova* foi desenvolvido pela *Apache* e em 2019 o *Ionic* lançou seu próprio serviço, *Capacitor*.

Angular é um *framework* para *web* criado pela Google (ANGULAR, 2021), que auxilia na construção de páginas, gerenciamento de rotas, e tem módulos para comunicação com WebServices. Em 2019 foi lançado *Ionic* 4 que trouxe grandes mudanças na base da *framework* para que pudesse ser usada com *JavaScript* individualmente puro, e abrindo a possibilidade de utilizar qualquer *framework* paralelamente. Para este projeto foi escolhido o *Ionic* pela sua galeria de componentes acompanhado do *Angular*, para aproveitar de seus módulos para WebService.

Blini (2016) apresenta algumas vantagens do framework, entre estas se destacam: “*Write once, run anywhere”* (um código fonte para múltiplas plataformas); A curva de aprendizado é pequena; Boa documentação e diversos guias de uso disponíveis. Já para as desvantagens, a preocupação principal é com o Android, pois aparelhos mais antigos podem sofrer com pouca performance na execução da aplicação e não é garantido que todas as funcionalidades se integrem corretamente com tais dispositivos, assim como componentes nativos que são dependentes da plataforma com designs finais divergente, comparando a execução no Android com a execução no iOS.

2.2. DESENVOLVIMENTO DE WEB APIS

O conceito de API é uma coleção de rotinas e padrões de programação para acesso a um *software* externo. É feita uma requisição para API com uma entrada, o *software* em questão faz todo o processamento e retorna a saída ao remetente da requisição. Uma Web API é implementada geralmente utilizando o protocolo HTTP, assim qualquer aplicação capaz de fazer requisições HTTP pode se comunicar com esta API, um dos paradigmas de construção de Web APIs é o REST, quando um projeto implementa corretamente os conceitos, ele é considerado RESTful (KULKARNI e TAKALIKAR, 2018).

Neste capítulo serão descritas as tecnologias a serem usadas na construção da API do aplicativo proposto neste trabalho.

2.2.1. Java

Java é uma linguagem de programação que geralmente é aplicado o paradigma de orientação a objetos de forma prolixa, e têm sido muito usados como linguagem para APIs por sua robustez e variedade de bibliotecas disponíveis.

Um dos conceitos que facilita grande parte da implementação de uma aplicação com conexão a um banco de dados é o JPA. Uma Implementação do JPA permite tratar a persistência de dados em uma aplicação, garantindo a consistência da definição do banco de dados com os objetos mapeados no projeto em questão de chaves primárias, validação de dados e relacionamentos. Outro ponto forte do JPA é a possibilidade de manipular um registro como instância de um objeto, a interface JPA tratará toda lógica por trás da busca, inserção, atualização e exclusão de dados.

O que separa uma simples API de um WebService, é justamente a comunicação cliente servidor por meio de requisições, mais comumente requisições HTTP. Este conjunto é o paradigma do REST, um conceito que guia a criação de serviços de requisições HTTP, definindo regras, como a de quais métodos devem ser utilizados para cada operação que a API processará, ao seguir estas regras a API pode ser categorizada como RESTful. Esta arquitetura não é obrigatória, mas ao respeitá-la, estará em um padrão que facilitará para clientes externos que consumiram os serviços e esperam o mapeamento padrão.

Já para o projeto como um todo, a arquitetura MVC é ideal para APIs e aplicações Web segundo LUCIANO e ALVES (2011). Sua organização agrupa classes em: *model*, *view* e *controller*.

O fluxo da estrutura MVC está demonstrada no diagrama da Figura 3. Da esquerda para a direita, *Database* representa o repositório onde os dados serão persistidos, e não deves ser diretamente exposta ao usuário do sistema, e para isso, uma aplicação intermediará estes dados (tanto recebidos quanto enviados) para que sejam tratados e redirecionados corretamente. A aplicação em si, implementará a camada *Model que* será a estrutura de dados mais próxima do mapeamento do repositório.

O *Controller* é um módulo da aplicação que será responsável receber e responder requisições HTTP (Representado por *HTTP Request*) vindas do usuário. Estas requisições podem ser para leitura ou gravação de dados, e é responsabilidade do *Controller* aplicar regras de negócio e formatar os dados de forma apropriada conforme o destino, seja este, o banco de dados ou a camada *View*. A parte visível ao usuário, e que permitirá o envio de requisições HTTP, é a *View*. Nesta camada não conterá regras de negócio e validações rígidas, mas sim a exibição das informações e interfaces para interações, como o cadastro, atualização ou exclusão dos dados.

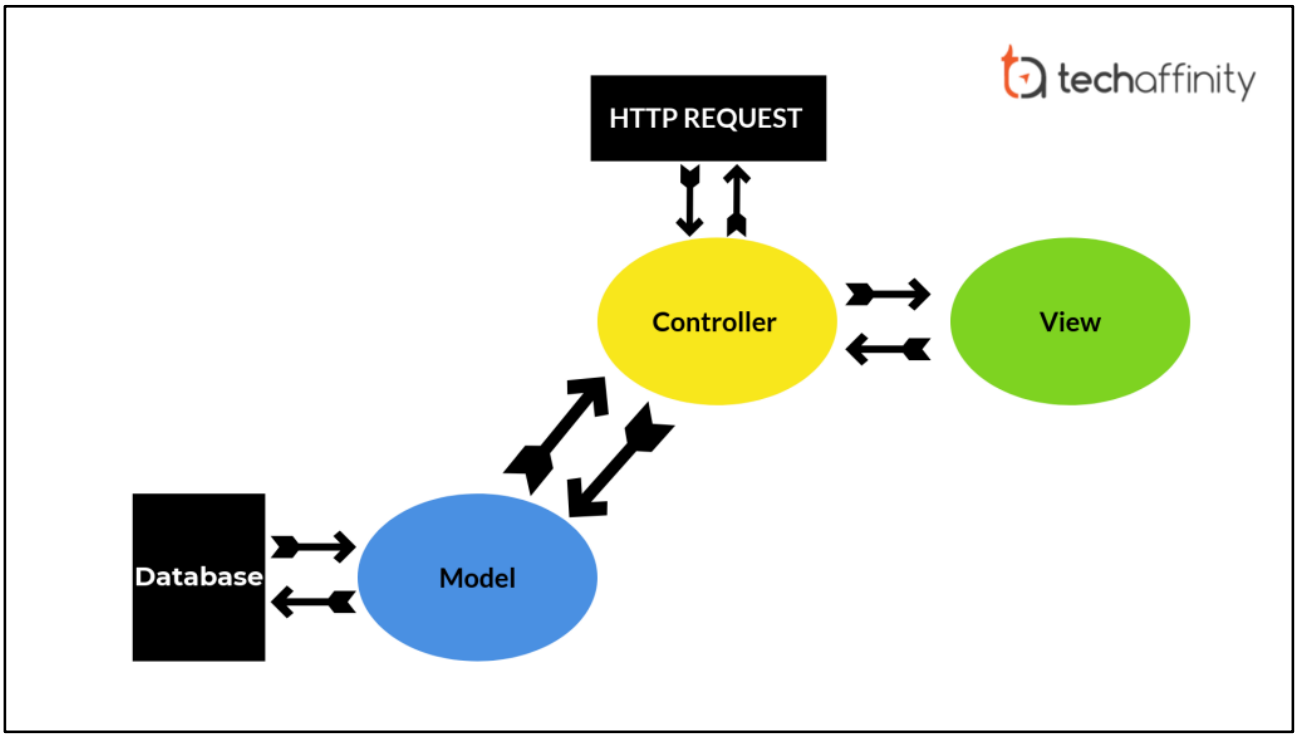

Figura 3: Diagrama que representa a estrutura *MVC* com requisições *web* e conexão com a base de dados.

Fonte: TechAffinity, 2020

Para este projeto, um aplicativo móvel em Ionic portará a *View*, então o fluxo de dados ocorre de forma que: o *Controller,* situado no servidor Java, redireciona as informações, mapeados como os *models,* através de requisições HTTP, para recuperação ou persistência no banco de dados.

2.2.2. Spring Boot

Spring é um *framework open source* para Java com diversas soluções para o desenvolvimento de projetos. O foco inicial era em aplicações corporativas de grande escala, mas chega a ser usado em projeto de qualquer finalidade atualmente pois proporciona facilidades gerais e configurações padrão que atende a maioria das aplicações, não somente par WebServices, mas por sua praticidade ganhou popularidade no desenvolvimento de APIs REST. Spring Boot na verdade é um projeto base configurado com algumas das bibliotecas da Spring prontas para APIs em Java. A conveniência do Spring Boot está disponível das seguintes formas: Pela linha de comando em uma máquina com o Spring Boot CLI instalado, ou pelo site Spring Initializr, onde é possível selecionar os parâmetros da geração do projeto graficamente. Informações do guia *Spring Boot in action* de Craig Walls (2016) e documentação oficial (2021).

Neste trabalho foi gerado o projeto com a ferramenta online Spring Initializr, que contará com as dependências: Spring Data JPA, Spring Web, Spring DevTools, Spring Test e Spring Validation. Assim como outras dependências externas como o *driver* do banco de dados, neste caso do PostgreSQL, e Lombok.

Uma das facilidades de destaque da *framework*, é o fato de não precisar configurar separadamente um servidor de aplicação, pois a aplicação conta com um servidor Tomcat interno e ao ser executada na JVM já se encontra funcional na rede (MOSCHETTI, 2020). Também é válido destacar os conceitos de inversão de controle e injeção de dependência em que é baseado e são utilizados no momento da implementação de componentes específicos do projeto.

3. PROPOSTA DO ESTUDO DE CASO

Este trabalho tem como proposta um aplicativo de *personal trainer* desenvolvido com as tecnologias mencionadas nos capítulos anteriores.

3.1. DEFINIÇÃO DO PROBLEMA A RESOLVER

Segundo um estudo da *Freeletics* conduzido em 2019, aproximadamente 40% dos brasileiros irão dar preferência a aplicativos de exercícios físicos a comparecer às academias para seguir um estilo de vida saudável. Este estudo conduzido pela Freeletics com mais de dois mil brasileiros em 2019 apresenta dois principais motivos pela preferência dos aplicativos à uma academia com estabelecimento físico, o primeiro sendo comodidade, poder praticar os exercícios sem cronogramas, do lugar que preferir, e o segundo sendo o custo da mensalidade das aulas acompanhadas. A Figura 4 apresenta os resultados de uma entrevista com 2.046 pessoas sedentárias no brasil.

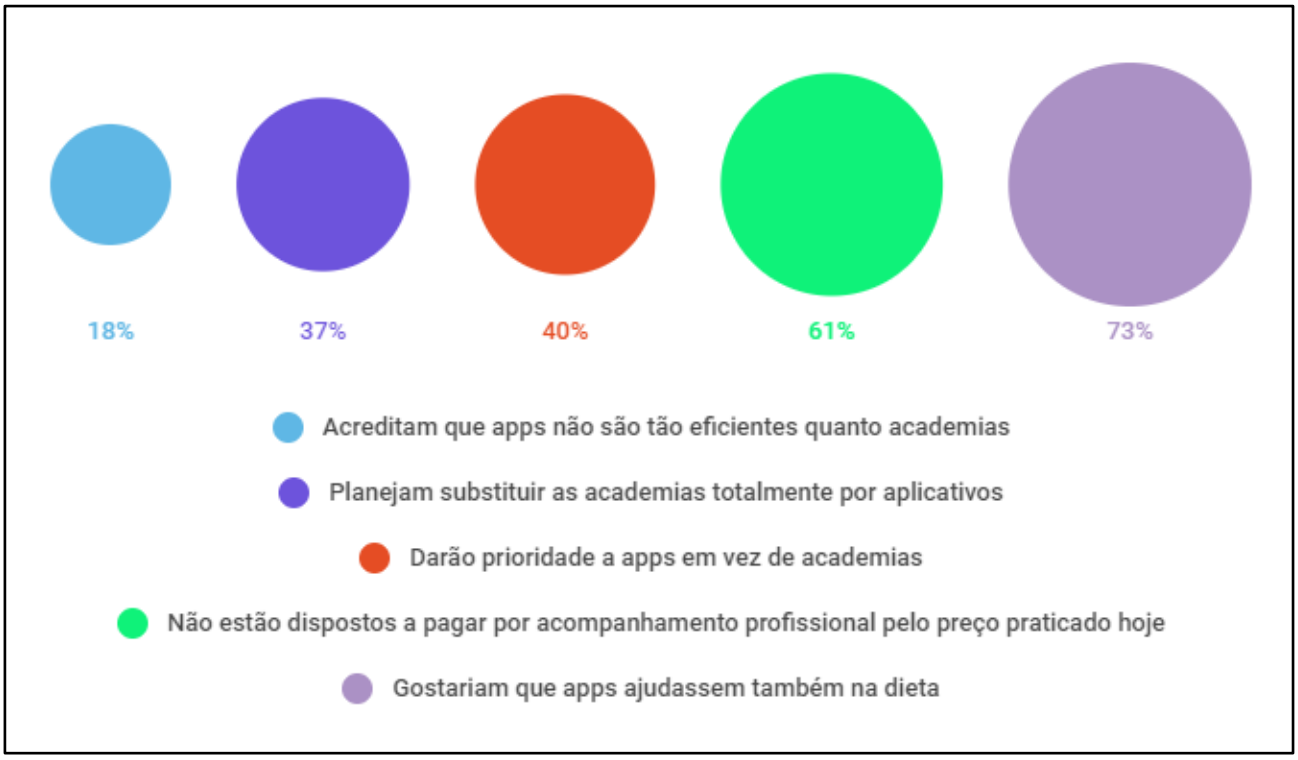

Figura 4: Mapa Nacional do Impacto da Tecnologia no Esporte e Sedentarismo

Fonte: *Freeletics*, 2019

Com o acontecimento recente da pandemia do COVID-19, por lei, muitos estabelecimentos foram impedidos de abrir, para evitar a aglomeração de pessoas, e as academias de ginástica em geral tiveram grande perda, levando vários alunos a apelarem para o uso dos aplicativos. As implementações existentes tentam substituir o conhecimento de um especialista por dicas pré-cadastradas que podem não servir a um certo perfil. Mesmo que o conteúdo tenha a qualificação e qualidade profissional, acaba sendo genérico.

O diferencial da implementação deste trabalho foi na elaboração das aulas customizadas, e no *feedback* que pode ser do aluno para o instrutor. Reduzindo assim os riscos existentes na prática de atividades físicas sem acompanhamento profissional, umas das maiores preocupações dos instrutores de academia. Riscos comuns são a carga excessiva e posição ou movimentos errados que podem trazer de pequenas a grandes lesões (PERUCCI, 2016).

3.2. INTRODUÇÃO À APLICAÇÃO

A aplicação consiste em uma plataforma que os instrutores de atividades físicas podem cadastrar exercícios genéricos e com eles elaborar aulas personalizadas e enviá-las aos alunos de destino. A principal proposta é a modularidade que não limita o *personal trainer* ao pensamento de generalizar uma aula, já que podem ser personalizadas por aluno de forma prática.

Usuários do tipo instrutor poderão cadastrar exercícios com uma pequena descrição e exemplos, que mais tarde serão atribuídos a um *wrapper*, onde serão definidos as características variáveis do exercício como repetições, duração e variante do material utilizado, um conjunto destes *wrappers* formarão uma aula que pode ser pública ou privada sendo a única diferença que aulas privadas são apenas acessadas com o *link* gerado na hora do cadastro, enquanto aulas públicas poderão ser encontradas ao explorar a biblioteca global do aplicativo.

Usuários do tipo aluno poderão buscar por aulas públicas na aba de pesquisa do aplicativo, ou acessar uma aula privada através do *link* enviado pelo *personal trainer*. Ao encontrar a aula desejada, estarão disponíveis as opções de adicionar a aula na biblioteca pessoal, marcar tal aula como favorita para aparecer no topo da biblioteca independente da data

adquirida e iniciar uma nova sessão com o cronograma da aula. Posteriormente, poderão enviar esta sessão como um relatório ao *personal trainer* criador da aula, para ser avaliado o progresso do aluno nas atividades.

