LIGA DE ENSINO DO RIO GRANDE DO NORTE

CENTRO UNIVERSITÁRIO DO RIO GRANDE DO NORTE

ESPECIALIZAÇÃO EM SISTEMAS CORPORATIVOS

VACINAPP: UM APLICATIVO PARA O CONTROLE DE VACINAS

NATAL/RN

2017

LUCAS PROCÓPIO DE ARAÚJO CARVALHO

VACINAPP: UM APLICATIVO PARA CONTROLE DE VACINAS

Trabalho de conclusão do curso de Especialização em Sistemas Corporativos apresentado ao Centro Universitário do Rio Grande do Norte (UNI-RN) como requisito final para obtenção do título pós-graduando em sistemas corporativos.

Orientador: Prof. Lucas Farias de Oliveira.

NATAL/RN

2017

AGRADECIMENTO

Agradeço primeiramente a Deus que sempre agracia a minha vida com sabedoria, paz, amor e felicidade. Agradeço também do fundo do meu coração a minha esposa Renata Carvalho pelo apoio e compreensão na jornada de desenvolvimento desse trabalho de conclusão de curso, bem como agradeço aos meus pais Airton e Célia, irmãs Daniella e Mirella, cunhados Higor, Daniel e Júlio, amigos e família de maneira geral por todo apoio do dia a dia.

Gostaria de registrar também um agradecimento especial ao meu orientador Lucas Oliveira, por ter me aceitado como orientando e por de fato me mostrar, com suas ações, o que é orientar um aluno em um trabalho de conclusão de curso, agradecendo também a todas as dicas e conhecimentos passado por ele para mim.

Quero deixar registrado também meu agradecimento aos amigos de trabalho, aos meus companheiros de pós, e de toda hora, e ao corpo docente da UNIRN.

Para encerrar deixo meu registro de agradecimento a série de jogos The Legend of Zelda e sua brilhante trilha sonora, que foi um motivador a mais para remediar os momentos em que o cansaço surgia.

RESUMO

Quando desenvolvemos um software visamos sempre o quão útil aquele software será para as pessoas, seja em termos de utilidade prática de maneira geral ou mesmo para o entretenimento. O software desenvolvido nesse trabalho tem como ênfase a utilidade pública, mais precisamente a saúde pública, sendo o objetivo dele, catalogar as vacinas de um indivíduo ou mesmo de seus dependentes, gerenciando o controle de vacinação de maneira geral.

Além do foco na oferta de valor com o software desenvolvido, diversas tecnologias diferentes foram utilizadas com o intuito de tornar o desenvolvimento mais otimizado e eficaz, sempre tentando manter as boas práticas de programação e a padronização adotada pela comunidade de desenvolvedores de software.

Palavras-chave: Software, Vacina, Desenvolvimento.

ABSTRACT

When we develop a software, we often thought about how useful to people that software would be, either in general matters or in terms of entertainment. The Software developed in this work was build for public utility, precisely public heath, with its primary goal the storage of a vaccines catalog from a person or his/her children.

Besides the focus on the value offer from the software developed, this work used many different technologies aiming the optimization and efficacy of the development, always trying to keep the good practices of programming and the standardization of software development community.

Keywords: Software, Vaccination, Development.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLA

API	*Application Programming Interface*
IDE	*Integrated Development Environment*
REST	*Representation State Transfer*
HSQLDB	*Hyper Structured Query Language Database*
UML	*Unified Modeling Language*
JAR	*Java Archive*
JPA	*Java Persistence API*
MVC	*Model View Controller*
HTTP	*Hypertext Transfer Protocol*
JSON	*Java Script Object Notation*
CRUD	*Create Read Update Delete*
XML	*Extensible Markup Language*
MVP	*Minimum Viable Product*

SUMÁRIO

1 INTRODUÇÃO

Muitas pessoas não se previnem contra as doenças ou mesmo quando tentam, elas não fazem o controle das vacinas corretamente. Muitas vezes o controle de vacinação não é realizado, pois as pessoas carecem de informações sobre quais prevenções existem para determinados tipos de doenças ou mesmo porque esquecem as datas de vacinação ou ainda, no pior caso, porque perdem o cartão de controle de vacinação.