No perfil de ambos os tipos de usuários haverá a listagem dos meios de contato externo com o usuário, possibilitando por exemplo alunos tirarem dúvidas com instrutores e outras formas de *feedback*.

3.3. DIAGRAMA DE CASO DE USO

A aplicação possui dois atores principais: instrutor e aluno. Grande parte das rotinas são do usuário instrutor, pois ele atuará como criador de conteúdo (público ou privado). Entretanto, espera-se que o maior fluxo de dados venha por parte dos alunos que realizarão diversas sessões em cada aula adquirida.

O diagrama de caso de uso da Figura 5, representa as ações disponíveis na visão do usuário de tipo “Aluno”.

Para o aluno, encontram-se as opções de gerenciamento de conta comum a todos os usuários, sendo elas o cadastro, login, alteração dos dados pessoais e desativação da conta. Após autenticado é possível navegar para as rotinas referentes a sua biblioteca de aulas adquiridas e o repositório de aulas públicas.

Em sua biblioteca encontra-se uma lista de suas aulas adquiridas as quais usuário tem a opção de iniciar uma nova sessão. Uma sessão servirá de relatório ao instrutor posteriormente, serão guardados hora de início, hora de término e opcionalmente comentários adicionais do aluno sobre a sessão. Ainda em sua biblioteca as funções de gerenciamento planejadas são a exclusão da aula de sua lista e a marcação como favorito para fixá-la no topo para fácil acesso.

A aquisição de aulas pode ser feita navegando pelo repositório de aulas públicas disponíveis a todos os usuários ou através de um *link* de acesso gerado em aulas privadas, tais apenas podem ser obtidas pelo *link* e não são alcançáveis no repositório público.

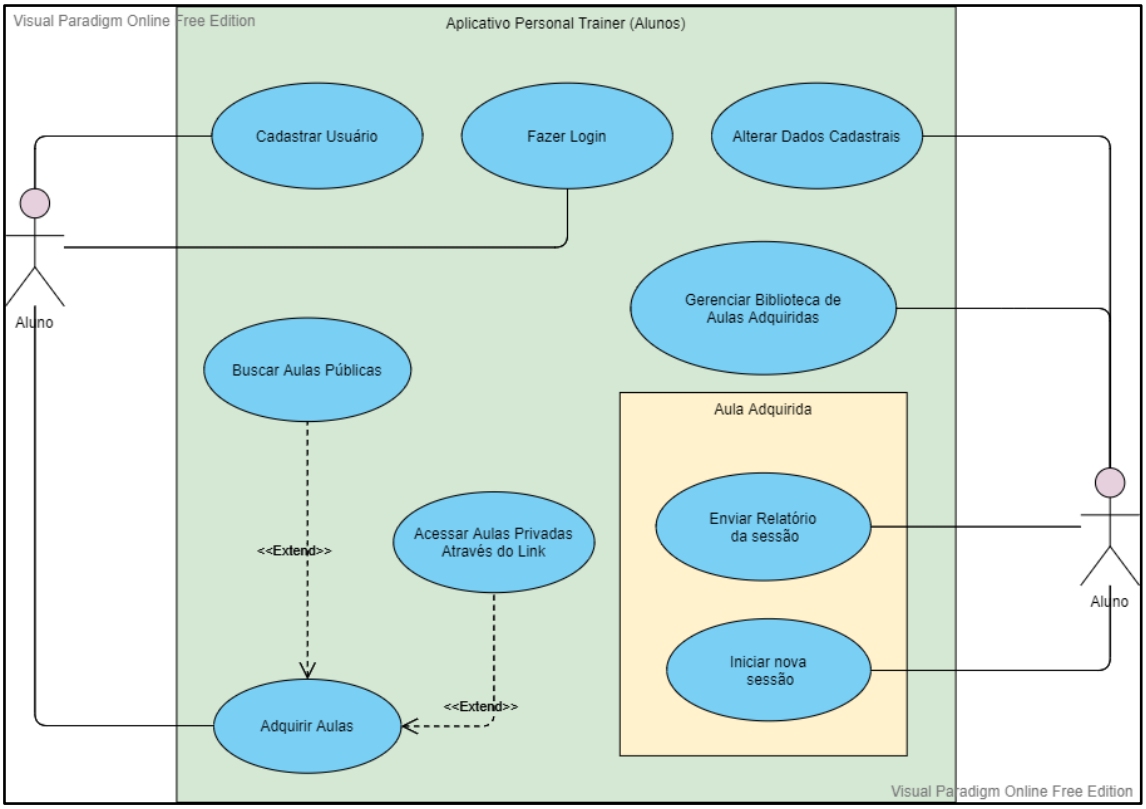

Figura 5: Diagrama de casos de uso em que o ator é o usuário do tipo aluno.

O diagrama de caso de uso da Figura 6, representa as ações disponíveis na visão do usuário de tipo “Instrutor”.

O instrutor também possui as rotinas referentes a seu usuário como login e alteração dos dados, porém o cadastro deste tipo de usuário será restrito inicialmente e serão inclusos por demanda. Esta regra permite controlar quem tem permissões para publicar na plataforma.

O cadastro de exercício é o início do fluxo principal do instrutor. Em um exercício pode, além de especificá-lo por um título e descrição, adicionar arquivos de mídia como fotos e vídeos para que alunos usem de guia e reduza o risco más interpretações de instruções escritas. Os exercícios cadastrados são exibidos no momento da criação de uma nova aula, permitindo com que o conteúdo possa ser reaproveitado em contextos diferentes. Este reaproveitamento é possível por uma configuração intermediária entre a aula e o exercício, em que é possível especificar as variáveis dependentes do público alvo, sendo elas: séries, repetições, duração, carga do material e uma descrição para parâmetros adicionais. A elaboração de uma aula se baseia em uma sequência destas estruturas que parametrizam os exercícios.

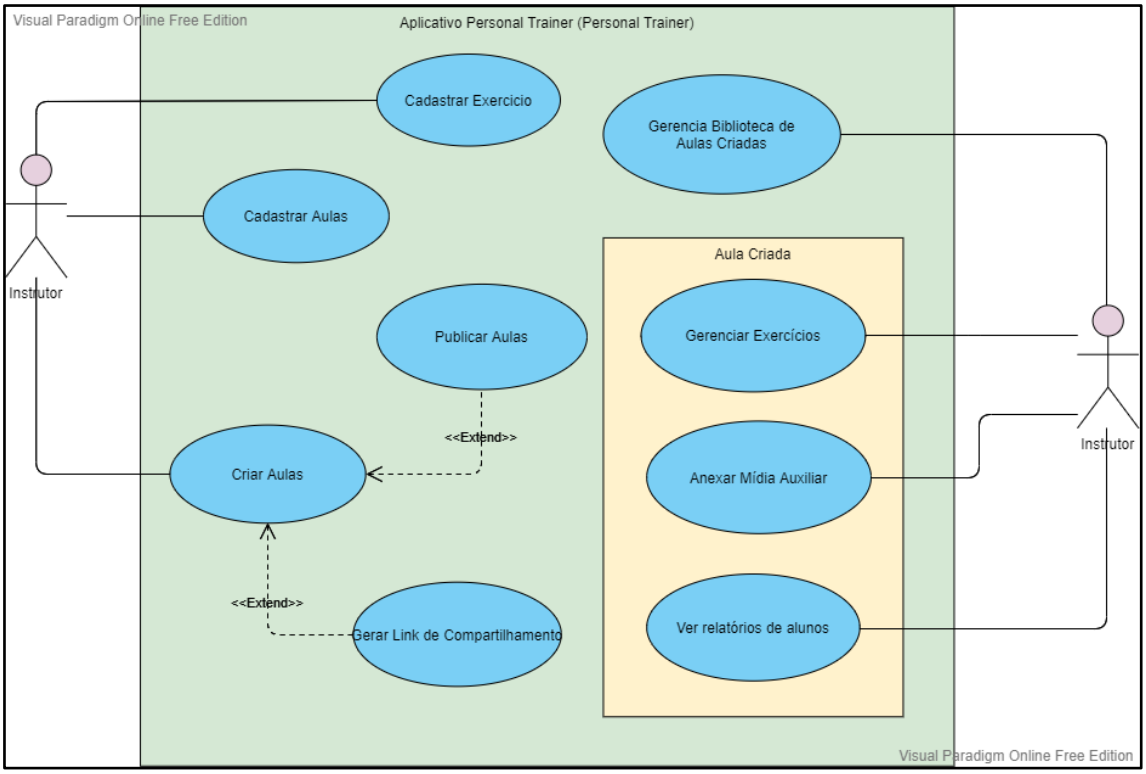

Figura 6: Diagrama de casos de uso em que o ator é o usuário do tipo instrutor.

Outras informações que podem ser incluídas ou alteradas em uma aula, são título, descrição, imagem de capa e a visibilidade, podendo ser pública ou privada. O que diferencia uma aula privada de uma publica, é a forma de um aluno adicioná-la à sua biblioteca pessoal. Sendo que aulas privadas sejam acessadas apenas por quem possui um *link* gerado no momento de criação enquanto aulas públicas aparecerão cronologicamente em uma listagem para todos os usuários do tipo “Aluno”.

3.4. MODELAGEM DO BANCO DE DADOS

O SGBD escolhido foi o PostgreSQL 13, um banco de dados relacional *open source*. A Figura 7 apresenta o diagrama entidade relacionamento, construído utilizando SchemaSpy, uma ferramenta *open source* em Java que analisa a base de dados especificada em qualquer tipo de SGBD e gera uma visualização da estrutura das tabelas.

As tabelas do banco de dados foram criadas dinamicamente conforme necessidade da aplicação, utilizando uma propriedade do *driver* do banco de dados para o Spring. Classes que possuem anotações do *plugin* de persistência serão mapeadas de acordo com o nome

da tabelas especificada, tipo de cada atributo, nome de coluna especificada e relações de chave estrangeira.

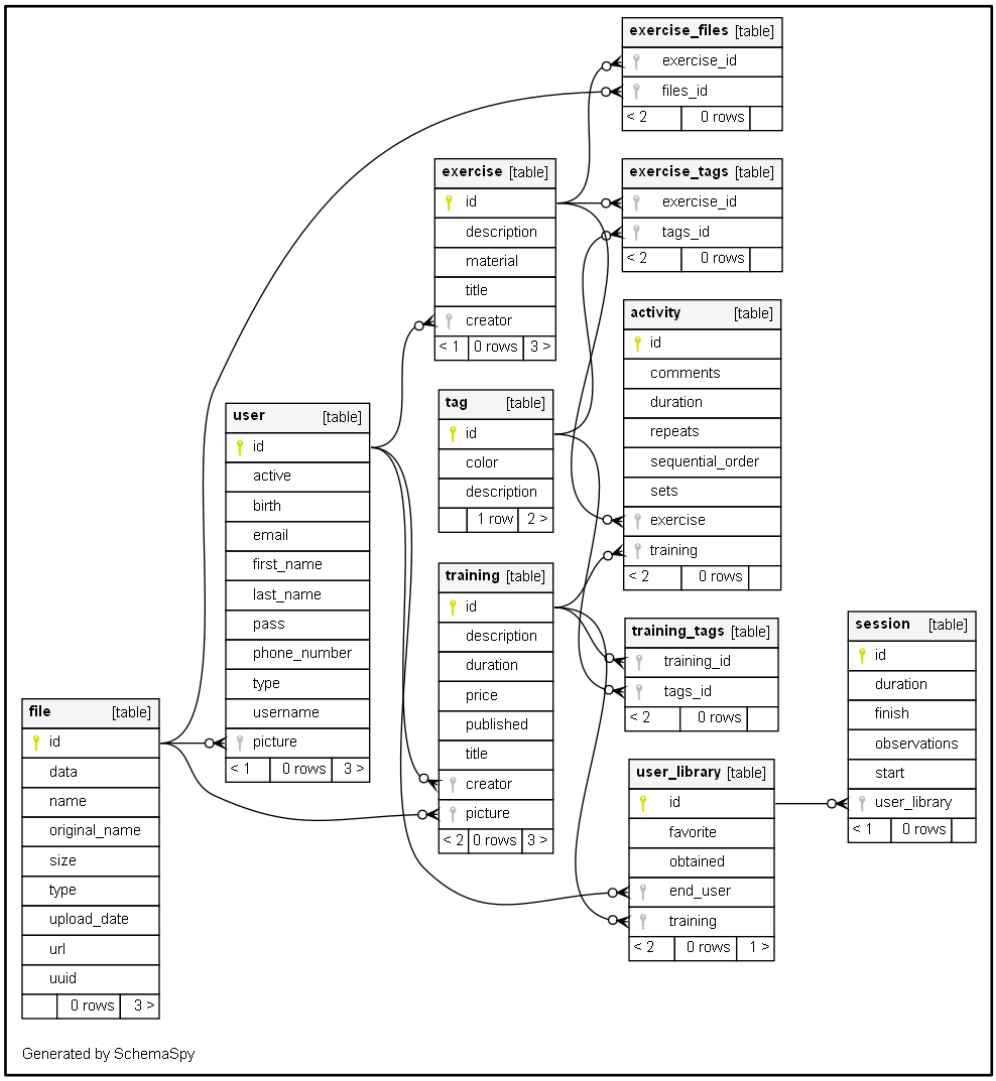

Figura 7: Diagrama entidade relacionamento gerado pelo SchemaSpy.

O resultado da geração são as tabelas de entidade assim como tabelas auxiliares para relações muitos para muitos, como é o caso do *exercise_files*, *exercise_tags* e *training_tags* que não possuem chave primaria e seu único propósito é a integração na aplicação.

Foi decidido que o armazenamento de arquivos ocorre no próprio banco de dados, salvando os *bytes* do arquivo diretamente junto com a entidade. Apesar deste método ser menos complexo, pode comprometer o desempenho futuramente, quando for necessário a

movimentação de múltiplos arquivos simultaneamente, imaginando que o sistema suportará múltiplos usuários que consumirão o serviço.

3.5. MODELAGEM DA API

Para a API foi utilizado Java na JDK 11 juntamente com o *framework* para WebServices, Spring Boot 2. No momento de geração do projeto base com Spring Initializr, foi escolhido o Maven como gerenciador de pacotes. O mesmo é responsável pela a interpretação do arquivo *pom.xml*, que contém a definição das dependências a serem baixadas e especificações do projeto, como configurações ou metadados usados no momento da geração da aplicação Java.

O projeto também conta com bibliotecas de facilitação de desenvolvimento como *Spring DevTools,* uma dependência para apoiar no processo de criação do projeto com configurações padrão para um servidor de aplicação, e facilidades durante o desenvolvimento como o *Live Reload,* a compilação e execução automática ao detectar mudanças no código fonte, de forma que apenas sejam aplicadas apenas as mudanças e não seja necessário compilar todos os arquivos novamente.

Swagger-ui é outra dependência do projeto que percorre o projeto buscando pelas classes contendo controladores de requisições (anotados com *RequestController*) e prepara uma página web que apresentam todos os métodos mapeados como requisições HTTP a partir das classes de *Controller* analisadas.

Nesta página é possível ver algumas informações sobre a aplicação que podem ser configuradas para exibir metadados do projeto como o título, autor, licença e versão. Em outra seção são listados os métodos agrupados por classe, estes métodos podem ser testados pela interface do Swagger-ui, tal que conforme o método mapeado, disponibiliza campos para serem informados parâmetros e corpo da requisição, com detalhes de qual tipo de dado o método retorna, quais são as possíveis respostas da requisição e uma parte dedicada para autorização caso a aplicação possua tal camada de segurança. Por fim é possível consultar todos os modelos que fazem parte de parâmetros ou retorno dos métodos listados.

O Swagger-ui é uma excelente ferramenta para documentação, pois além de coletar dinamicamente os *endpoints* e entidades, e exibir em um alto nível de detalhe, permite o teste funcional dos métodos sem depender de outra aplicação.

Para acelerar o processo de desenvolvimento das classes foi utilizado Lombok, uma dependência que tem como propósito reduzir os códigos *boilerplate* (códigos que são replicados em múltiplas classes, diferenciando-se apenas no contexto). Exemplos de código *bloilerplate* são construtores com ou sem argumentos, *getters* e *setters* de cada propriedade e a estrutura de *builders*, classes que facilitam a instanciação de uma entidade, sendo todos os parâmetros opcionais e não seguem a ordem de um construtor específico.

3.5.1. Funcionalidades

Para as principais entidades mapeadas no banco de dados foram criadas classes de serviço e controladores de requisição. As classes de serviço têm em comum as operações de busca de todos os dados sem filtros, a busca de uma entidade pela sua chave primária (*id*), a exclusão da entidade informando seu *id* e a persistência da entidade, que será utilizado tanto para cadastro como para alteração, sendo a única diferença que no cadastro a entidade inicialmente não possui *id*.

Para algumas entidades os dados são dependentes do usuários que requisita, logo estas terão uma busca específica no banco que filtra os resultados pelo *id* de usuário o qual a entidade referência. Exemplos de classes que implementarão a busca por usuário são: aulas filtradas pro criador, exercícios filtrados por criador e a biblioteca de aulas adquiridas pelo usuário a que a biblioteca pertence.

Outras classes necessitam de funcionalidades específicas com mais tratamento, um exemplo é o *login* do usuário, onde deve ser informado seu *username* e senha. Assim como o cadastro de usuário que deve garantir que o *username* escolhido não esteja em uso.

3.5.2. Diagrama de classes

A Figura 8 apresenta o diagrama de classes das entidades mapeadas do banco de dados para o Java de forma orientada a objetos. Estão presentes apenas as classes que mapeiam

entidades, seus métodos funcionais fazem parte das classes de serviço e controle seguindo o modelo MVC.

No diagrama de classes da aplicação, pode se notar que todas as classes de entidade que serão persistidas no banco de dados, estendem as propriedades da classe *BaseEntity*. Seu propósito é ser uma classe genérica que pode ser persistida, desta forma, métodos também podem ser generalizados.

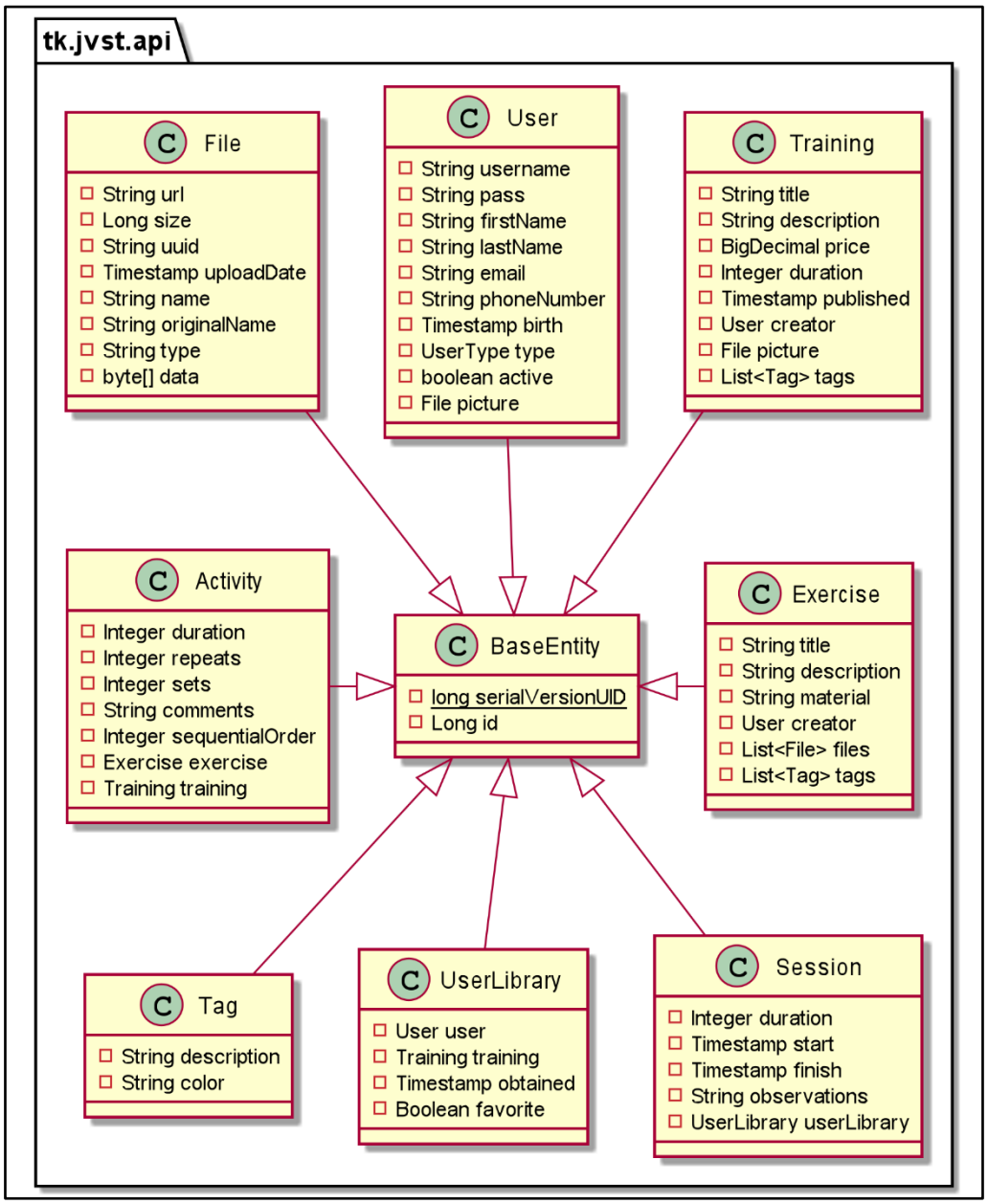

Figura 8: Diagrama de classes da *API*.

3.6. MODELAGEM DO APLICATIVO

O aplicativo foi modelado utilizando os componentes disponíveis no Ionic 6 baseado no Angular 10, o mesmo atenderá os dois tipos de usuários separando as funcionalidades na hora do *login*.

3.6.1. Funcionalidades

Primeiramente as funcionalidades genéricas, que ambos os tipos de usuário têm acesso, são principalmente relacionadas a informações pessoais do usuário, nestas estão incluídas: cadastro, *login*, atualização de dados.

Caso o usuário for do tipo instrutor, suas funcionalidades são limitadas as genéricas, em adição ao gerenciamento de suas aulas, exercícios e relatórios. Estes usuários poderão criar uma biblioteca própria de exercícios que serão usados na elaboração de suas aulas, uma *guideline* a ser seguida, é de generalizar o máximo possível um exercício pois posteriormente será decidido a carga em tempo, repetições ou material individualmente para cada aula que contemplá-lo. Será possível adicionar um vídeo ou imagens como exemplo da execução do exercício para guiar o aluno.

Ainda na parte do instrutor, no gerenciamento de suas aulas criadas, será possível adicionar uma imagem de capa, alterar a visibilidade da aula e informações de metadados. Nesta visão de aulas estão listados os relatórios mais recentes de alunos que as praticam, para avaliação do instrutor.

A visão de um usuário do tipo aluno começa no seu *dashboard*, com informações rápidas de últimas aulas praticadas e algumas estatísticas. O aluno pode usar a pesquisa global para encontrar aulas com base nos termos pesquisados ou *tags* selecionadas, ao abrir a página da aula encontrada o usuário tem a opção de adicionar a biblioteca pessoal. Apenas aulas que foram adicionadas à biblioteca podem ser iniciadas pelo usuário, ao iniciar uma aula, é criado um relatório em que o aluno pode adicionar comentários sobre a sessão praticada e enviá-lo para o instrutor criador da aula.

3.6.2. Proposta de interface gráfica

Em seguida o design de algumas das telas principais que foram implementadas, com o estilo padrão *Material Design* e ideia principal de navegação pelo menu lateral. Estes protótipos foram criados com a ferramenta de prototipação e planejamento de projetos, Whimsical. Uma ferramenta online com recursos gratuitos que apesar de limitados, satisfaz como uma ferramenta para planejar *layouts*.

A pesquisa por aulas poderá ser feita por palavras chaves e utilizando as tags, adicionadas pelo criador da aula para categorizar sua aula com base no seu conteúdo. As tags existentes são exibidas diretamente na página de pesquisa para dar ao aluno uma visão geral e atalhos práticos.

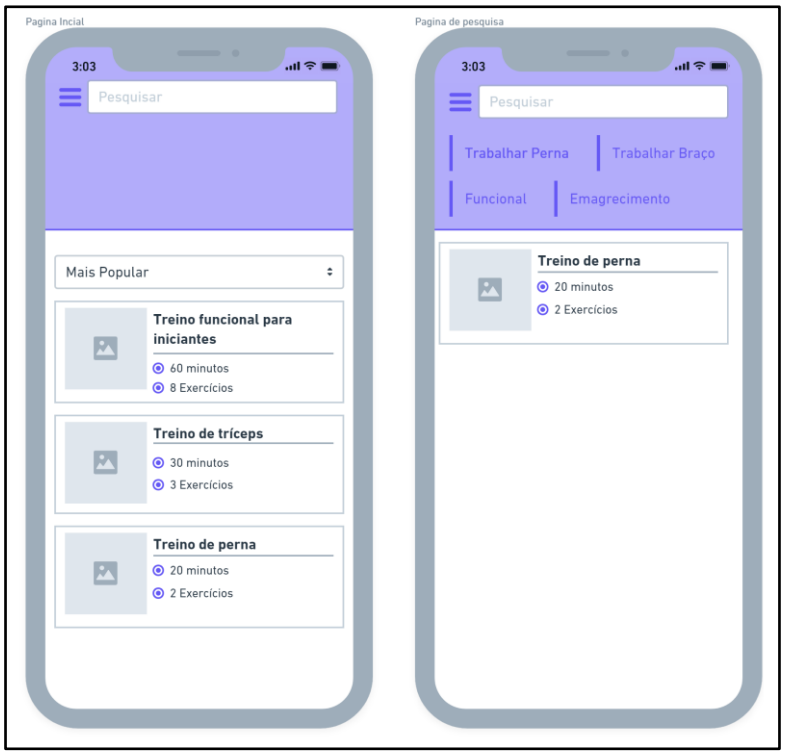

Figura 9: Protótipo da página Inicial e página de busca de aulas com suporte a filtro por *tag*.

A tela inicial da aplicação será a lista de aulas disponíveis, porém haverá abas onde o usuário poderá ir diretamente a sua biblioteca de aulas adquiridas ou seu perfil, onde fica a grande diferença dos dois tipos de usuário. Usuários do tipo instrutor poderão pela sua página de perfil acessar o cadastro de novos conteúdos, como exercícios e aulas. Para ambos os tipos de usuário, esta tela conterá uma opção de edição para seus dados pessoais assim como o *logout.*

A página de perfil de outro usuário pode ser acessada por ambos os tipos de usuário por caminhos diferentes. Para o aluno, através da página de uma aula é possível abrir o perfil do criador, e para o instrutor, nos relatórios de sessões recebidos.

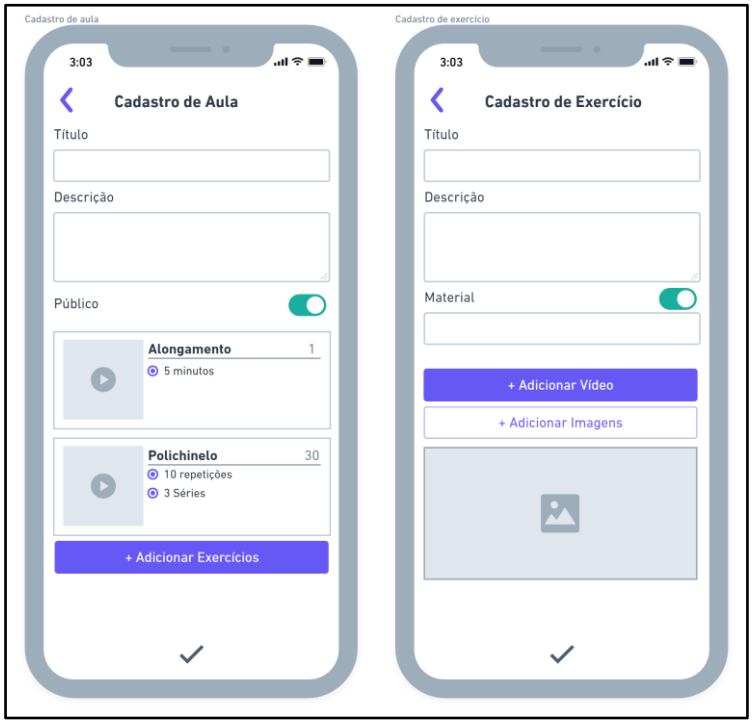

Figura 10: Protótipo da página de cadastro/atualização de aulas e exercícios.

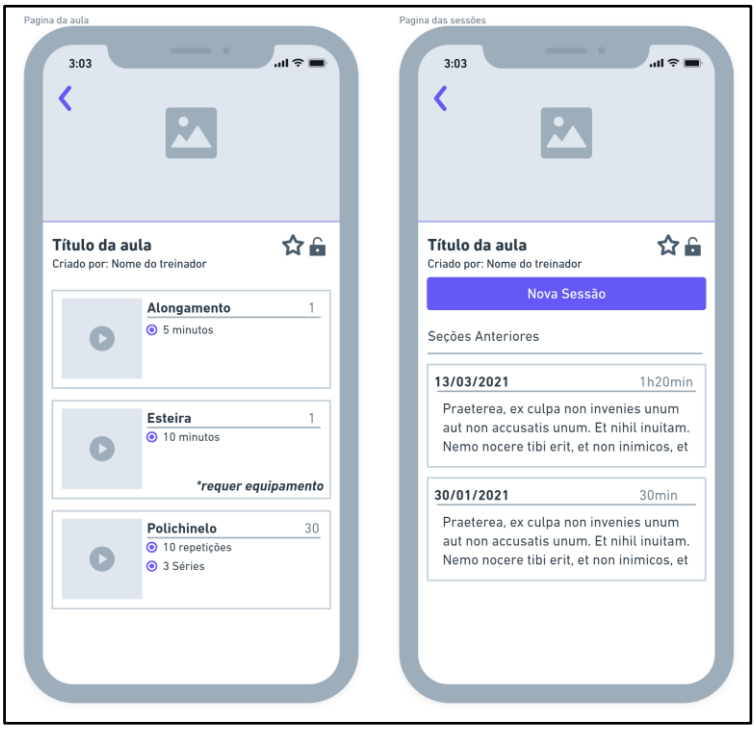

Figura 11: Protótipo da página de detalhes da aula na visão do aluno e página de sessões de uma aula.

4. DESENVOLVIMENTO

No processo de desenvolvimento, foi seguida a ordem de dependência de cada parte da aplicação, começando pela configuração do banco de dados, seguido da criação dos projetos e repositórios para versionamento. Nesta seção serão apresentados pontos de destaque das tecnologias utilizadas e por fim as interfaces que satisfazem os casos de uso propostos.

4.1. Desenvolvimento da API

A estrutura do projeto foi organizada de forma que cada pacote contém todas as classes diretamente relacionadas com tal entidade, com exceção dos pacotes: *configuration, generic* e *util*.