A proposta desse trabalho é a apresentação do software “Vacinapp”, que é uma ferramenta para o autogerenciamento de vacinação da população e também um portal de acesso a informações de utilidade pública no âmbito da saúde, dando ênfase nos aspectos técnicos do seu desenvolvimento. Como mencionado, o software desenvolvido nesse trabalho tem como finalidade o autogerenciamento das vacinas tomadas por um indivíduo, esse aspecto do autogerenciamento aliado ao fato de que na maioria dos casos o indivíduo se desloca de sua residência para realização de vacinações e que os dispositivos móveis (Celulares, Tablets e etc) estão cada vez mais populares, fizeram com que o software fosse desenvolvido para a plataforma dos dispositivos móveis, onde o software é mais comumente chamado de aplicativo.

Além do aplicativo, um servidor também foi desenvolvido como parte do projeto da aplicação como um todo. Esse servidor funciona como uma API REST que fica disponível para que o software do dispositivo móvel se comunique com ele sempre que necessário, como nos casos de acesso aos dados armazenados na base de dados ou mesmo para persisti-los.

Cada elemento do projeto será detalhado em termos do desenvolvimento, especificando a tecnologia utilizada, a justificativa para que ela tenha sido utilizada e em alguns casos serão apresentados comparativos com outras tecnologias para que a justificativa de utilização seja reforçada. Quanto à disposição do trabalho, primeiramente os aspectos em torno do servidor serão detalhados para só então especificar o aplicativo desenvolvido.

2 VISÃO GERAL DO SISTEMA

O software alvo desse relatório técnico é constituído de duas grandes partes, uma dessas partes é um servidor de arquitetura REST que se comunica com a outra parte que é a do aplicativo, intitulado de “Vacinapp”, desenvolvido para o sistema operacional Android.

O software “Vacinapp” tem como função principal o registro das vacinações realizadas por um indivíduo, por essa razão o caso de uso de cadastro de vacinação foi tomado como o estudo de caso desse relatório técnico, onde as tecnologias utilizadas, bem como a estrutura do projeto serão detalhadas tendo esse caso de uso como foco principal.

Basicamente a funcionalidade principal do sistema é formada por três principais passos, que são: a definição da doença prevenida pela vacina tomada, especificação das doses dessa vacina, ou seja, a dose que está sendo tomada e as próximas que virão (caso a vacina seja composta por mais de uma dose), por fim o preenchimento dos demais dados da vacinação. Esses três passos estão especificados abaixo pela documentação dos casos de uso de cada um dos passos citados, passos esses que serão detalhados em termos técnicos mais adiante.

UC001 – Cadastrar Vacinação

Sumário:

Usuário realiza a operação de cadastro de uma nova vacinação.

Atores:

Usuário do Sistema.

Pré-condições:

Os dados referentes aos catálogos das doenças e suas respectivas vacinas estarem previamente cadastrados na base de dados.

Fluxo Básico:

Este caso de uso se inicia quando o ator acessa a funcionalidade “Gerenciar Minhas Vacinas” na tela da gerência de vacina.

A001 – O sistema apresenta uma tela com um campo de inserção de texto para definição da doença, cuja prevenção foi realizada através da vacina tomada, um botão para definição das doses associadas aquele tipo de vacina, bem como um campo texto para inserção do Lote referente a vacina.

A002 – O ator seleciona a opção “Inserir”.

Fluxo Alternativo:

FA001 – Se a doença escolhida estiver associada a uma vacina que precise de renovação.

ALT001.1 – O sistema exibirá uma opção extra no cadastro da vacinação, essa opção é um botão que exibirá um calendário para inserção da data em que a vacina precisará ser renovada.

FA002 – Se o usuário escolher a opção “Cancelar” na tela de cadastro da vacinação.

ALT002.1 – O sistema retornará para a tela de gerência de vacinas.

Figura 1 – Documentação do caso de uso “Cadastrar Vacinação”.

Fonte: Print Screen da documentação do software “Vacinapp”.

UC002 – Escolher Doença

Sumário:

Usuário realiza a operação de escolha de para qual doença a vacina foi destinada.

Atores:

Usuário do Sistema.

Pré-condições:

O usuário ter selecionado o campo texto para definição da doença na tela de registro da vacinação (UC001).

Fluxo Básico:

Este caso de uso se inicia quando o ator seleciona a opção de definição da doença na tela de registro da vacinação.

A001 – O sistema apresenta uma tela com um campo texto, que quando preenchido irá sugestionar as opções de doenças conforma o texto previamente escrito.

A002 – O ator seleciona uma das opções sugeridas pelo sistema.