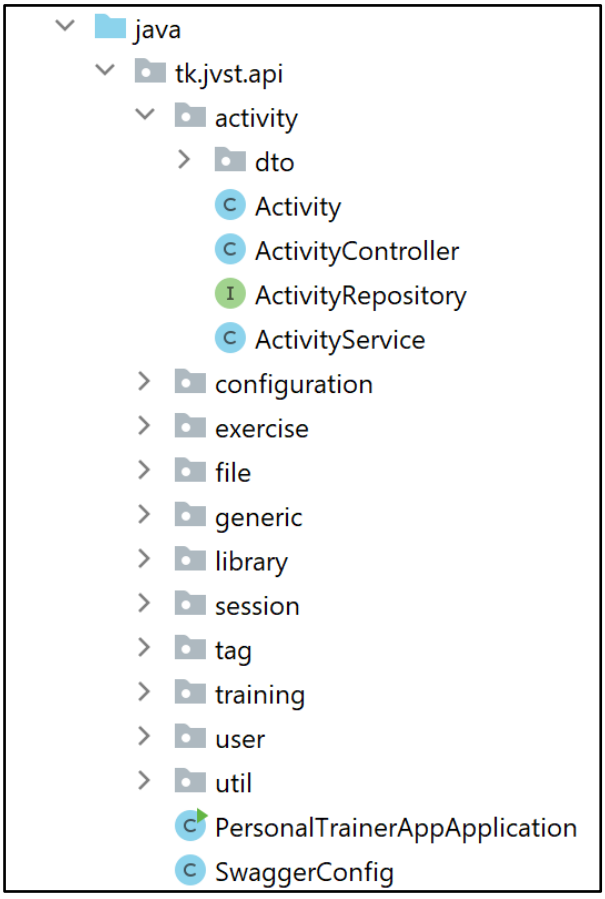

Figura 12: Estrutura de arquivos do projeto da API.

Em cada pacote de entidade, estão presentes as classes: modelo, controlador de requisições, repositório de dados, e serviços respectivamente, como apresentado na Figura 12. Ocasionalmente haverá um pacote para DTOs (Objetos de Transferência de Dados).

O ponto de partida da modelagem das classes, foi a definição de um entidade base. Esta entidade serviu como uma garantia que todas as classes que herdam suas propriedades podem ser persistidas. Seu único atributo acessível é o *id,* e suas anotações indicam que este campo é o atributo identificador com geração automática da entidade, ou neste caso, a entidade que herdará da classe *BaseEntity*.

BaseEntity.java

```java
package tk.jvst.api.generic;

import ...

@NoArgsConstructor
@MappedSuperclass
@SuperBuilder
public class BaseEntity implements Serializable {

    private static final long serialVersionUID = 1L;

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

    public Long getId() { return id; }

    public void setId(Long id) { this.id = id; }

}
```

Figura 13: Classe *BaseEntity.java.*

Desta forma é possível fazer uma generalização de todas as classes que podem ser persistidas e criar serviços que aceitam somente tais classes. Na Figura 14, a classe apresentada é a *BaseService*, uma classe de serviços que utiliza do recurso *Generics* da linguagem Java. A classe é abstrata e a implementação depende do elemento *T*, onde *T* é uma classe que estende de *BaseEntity*.

Por se tratar de métodos genéricos, não é possível a implementação de regra de negócio de entidades específicas nesta camada. Ao desenvolver as classes que herdam da *BaseService,* foi possível sobrescrever os métodos conforme cada serviço necessitasse.

BaseService.java

```java
package tk.jvst.api.generic;

import ...

public abstract class BaseService<T extends BaseEntity> {

    private BaseRepository<T> repository;

    protected BaseService(BaseRepository<T> repository) { this.repository = repository; }

    public List<T> findAll() { return repository.findAll(); }

    public T findById(Long id) {
        Optional<T> entity = repository.findById(id);
        if (entity.isEmpty()) {
            throw new EmptyResultDataAccessException(1);
        }
        return entity.get();
    }

    public List<T> saveAll(List<T> entities) { return entities.stream().map(this::save).collect(Collectors.toList()); }

    public T save(T entity) {
        return repository.save(preProcess(entity));
    }

    public void deleteById(Long id) { repository.deleteById(id); }

    public T preProcess(T obj) { return obj; }
}
```

Figura 14: Classe BaseService.java.

O método “*preProcess”* é responsável por realizar a inserção de dados no repositório e seu propósito é ser uma chamada de validação de dados que será sobrescrita com as regras de cada entidade no momento da herança.

Os modelos da aplicação são estruturados de forma orientada a objeto, representado à esquerda na Figura 15, e se integram com as bibliotecas de persistência para que no banco de dados, sejam recebidos na estrutura relacional. Porém ao receber requisições externas não é garantido a integridade da estrutura de dados, por isto, foi implementado para os modelos mais complexos uma classe DTO.

Esta classe, representada a direita na Figura 15, não precisa conter exatamente os mesmos campos e tipos de dados em contraste com o modelo original que deve ser o mais próximo da base de dados o possível. O propósito dos DTOs é omitir ou acrescentar dados para uso em uma requisição, isto também garante um nível a mais de segurança e integridade dos dados, uma vez passarão por um processamento antes de serem enviados ou ao serem recebidos. Este tipo de classe geralmente vai conter valores mais próximos dos primitivos *string, number* e *boolean*, tipos de dados suportados pelo padrão JSON, reduzindo até a relação de uma entidade a seu *id* para que possa ser buscada pela aplicação diretamente do repositório original.

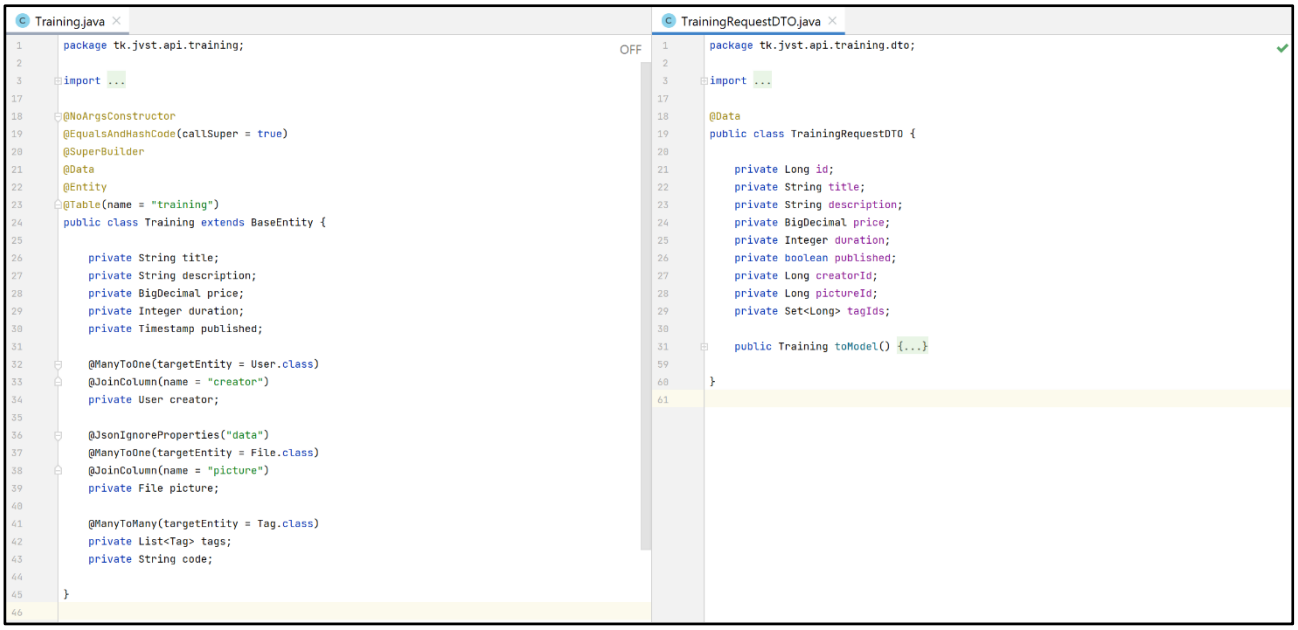

Figura 15: Comparação de *model* e DTO.

O controlador é responsável por receber e lidar com requisições enviadas para o *endpoint* para qual ele está mapeado. Na Figura 16, como exemplo, foram implementados os serviços relativos aos exercícios, que aguardarão serem acionados quando a URL da aplicação for seguida de “/exercise” com um método *get* ou *post.* Para uma requisição *get* a este serviço, é necessário informar o *id* do exercício para que sejam retornadas as informações completas do mesmo, ou acrescentando “/creator” ao *path*, deve ser informado um *id* de usuário e será retornado todos os exercícios cadastrados pelo mesmo.

ExerciseController.java

```java
package tk.jvst.api.exercise;

import ...

@Api(value = "Exercise", tags = {"Exercise"})
@RestController
@RequestMapping("/exercise")
public class ExerciseController {

    @Autowired
    private ExerciseService service;

    @GetMapping("/creator")
    public ResponseEntity<List<Exercise>> findAllExercisesByCreator(@RequestParam Long creatorId) {
        List<Exercise> exercises = service.findAllExercisesByCreator(creatorId);
        return ResponseEntity.ok(exercises);
    }

    @GetMapping
    public ResponseEntity<Exercise> findExerciseById(@RequestParam Long exerciseId) {
        Exercise exercise = service.findById(exerciseId);
        return ResponseEntity.ok(exercise);
    }

    @PostMapping
    public ResponseEntity<Exercise> persistExercise(@RequestBody ExerciseRequestDTO exerciseRequestDTO) {
        Exercise exercise = service.persistExercise(exerciseRequestDTO);
        return ResponseEntity.status(HttpStatus.CREATED).body(exercise);
    }

}
```

Figura 16: Classe ExerciseController.java.

No método *post*, a requisição deverá conter um corpo com o conteúdo no formato JSON respeitando o objeto DTO especificado para cadastro ou atualização de um exercício. A diferenciação das duas ações e feita pela validação do *id* da entidade, que caso não presente, indica que este é um novo registro, e nos casos em que o id está presente, e corresponde a uma entidade existente, tal entidade é recuperada e seus dados atualizados.

Os parâmetros dos métodos *get*, anotados com *RequestParam*, serão informados como parte da URL a partir de seu *endpoint*. No método *post*, é necessário que a requisição possua como corpo um objeto JSON que deverá ser mapeado para a classe *ExerciseRequestDTO*.

Com a codificação dos serviços concluídos, foi possível testa-los de maneira prática a partir da interface do Swagger-ui. A configuração padrão desta biblioteca analisa o projeto e

encontra todas as classes com a anotação *ResquestController,* criando grupos baseados em tais classes, cada um contendo os *endpoints* mapeados pelos métodos HTTP.

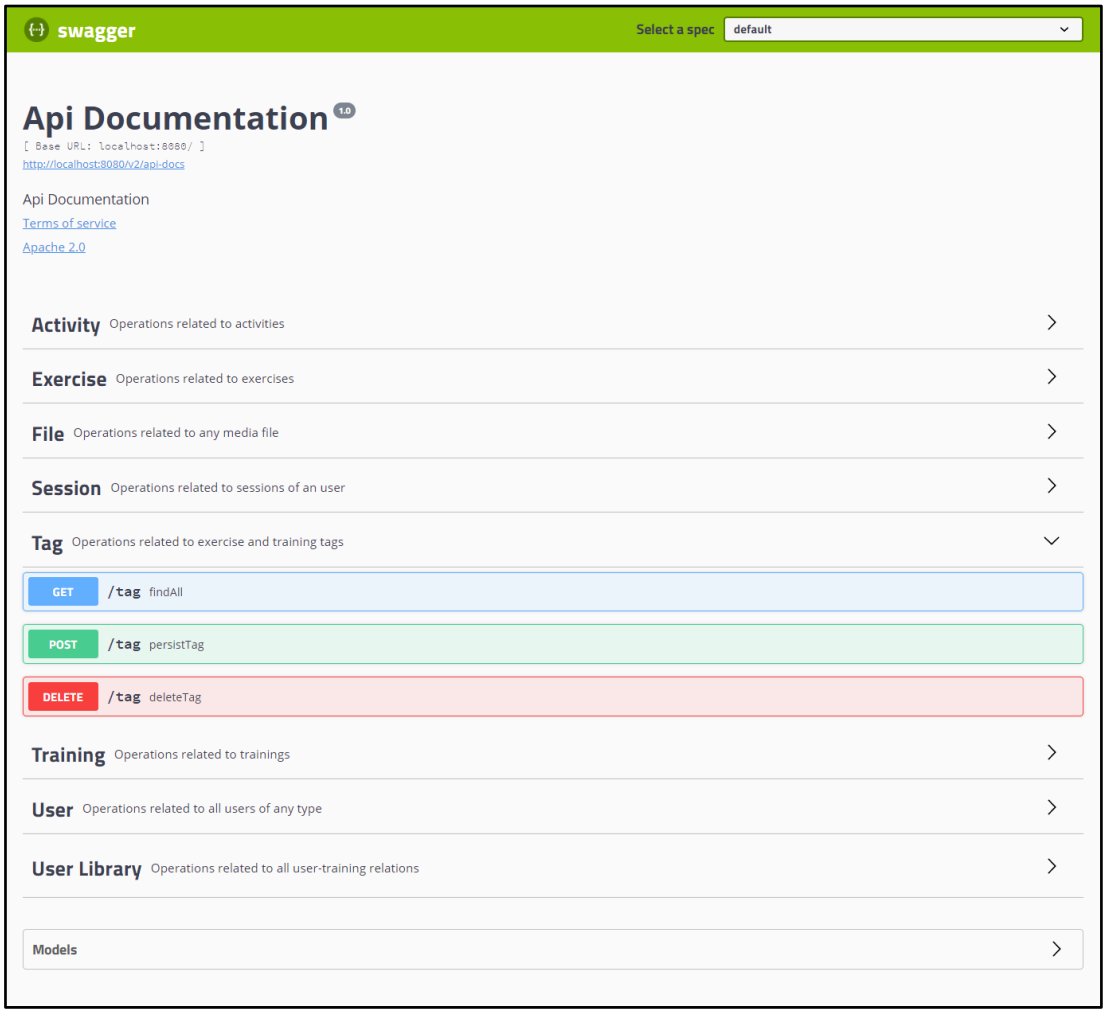

Figura 17: Interface do Swagger-ui.

Por fim, e gerado o arquivo compilado da aplicação que será executado em uma máquina virtual Java. Esta maquina foi configurada em um ambiente de virtualização, onde outras maquinas com propósitos diferentes também são executadas paralelamente, cada instancia é denominada um *container*. O *software* Docker foi utilizado para o gerenciamento destes *containers*, iniciado por um arquivo de configuração responsável por instanciar *tempates* de aplicações, sendo elas, o banco de dados Postgres, a ferramenta PgAdmin que atua como interface visual para o gerenciamento deste banco de dados, e a aplicação Java.