A003 – após seleção, o sistema automaticamente redirecionará o usuário para a tela de registro da vacinação (UC001).

Figura 2 – Documentação do caso de uso “Escolher Doença”.

Fonte: Print Screen da documentação do software “Vacinapp”.

UC003 – Definir Doses

Sumário:

Usuário realiza a operação de pré cadastro das doses referente a vacina tomada.

Atores:

Usuário do Sistema.

Pré-condições:

O usuário ter selecionado o botão “Definir Doses” na tela de registro da vacinação (UC001).

Fluxo Básico:

Este caso de uso se inicia quando o ator seleciona o botão “Definir Doses” na tela de registro da vacinação (UC001).

A001 – O sistema apresenta uma tela com um campo texto para definirão de um número que representa a ordem da dose tomada e de um calendário para definição da data em que a dose deve ser tomada.

A002 – O ator seleciona a opção “Inserir”.

Fluxo Alternativo:

FA001 – Se o usuário inserir a quantidade de doses igual ao número máximo de doses daquela vacina específica.

ALT001.1 – O sistema inativará o botão “Inserir” permitindo apenas que o usuário selecione a opção “Voltar”.

FA002 – Se o usuário escolher a opção “Voltar” na tela de definição das doses.

ALT002.1 – O sistema retornará para a tela de registro da vacinação.

Figura 3 – Documentação do caso de uso “Definir Doses”.

Fonte: Print Screen da documentação do software “Vacinapp”.

2.1 ESTRUTURA DO PROJETO

O projeto do servidor REST foi desenvolvido na IDE Eclipse, que é uma ferramenta consolidada e bem popular para o desenvolvimento de projetos da linguagem JAVA, e foram construídas trinta e duas classes organizadas em quatro pacotes distintos, conforme imagem abaixo.

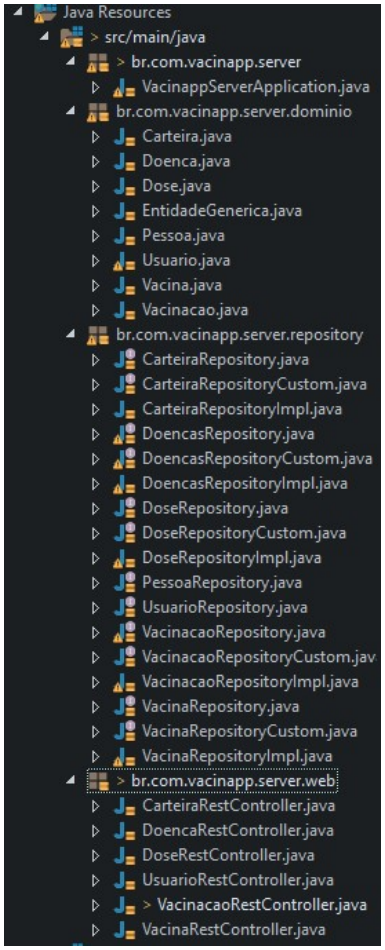


Figura 4 – Estrutura das classes no projeto do servidor REST.

Fonte: Print Screen projeto na IDE Eclipse.

Já em relação ao aplicativo para Android, esse foi desenvolvido na IDE Android Studio, ferramenta que evoluiu muito em termos de recursos e compatibilidade com outras tecnologias, se tornando uma das ferramentas mais populares no desenvolvimento de aplicativos do sistema operacional supracitado. A estrutura organizacional do projeto conta com um total de dezessete classes dispostas em três pacotes distintos, além das classes o projeto também dispõe de cinco “Activity” (ou “Activities”) que é a forma padrão de nomenclatura para as telas do sistema. A estrutura do projeto encontra-se nas Figuras 5 e 6 abaixo.

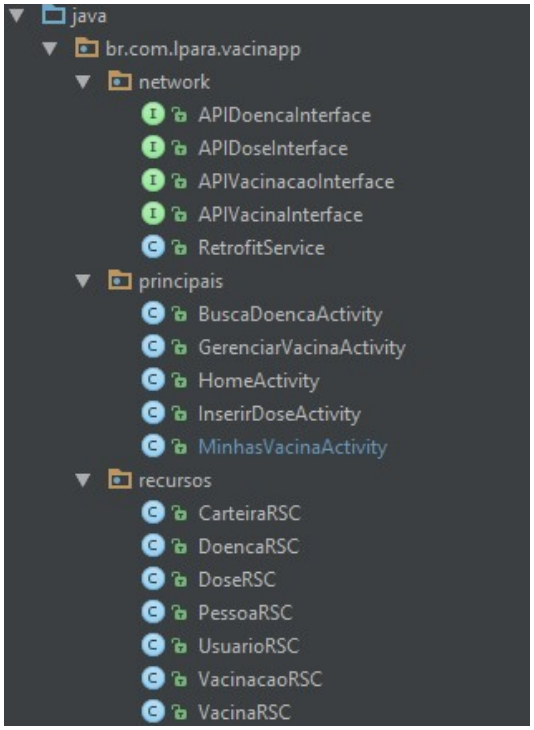


Figura 5 – Estrutura das classes no projeto do aplicativo “Vacinapp”.

Fonte: Print Screen projeto na IDE Android Studio.

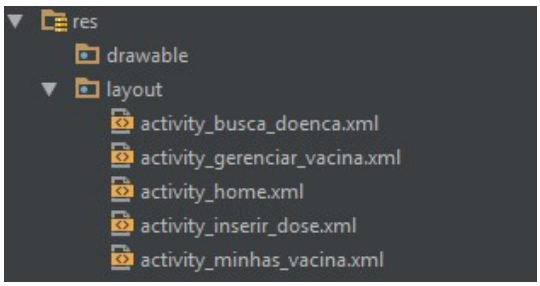


Figura 6 – Estrutura das “activities” no projeto do aplicativo “Vacinapp”.

Fonte: Print Screen projeto na IDE Android Studio.

O aspecto negocial do sistema tem como pilar de sustentação as classes de domínio juntamente com seus relacionamentos. A principal classe de domínio é a classe “Vacinacao”, pois é ela que representa o *core* do sistema. Em termos de conceito, temos que um usuário do sistema possui uma carteira de vacina que armazena vacinações de diversas vacinas diferentes, responsáveis por prevenir uma ou mais doenças específicas. Todas essas características são reflexo de como está estruturado o mapeamento das classes e como o sistema realiza as operações em cima desse mapeamento (Figura 7).

As operações realizadas pelo sistema serão detalhadas nas seções mais adiante desse relatório técnico.

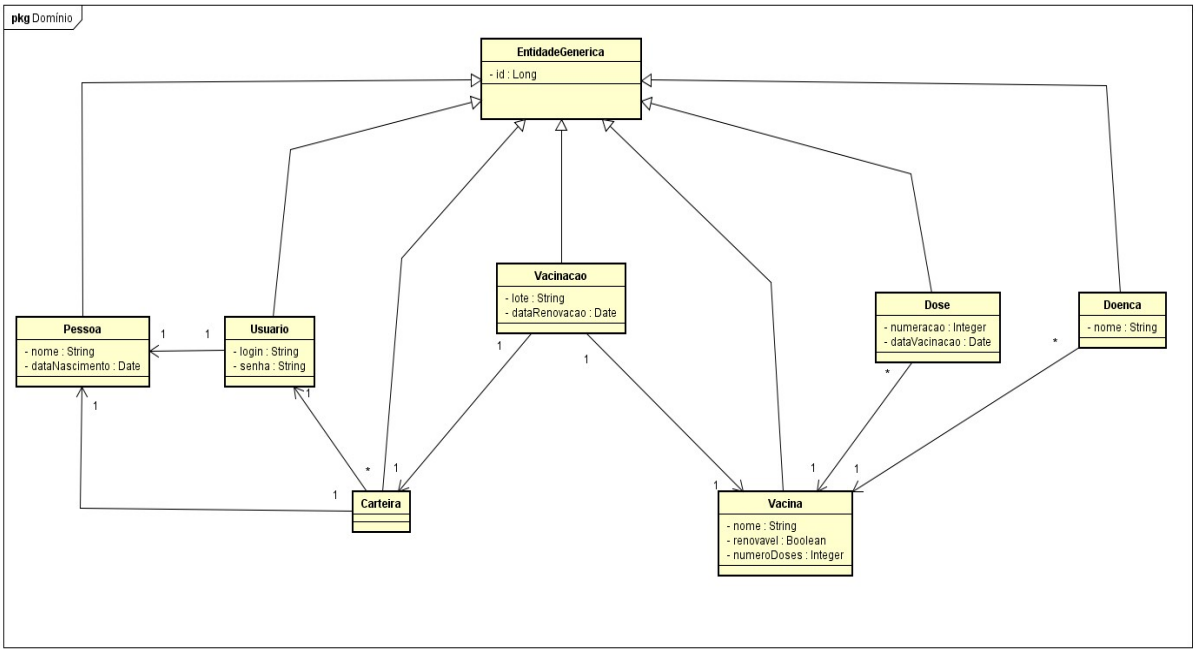


Figura 7 – Diagrama UML de classes de domínio do sistema.