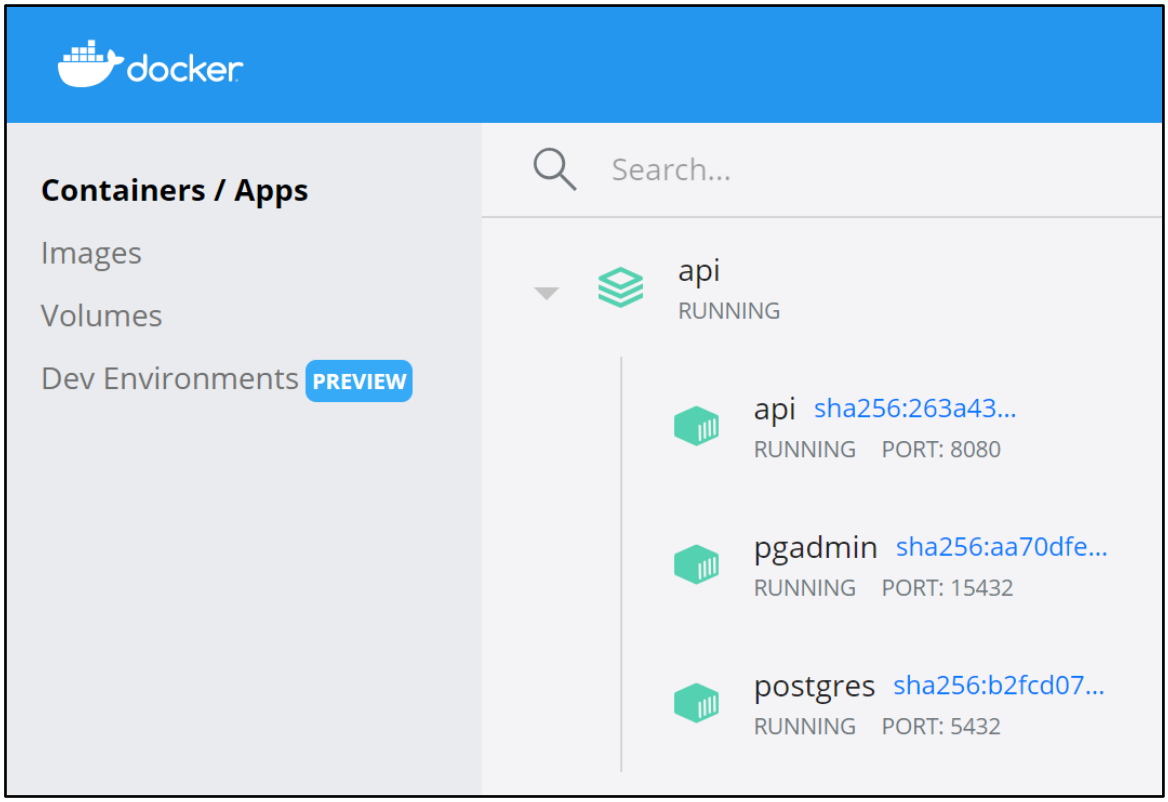

Figura 18: Containers Docker utilizados.

Os *containers* funcionam em uma rede interna da aplicação Docker, e em sua definição deve ser especificada qual porta interna e externa devem ser usadas. A porta interna será usada para a comunicação entre *containers,* enquanto a externa é a porta da máquina física que hospeda a aplicação Docker. O uso de *containers* é uma forma de padronizar o ambiente de execução em diferentes maquinas assim como serialização de instancias da aplicação permitindo escalonamento.

4.2. Desenvolvimento do Aplicativo Móvel

O aplicativo em Ionic foi iniciado a partir do comando “ionic start trainer-app tabs --type=ionic-angular”. Parte da CLI (Interface de linha de comando) do Ionic, tem como função a geração da estrutura de arquivos para uma aplicação Ionic, o parâmetro *trainer-app* é o nome inicial escolhido para o projeto. O próximo parâmetro, *tabs* indica que o projeto iniciará com a demonstração do componente de abas, este componente foi aproveitado na criação da página inicial. Por fim foi especificado o tipo desta aplicação como Ionic com integração ao Angular.

Neste modelo, os arquivos com o prefixo “app” fazem parte do componente principal que iniciará a aplicação e carregará as configurações de módulos. Os pacotes do projeto da aplicação foram organizados de forma diferente da API. Classes foram dividias entre modelos, serviços e páginas, enquanto os diretórios *config* e *util* possuem arquivos e classes com funções utilizadas por todo o projeto. O diretório *service* contêm classes que utilizam do módulo de cliente HTTP do Angular, cada uma destas pode ser associada ao controlador da API de sua respectiva entidade.

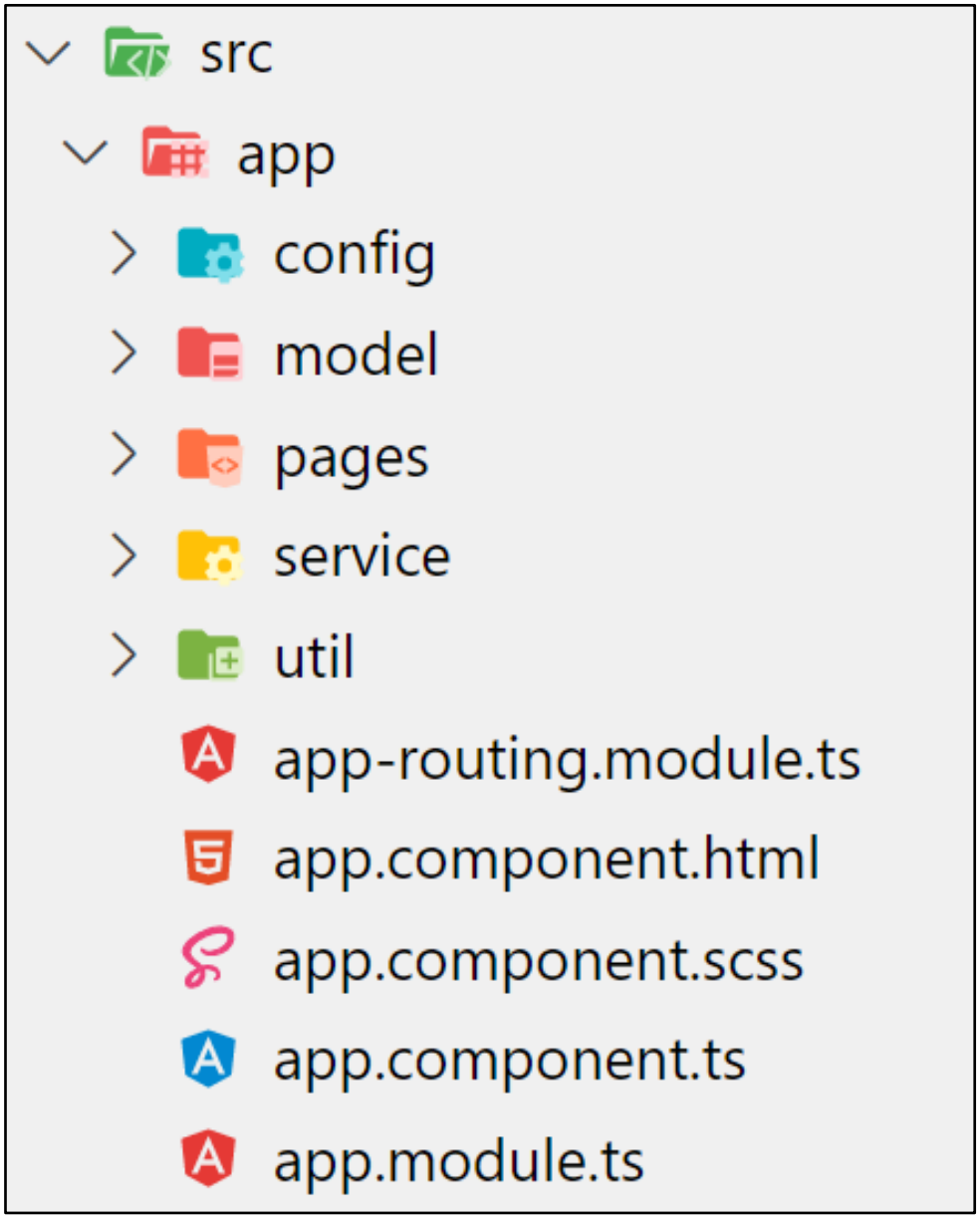

Figura 19: Estrutura do projeto Ionic.

O diretório *pages* contém todos os módulos que definem páginas completas, como uma tela de cadastro ou um *dashboard*. Alguns módulos deste diretório contém a especificação de dois ou mais componentes para seu contexto, e com o recurso de roteamento do Angular é possível redirecionar o usuário ao módulo que for necessário.

O Ionic possui serviços internos de utilidade que foram usados em toda a aplicação, porém os parâmetros usados em alguns métodos, eram repetidos com frequência em diferentes classes. Para que fosse mantido o padrão no uso destes serviços, algumas classes auxiliares foram implementadas com a chamada destes métodos utilizando parâmetros padronizado, algo que também contribuiu na redução de complexidade em componentes que utilizavam dos serviços. Na Figura 19, a classe *LoadingService* é um exemplo de um serviço padronizado. Seu propósito é exibir e esconder uma mensagem de carregamento durante um processo assíncrono.

loading.service.ts

src > app > service > loading.service.ts > ...

```
You, a week ago | 1 author (You)
1  import { Injectable } from '@angular/core';
2  import { LoadingController } from '@ionic/angular';
3  import { Literals } from 'src/app/util/literal-util';
4
   You, a week ago | 1 author (You)
5  @Injectable({
6      providedIn: 'root'
7  })
8  export class LoadingService {
9      public literals = Literals.getLiterals();
10     public loadingStack: any[] = [];
11
12     constructor(private loadingController: LoadingController) {}
13
14     public async show() {
15         const loading = await this.loadingController.create({
16             message: this.literals.common.wait,
17             translucent: true
18         });
19         this.loadingStack.push(loading);
20         await loading.present();
21     }
22
23     public hide(delay: number = 100): void {
24         setTimeout(() => this.loadingStack.pop().dismiss(), delay);
25     }
26
27     public hideAll() {
28         setTimeout(() => {
29             this.loadingStack.forEach((modal) => {
30                 this.hide(0);
31             });
32         }, 1000);
33     }
34 }
35
```

Figura 20: Classe *loading.service.ts*.

Para a exibição de textos na tela e mensagens de confirmação, foi necessário adotar a prática de utilizar um arquivo de termos. As vantagens desta técnica são a padronização de textos estáticos, uma manutenção mais fácil por ser concentrado em um único arquivo e a facilidade ao internacionalizar a aplicação, com um outro arquivo na mesma estrutura para outro idioma e uma classe para gerenciar qual idioma deve ser exibido. A aplicação deste trabalho pode ser visualizada nos idiomas português e inglês, a escolha é feita a partir do idioma do aparelho do usuário.

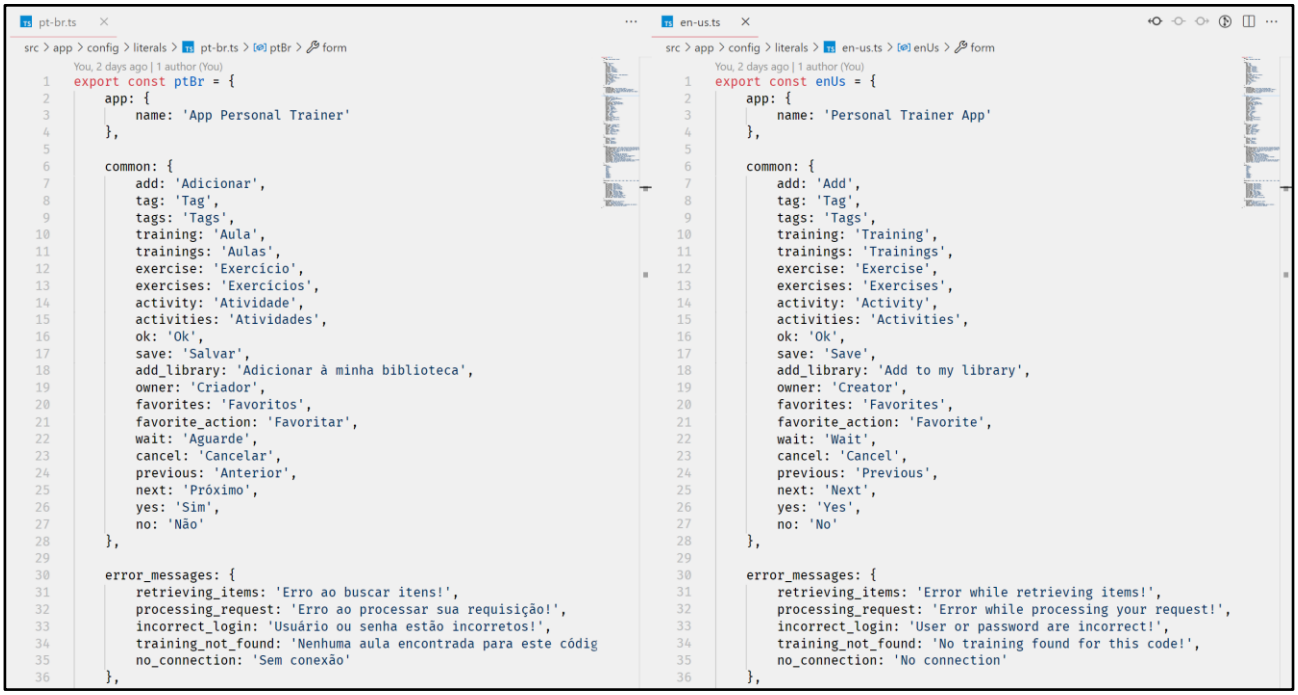

Figura 21: Arquivos de idiomas.

A comunicação entre as duas aplicações é possível devido ao módulo de cliente HTTP oferecido pelo *framework* Angular. Uma instancia deste módulo, disponibiliza funções para cada método HTTP, e para toda requisição é necessário informar a URL a ser utilizada. Métodos como *post* e *put*, requerem um parâmetro adicional que será enviado como o corpo da requisição. Opcionalmente é possível informar o tipo de dado que é esperado na resposta para uma maior garantia de integridade na utilização do serviço implementado.

A Figura 22 é um exemplo de serviço implementado para a entidade *exercise*, é possível comparar cada função implementada com seu respectivo *endpoint* da API exibido na Figura 16.

exercise.service.ts

src > app > service > exercise.service.ts > ...

```
You, a month ago | 1 author (You)
import { HttpClient } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { Exercise } from '../model/exercise/exercise.model';
import { ExerciseRequestDTO } from './../model/exercise/exercise-request-dto.model';
import { ServerService } from './server.service';

You, a month ago | 1 author (You)
@Injectable({
    providedIn: 'root'
})
export class ExerciseService {
    constructor(private serverService: ServerService, private http: HttpClient) {}

    private url: string = `${this.serverService.getServer()}/exercise`;

    public findById(exerciseId: number): Observable<Exercise> {
        return this.http.get<Exercise>(`${this.url}?exerciseId=${exerciseId}`);
    }

    public findAllByCreator(creatorId: number): Observable<Exercise[]> {
        return this.http.get<Exercise[]>(`${this.url}/creator?creatorId=${creatorId}`);
    }

    public persistExercise(exerciseRequestDTO: ExerciseRequestDTO): Observable<Exercise> {
        return this.http.post<Exercise>(`${this.url}`, exerciseRequestDTO);
    }
}
```

Figura 22: Classe *ExerciseService*.

Para evitar a repetição da URL do servidor e do controlador de entidades, cada serviço inicia uma variável que acrescenta seu respectivo caminho à URL base, a qual também é centralizada por um serviço.

Na fase de compilação do aplicativo foi necessário utilizar a IDE Android Studio, com a SDK do Android para a geração do APK, instalador da aplicação para dispositivos Android. A exportação para aparelhos iOS, depende da IDE Xcode, exclusivo para computadores da Apple que operam no sistema macOS.

5. RESULTADOS

Serão exibidas algumas das interfaces desenvolvidas exemplificando os casos de uso mencionados no capítulo da proposta da aplicação. Imagens usadas nas capturas de tela como *placeholder* são originadas de uma API de código aberto, Picsum.

5.1. Funcionalidades Implementadas

Após a instalação e execução do aplicativo, são apresentadas na tela as opções referentes ao *login*, sendo a ação padrão a autenticação do usuário existente. Novos usuários devem escolher a opção “Cadastre-se” e inserir as informações iniciais, vale ressaltar que o nome de usuário deve ser único, e a existência de registros com o mesmo valor serão verificados no processamento da requisição cadastro.

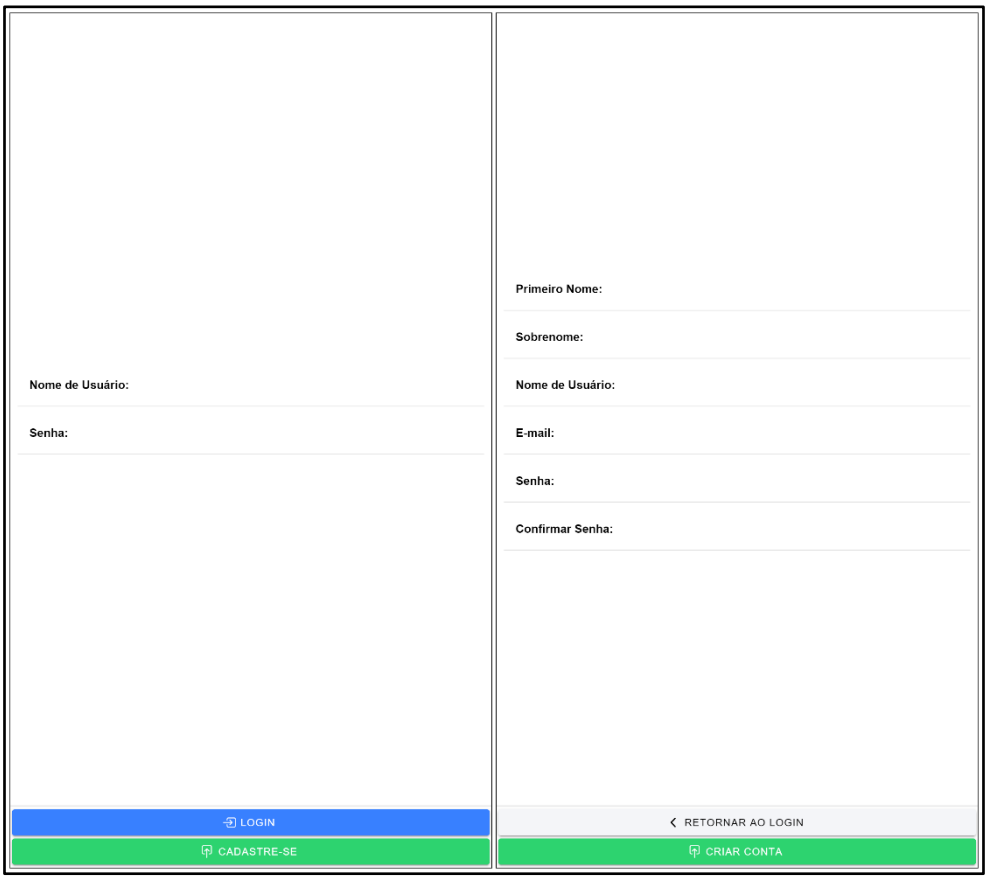

Figura 23: Tela de *login* e cadastro de usuário, respectivamente.

Após a autenticação do usuário, suas credenciais são salvas localmente no dispositivo, e serão verificadas ao tentar acessar as funções disponíveis a seu respectivo tipo de usuário. Estas credenciais permanecerão validas até o usuário não limpar os dados do aplicativo em seu aparelho ou selecionar a opção de *logout*, “Sair”.

A Figura 24 apresenta a principal diferença entre os tipos de usuário. A esquerda está o usuário “Aluno” com a maioria de suas funções nas abas “Explorar” e “Biblioteca”, restando apenas a opção de *logout* e listagem de sessões.

O usuário “Instrutor” possui as opções que o levarão ao cadastro de exercícios, aulas e *tags*, representado à direita na Figura 23. *Tags* podem ser usadas para categorizar tanto aulas quanto exercícios e servirá como uma palavra chave na pesquisa por aulas.

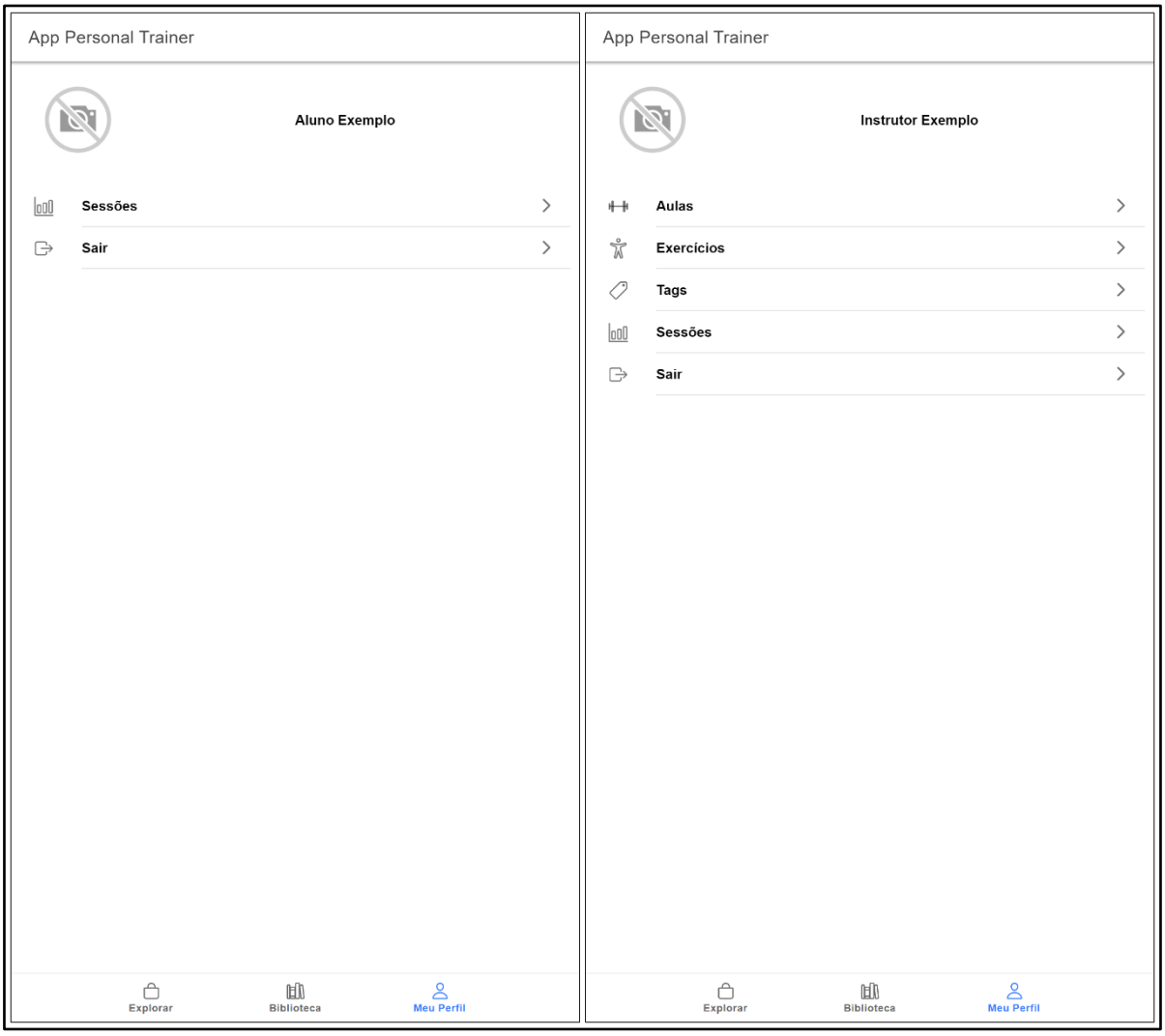

Figura 24: Opções da página de perfil.

5.2. Funcionalidades para o Instrutor

O fluxo principal do usuário instrutor são os cadastros de exercícios e aulas. Cada item de seu menu, redireciona o usuário para a listagem de todos os itens já cadastrados por ele, assim como, um botão para acessar a tela de cadastro para a inclusão de um novo item. A alteração do item pode ser feita pressionando-o a partir de sua listagem.

O primeiro cadastro a ser feito pelo usuário deve ser o de um exercício, onde deve ser informado, obrigatoriamente, um título, e opcionalmente uma descrição e o uso de materiais (caso haja). Pela mesma interface é possível incluir imagens e vídeos ao exercícios, utilizando a integração com o componente para selecionar arquivos nativo de cada dispositivo. Há também um botão que tem a função de abrir um *modal* de seleção de *tags*, que servirão para facilmente categorizar o exercício e atuará como palavra chave para busca de aulas em uma pesquisa.

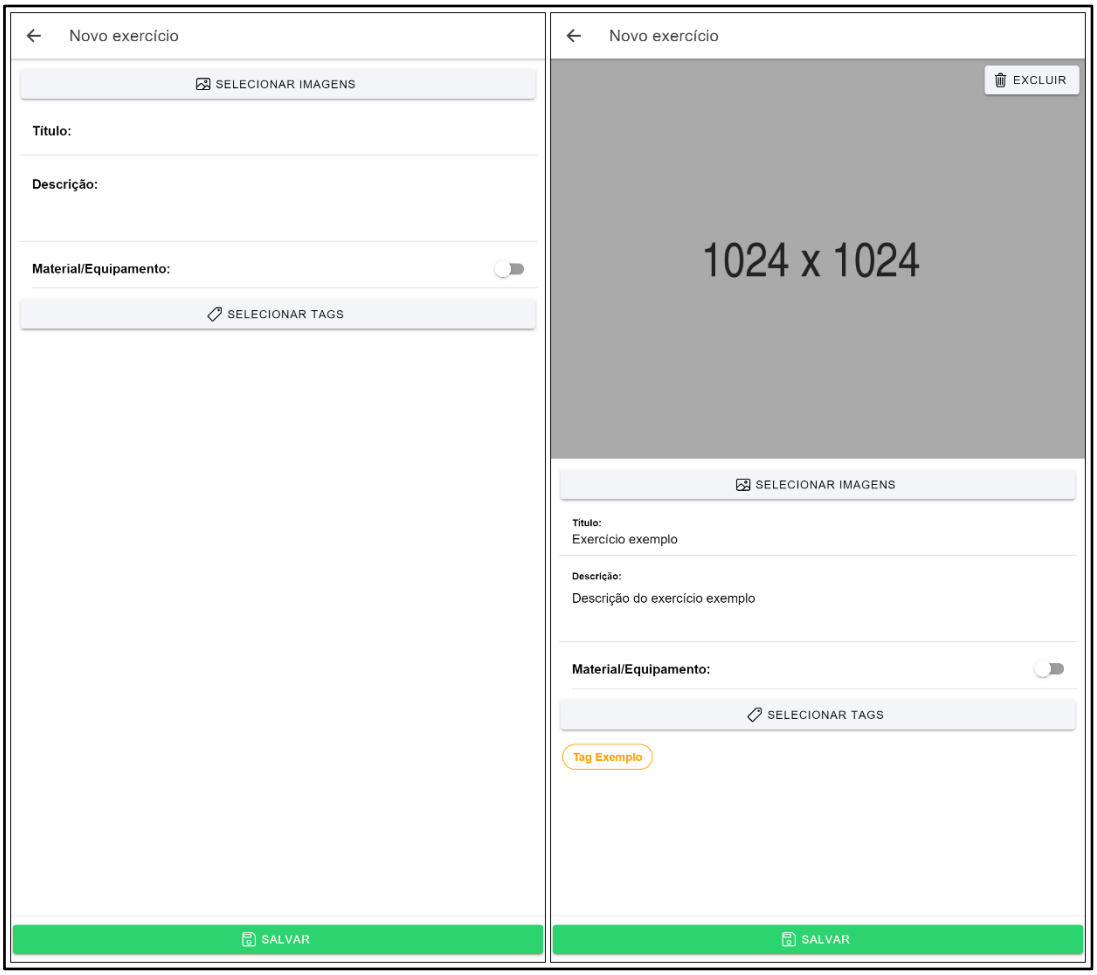

Figura 25: Telas de cadastro de exercícios e exemplo com os campos preenchidos.

O próximo item é o cadastro de aulas, que ao ser acessado inicialmente, não estará disponível a inclusão de atividades, e é solicitado que a aula seja cadastrada primeiro. Isto é necessário pois o registro de uma atividade depende da chave primária de uma aula para ser válida e a partir do momento que a aula é cadastrada, já é possível utilizar seu *id* para as próximas etapas. Assim como para um exercício, o único requisito para registrar uma aula é seu título, e as informações opcionais são sua imagem de capa, descrição, *tags*, atividades e o acionamento da publicação.

A imagem de capa é um elemento estético no contexto do cadastro de aulas, pois seu único propósito é a diferenciação no momento de visualização na listagem e na biblioteca do aluno.

A opção de publicar indica se, ao salvar uma aula, a mesma deverá imediatamente ser disponibilizada para o acesso global ou não. No estado inicial da rotina, a publicação é desativada para evitar divulgações acidentais ou que não contenham atividades. Para todas as aulas será gerado um código de seis dígitos hexadecimais, que pode ser usado para o compartilhamento de aulas não publicadas com outros usuários.

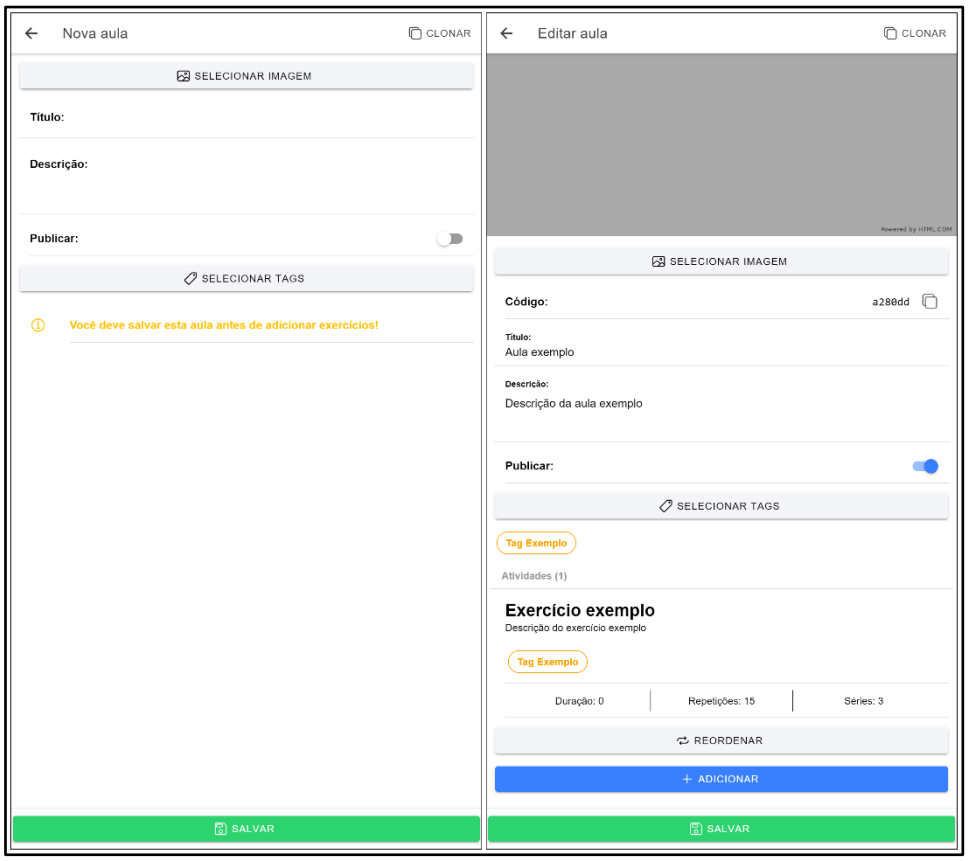

Figura 26: Tela de cadastro de aulas e exemplo com os campos preenchidos.

Após o registro da aula, é liberado o cadastro e gerenciamento de atividades de uma aula. Atividade foi o nome escolhido para ser o intermediário entre o exercício e a aula, contendo informações que incrementam o exercício com especificações variáveis. O cadastro de atividades contempla a seleção de um único exercício por atividade e permite habilitar ou desabilitar três atributos que podem ou não fazer parte do contexto do exercício.

Estes atributos, são a duração, repetições e séries, e podem ser usadas ao mesmo tempo ou individualmente, pois é comum para exercícios contáveis, como abdominais, serem contados em repetições e séries, enquanto os que se baseiam em resistência são delimitados por tempo. É possível também a adição de comentários à atividade no caso da necessidade de informar o aluno parâmetros que não se aplicam a contagem oferecida.

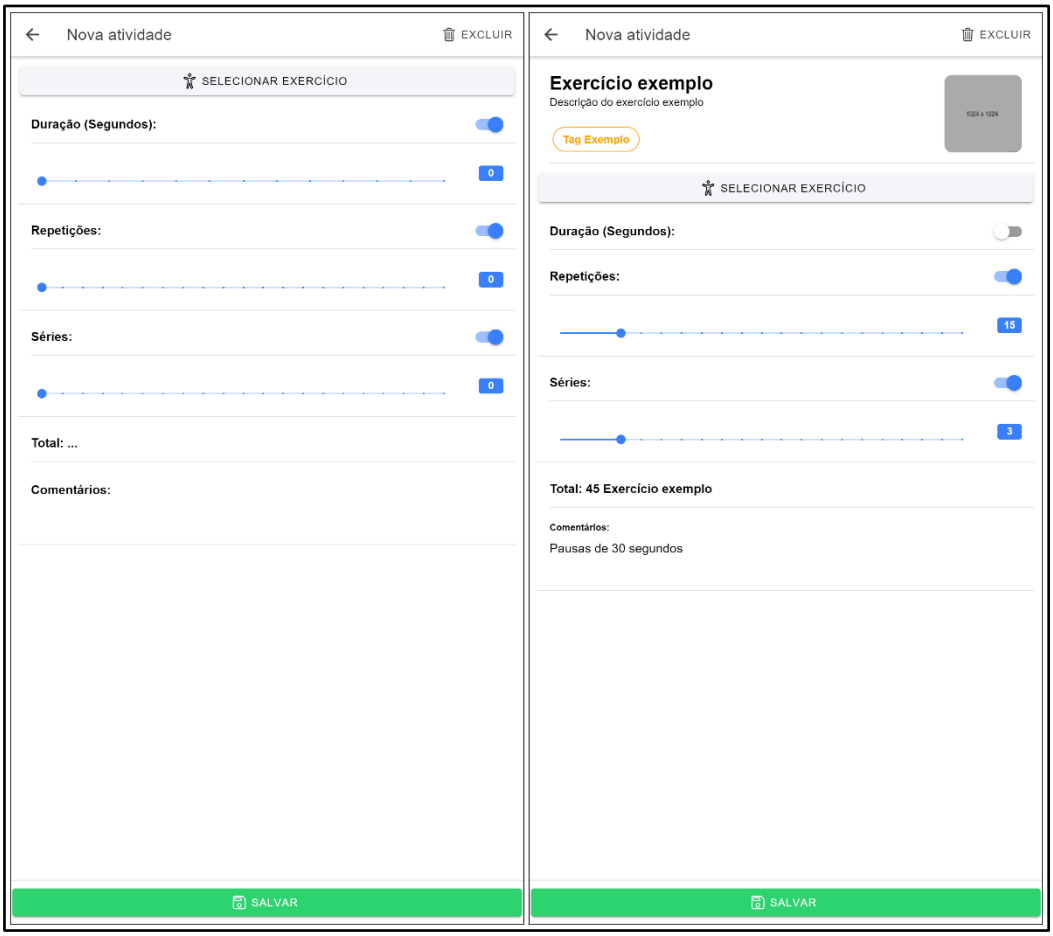

Figura 27: Tela de cadastro de atividades e exemplo com os campos preenchidos.

O usuário pode a qualquer momento ativar a opção de publicar a aula criada, para que seja visível a todos os usuários de tipo aluno. A validação de cadastro e atualização de aulas,

verifica o atributo de publicação e caso verdadeiro adiciona a data e hora do sistema como data de publicação. A listagem conta com um resumo da aula em um componente de *card*, dando ênfase na capa e exibindo informações básicas como título, data da publicação e *tags*.

5.3. Funcionalidades para o Aluno

A funcionalidade principal do aluno se localiza nas duas primeiras abas de sua tela inicial. Primeiramente, “Explorar”, carrega uma lista de aulas publicadas em ordem de data de publicação decrescente. Ao acessar uma aula por esta listagem, é possível ver detalhes como título, descrição e quantidade de exercícios, porém o detalhe dos exercícios como contagem, fotos e vídeos só podem ser visto ao acessá-la na aba “Biblioteca”. Para isso o aluno deverá primeiro acionar o processo de aquisição da aula, que no momento não possui restrições.

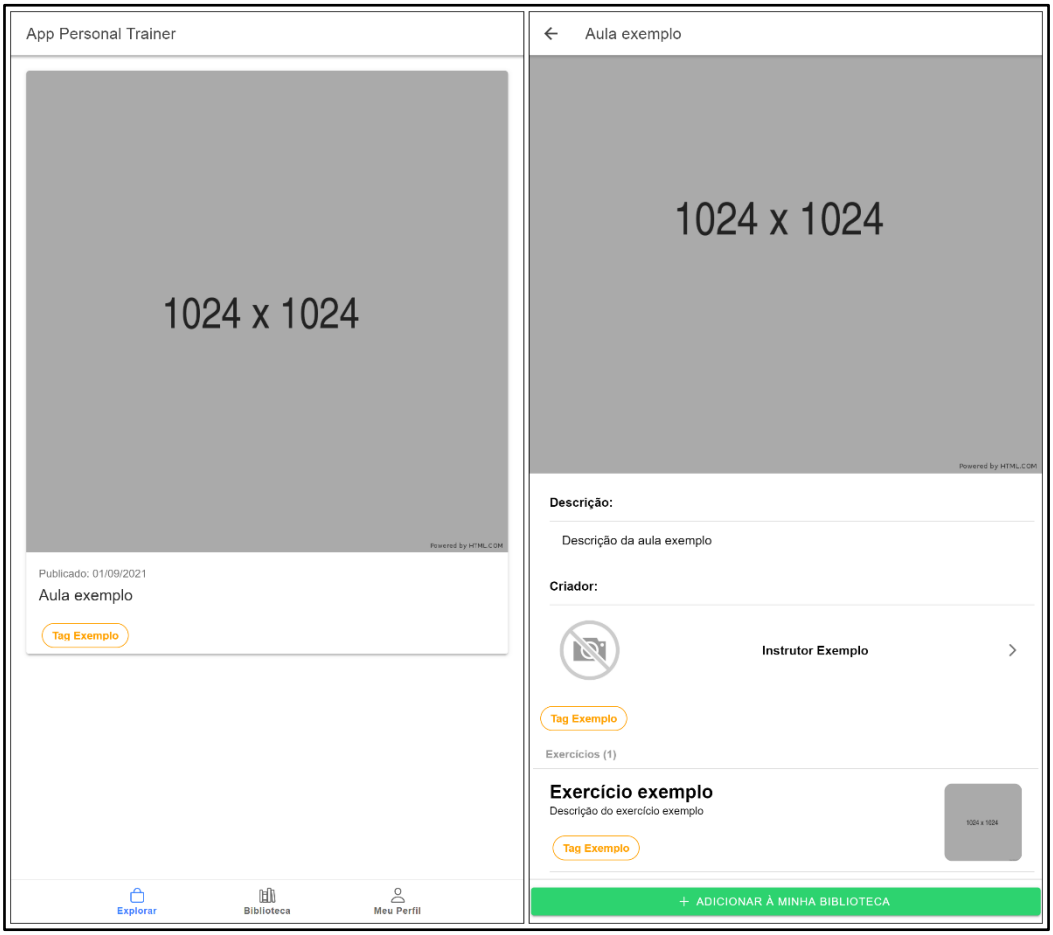

Figura 28: Listagem de aulas e página de detalhes antes da aquisição.

A página de uma aula aberta a partir da biblioteca, tem de diferencial, a opção de adicionar a aula aos favoritos, fazendo com que a mesma seja exibida no topo da biblioteca. Outra opção é a visualização detalhada do exercício ao pressioná-lo.

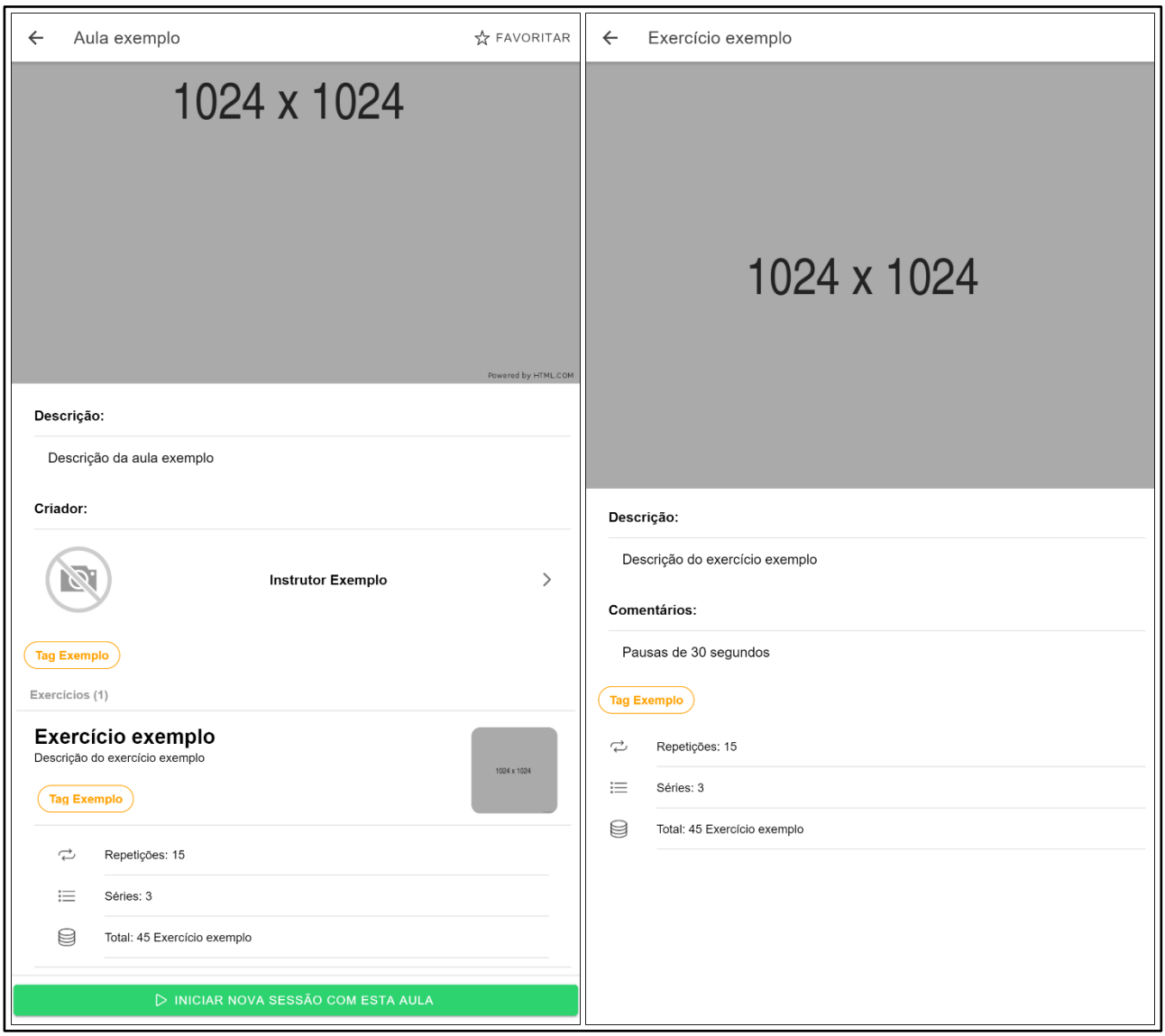

Figura 29: Página da aula na biblioteca e detalhamento do exercício.

A página de detalhamento do exercício é a forma em que o aluno vê as especificações do exercícios em uma aula, sendo elas: repetições, séries, duração, materiais necessários (caso haja), comentários sobre o exercício (caso haja) e todas as mídias vinculadas.

Para a aquisição de aulas não publicadas, na aba “Biblioteca” encontra-se um botão para a inserção do código de uma aula. Após a validação do código, o usuário é redirecionado

para a mesma tela de aquisição de aula que no exemplo da aula pública, diferenciando-se apenas na forma de acessá-la.

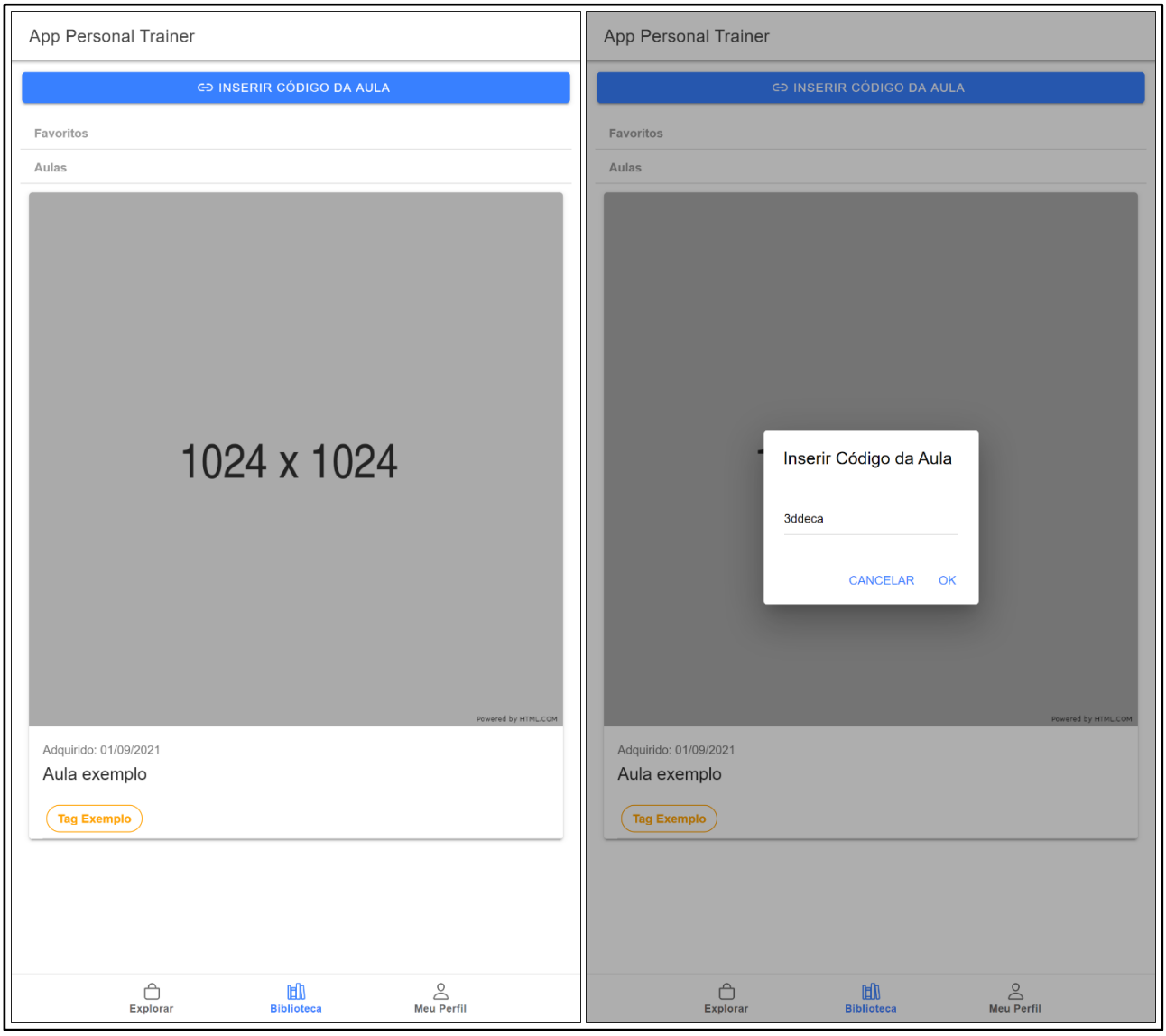

Figura 30: Listagem das aulas na biblioteca e botão para acesso a aulas não publicadas.

A Figura 31 mostra o fluxo que faz possível o *feedback* do aluno para o instrutor, o compartilhamento de sessões. Neste sistema, uma sessão representa uma aula praticada pelo usuário, que é iniciada a partir da visão da aula na biblioteca pelo botão “Iniciar nova sessão com esta aula”, salvando a data e hora de início.

Com uma sessão em andamento, o usuário pode navegar livremente por todas as atividades pertencentes a aula, podendo avançar ou retroceder quando necessário, e com acesso aos conteúdos detalhados de cada atividade.

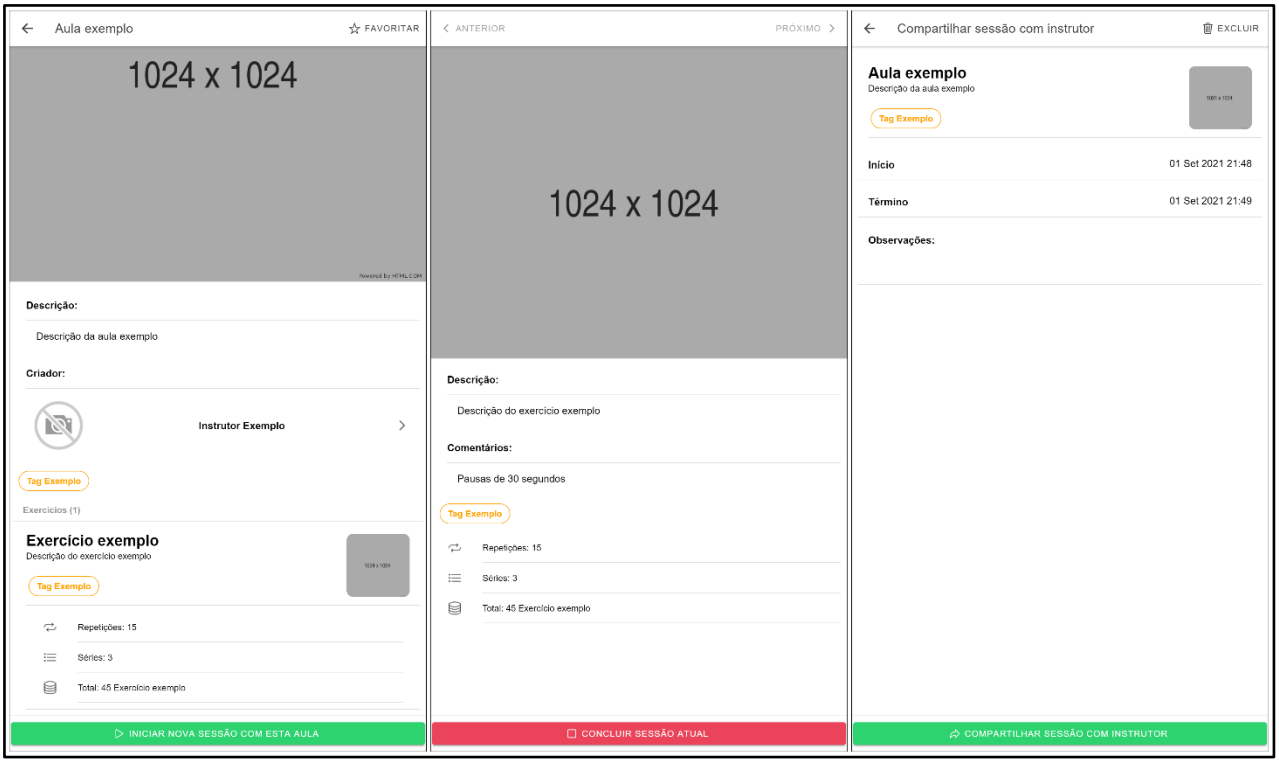

Figura 31: Fluxo de cadastro de uma sessão

Ao finalizar uma sessão através do botão “Concluir sessão atual”, é registrada a data e hora de término e o usuário é redirecionado para a tela de conclusão, onde é possível ajustar os registros de data e hora assim como adicionar observações pessoais sobre a aula ou sessão. Opcionalmente pode ser feito o envio das informações da sessão para o instrutor ou excluir as mesmas.

Na aba “Perfil” de ambos os tipos de usuários, há um item para a listagem de sessões. Onde um usuário instrutor pode ver sessões geradas a partir de aulas criadas por ele, enquanto o usuário aluno tem a listagem das próprias sessões.

5.4. Outros recursos

Junto a geração do projeto, são configurados por padrão os temas escuro e claro, e são aplicados conforme o tema utilizado pelo sistema do aparelho do usuário, se disponível. A Figura 32 é um comparativos visual dos dois temas, assim como uma demonstração da mesma interface nas duas opções de idioma disponíveis.

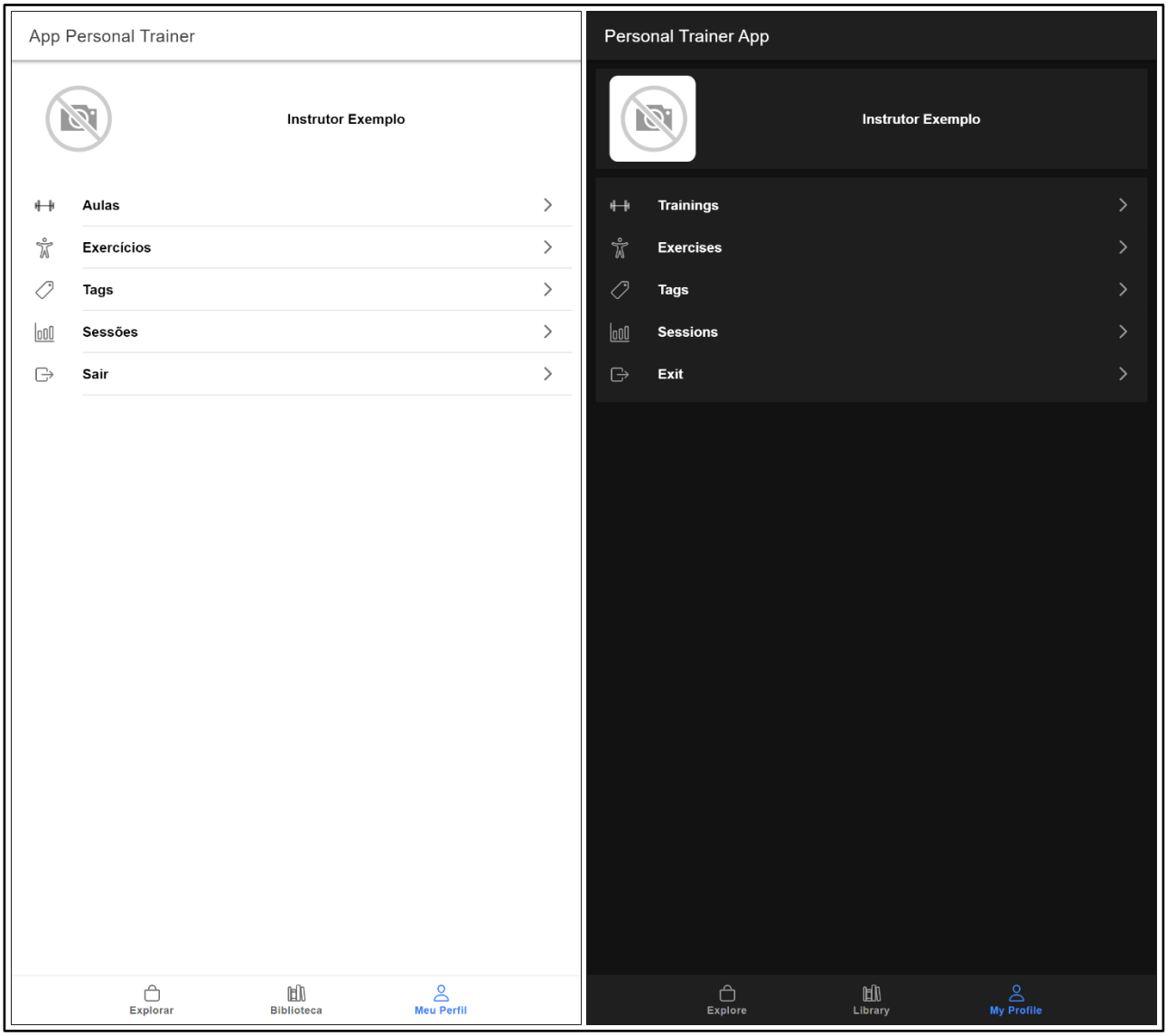

Figura 32: Comparativo dos temas e idiomas.

6. CONCLUSÕES

No primeiro momento foram explanados os conceitos utilizados no decorrer do projeto, focando-se no desenvolvimento do aplicativo móvel e da API. No capítulo de desenvolvimento móvel foi revisado o contexto das duas plataformas de sistema operacional em destaque no mercado profissional e as diferenças da programação nativa para a programação híbrida, assim como as ferramentas mais populares no momento para esta pesquisa científica. No tópico de WebServices foram apresentados alguns conceitos que serão bem utilizados durante o projeto para que no futuro essa API esteja com uma estrutura de fácil expansão para acomodar possíveis clientes alternativos, como uma aplicação Web.

Durante o desenvolvimento do aplicativo, foi notado como ponto positivo, execução da aplicação em um navegador, evitando todo o processo de geração dos pacotes de instalação e dependência de um aparelho móvel compatível com o *software*, possível devido a *framework* Ionic.

Como ponto negativo, não foi possível o teste da aplicação em um ambiente iOS por depender de programas exclusivos para sistemas macOS. As funcionalidades e *layouts* foram acomodadas para utilização no navegador e em aparelhos Android, e pode conter inconsistências com a compilação final para iOS.

O *software*, como resultado da fase de desenvolvimento, atende os objetivos do trabalho e pode ser considerado um MVC da proposta. Uma funcionalidade não implementada na versão final do projeto é o suporte a vídeos, que limita a inclusão de mídias ao exercício a imagens apenas. Apesar da estrutura de dados de tal funcionalidade já ser parte do projeto, não foram implementadas as interface para que usuários possam inserir ou interagir com tais tipos de mídia.

Para trabalhos futuros, pretende-se implantar o aplicativo de forma comercial, intermediando a aquisição de aulas de forma beneficiar o usuário que as pública, assim como melhorias de infraestrutura para um funcionamento mais eficiente, tanto da API quanto da aplicação móvel, em meio a um grande fluxo de dados com múltiplos usuários simultâneos.

Também estão planejadas melhorias técnicas, como o uso do aplicativo *offline* com um sistema de sincronismo e ferramentas de inteligência de dados para instrutores acompanharem estatísticas sobre suas aulas e alunos.

O aplicativo proposto possui potencial de contribuição no cenário atual da pandemia de 2021, como uma nova alternativa para atividades remotas em momentos de isolamento.

REFERÊNCIAS

ABDULRASOOL, Saleem. Introducing Swift on Windows. **Swift**. 22 de set. de 2020. Disponível em: <https://swift.org/blog/swift-on-windows/>. Acesso em: 15 de mar. de 2021.

ANGULAR, **Angular**. 2021. Disponível em <https://angular.io>. Acesso em: 29 de jul. de 2021.

BIØRN-Hansen, Andreas; MAJCHRZAK, Tim A. & GRØNLI, Tor-Morten. (2017, April). **Progressive Web Apps: The Possible Web-native Unifier for Mobile Development**. In WEBIST (pp. 344-351).

BLINI, Felipe. Desenvolvimento de Aplicativos Móveis com Javascript: Ionic, React Native e NativeScript. Qual escolher?. Disponível em: <https://medium.com/@felipeblini/desenvolvimento-de-aplicativos-m%C3%B3veis-com-javascript-ionic-react-native-e-nativescript-c303b17fba0d>. Acesso em: 29 de jul. de 2021.

EISENMAN, Bonnie. **Learning React Native. Building Native Mobile Apps with JavaScript**. Printed in the United States of America. Published by O'Reilly Media, Inc, 2015.

GOMIDE, Raphael Ribeiro. 10 motivos para desenvolver apps com Ionic. **IGTI Blog**. 11 de mai. de 2018. Disponível em: <https://www.igti.com.br/blog/10-motivos-desenvolver-apps-com-ionic/>. Acesso em 20 de nov. de 2020.

Google. Platform Architecture. **Android Developers.** 11 de mar. de 2021. Disponível em: < https://developer.android.com/guide/platform>. Acesso em 28 de jul. de 2021

GUSMÃO, Gustavo. Brasileiros começam a apostar em aplicativos para substituir academias. **Exame**. 8 de jan. de 2019. Disponível em: <https://exame.com/tecnologia/brasileiros-comecam-a-apostar-em-aplicativos-para-substituir-academias/>. Acesso em 04 de nov. de 2020.

IONIC, **Ionic Framework**. 2020. Disponível em <https://ionicframework.com/docs>. Acesso em: 29 de jul. de 2021.

KOAY, Alvin, Android vs iOS Market Share 2020: Stats and Facts. **MobileApps.com**. 16 de fev. de 2011. Disponível em: <https://www.mobileapps.com/blog/android-vs-ios-market-share>. Acesso em 17 de mar. de 2021.

KULKARNI, Chaitanya Mukund; TAKALIKAR, M. **„Analysis of REST API Implementation “**, 2018.

LUCIANO, Josué; ALVES, Wallison Joel Barberá. Padrão de arquitetura MVC: Model-view-controller. **EPeQ Fafibe**, v. 1, n. 3a, p. 102-107, 2011.

MOSCHETTI, Vitor A K, Explicando o Universo do Spring. **Medium**. 22 de ago. de 2020 Disponível em: <https://medium.com/@vitormoschetti/o-que-%C3%A9-o-spring-c672be3477b8>. Acesso em 13 de mar. de 2021.

NEVES, Jonathan; JUNIOR, Vilmar Mendes. Uma análise comparativa entre flutter e react native como frameworks para desenvolvimento híbrido de aplicativos mobile: Estudo de caso visando produtividade. **Ciência da Computação-Tubarão**, 2020.

NICOLL, Stéphane; SYER, Dave; BHAVE Madhura. Spring Initializr Reference Guide. **Spring**. 19 de fev. de 2021. Disponível em: <https://docs.spring.io/initializr/docs/current/reference/html/> . Acesso em 18 de mar. de 2021.

PERUCCI, Gustavo. Especialistas alertam para riscos de aplicativos fitness sem orientação. **Saúde Plena**, 04 de set. de 2016. Disponível em: <https://www.uai.com.br/app/noticia/saude/2016/09/04/noticias-saude,189718/especialistas-alertam-para-riscos-de-aplicativos-fitness-sem-orientaca.shtml>. Acesso em 26 de out. de 2020.

PRABHU, John. MVC Architecture & Its Benefits in Web Application Development. **TechAffinity**. 18 de jul. de 2019. Disponível em: <https://techaffinity.com/blog/mvc-architecture-benefits-of-mvc/>. Acesso em 18 de mar. de 2021.

SILVA, E. P. A. da; SOTTO, E. C. S. A UTILIZAÇÃO DO IONIC FRAMEWORK NO DESENVOLVIMENTO DE APLICAÇÕES HÍBRIDAS EM ARQUITETURA ORIENTADA A SERVIÇO. **Revista Interface Tecnológica**, [S. l.], v. 15, n. 1, p. 97-108, 2018. DOI: 10.31510/infa.v15i1.333. Disponível em: <https://revista.fatectq.edu.br/index.php/interfacetecnologica/article/view/333>. Acesso em: 29 jul. 2021.

SILVA, Marcelo Moro da; SANTOS, Marilde Terezinha Prado. Os Paradigmas de Desenvolvimento de Aplicativos para Aparelhos Celulares. **T.I.S. São Carlos**. ago. de 2014.

StatCounter, Mobile Operating System Market Share Worldwide, **StatCounter**. fev. de 2021. Disponível em: <https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-202002-202102> acesso em 13 de mar. de 2021.
The PostgreSQL Global Development Group. PostgreSQL: About. **PostgreSQL**. 11 de fev. de 2021. Disponível em: <https://www.postgresql.org/about/>. Acesso em 18 de mar. de 2021.

WALLS, Craig. **Spring Boot in action**. Manning Publications, 2016.