Fonte: Print Screen modelagem de classes da ferramenta Astah.

3 O LADO SERVIDOR

O projeto do servidor do sistema é um projeto Spring Boot e foi criado a partir do portal online do Spring Boot (http://start.spring.io/), onde é possível gerar um projeto de acordo com as exigências necessárias para se conseguir criar o sistema desejado. Para o servidor do projeto “Vacinapp” a estratégia adotada foi a de seguir a proposta passada pelo Spring Boot, que é a de se preocupar menos com configurações e mais com o aspecto negocial do sistema.

Dentre as opções possíveis no portal do Spring Boot, o servidor do “Vacinapp” foi gerado a partir das seguintes escolhas: Maven, Java, Web e JPA, onde todas essas escolhas serão detalhadas na seção a seguir.

3.1 A API REST EM DETALHES

O projeto Spring Boot gerado a partir do portal “SPRING INITIALIZR” traz diversas facilidades que contribuem muito para que o desenvolvedor possa de fato investir menos tempo na etapa de configuração do projeto, com o servidor do projeto “Vacinapp” não foi diferente.

Todas as configurações escolhidas foram definidas com o intuito de construir uma API REST que possa fornecer informações para o aplicativo que compõe o projeto “Vacinapp” como um todo.

Detalhando as configurações escolhidas para a API REST temos a primeira, que foi a de um projeto do tipo Maven. Um projeto do tipo Maven tem como característica a gestão das dependências, que fica toda a cargo dessa ferramenta, criando a possibilidade de que a dependência necessária seja reconhecida e baixada com um simples ajuste em um arquivo de configuração (Figura 8). Além dessa importante característica, o projeto Maven também se torna muito pequeno pelo fato de não armazenar os arquivos “.JAR”, das dependências, na estrutura do projeto, essa característica ajuda na transferência do projeto seja para um eventual backup ou para mudança de estação de trabalho. A linguagem de programação utilizada foi o JAVA, que também é a linguagem utilizada no aplicativo Android.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>br.com.vacinapp</groupId>
    <artifactId>vacinapp-server</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>war</packaging>

    <name>VacinappServer</name>
    <description>Server side of Vacinapp</description>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.8.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>


</project>
```

Figura 8 – Estrutura do arquivo de configuração do projeto servidor, “pom.xml”.

Fonte: Print Screen do arquivo “pom.xml” na IDE Eclipse.

Como é possível observar na figura 8, o arquivo “pom.xml” contém várias “tags” em sua estrutura, e essas “tags” tem funções distintas que podem ser a de definição das dependências do projeto, como também outras características tais como a versão específica da linguem JAVA que será utilizada, através da tag “<java.version>”, ou mesmo para definir a identificação do projeto formada pelas tags “groupId”, “artifactId” e “version” e etc. Todas as dependências ficam entre as tags “<dependencies>” e “</dependencies>”, sempre que inserido um novo valor, o Maven já se encarregará de baixar a biblioteca/framework necessário.

A outra característica selecionada para o projeto do servidor foi “Web”, essa opção já configura o projeto para utilização do *Spring MVC*, como também inclui um *container* embarcado, uma espécie de servidor de aplicação *Tomcat* “embutido” no projeto, eliminando a necessidade da inclusão e configuração manual do servidor de aplicação.

O Spring Framework contém diversas ferramentas para auxiliar o desenvolvimento de aplicações, na API REST desenvolvida para o fornecimento de dados ao aplicativo “Vacinapp”, as ferramentas do Spring Framework utilizadas foram o *Spring MVC* e o Spring JPA.

O fluxo de comunicação do projeto está representado na figura 9 abaixo, a partir dele é possível compreender que o aplicativo fará as requisições HTTP para a API REST e essa API por sua vez irá acessar a base de dados para a busca, inserção, alteração ou remoção de um dado, retornando a resposta para o cliente que o solicitou, que no caso é o aplicativo.

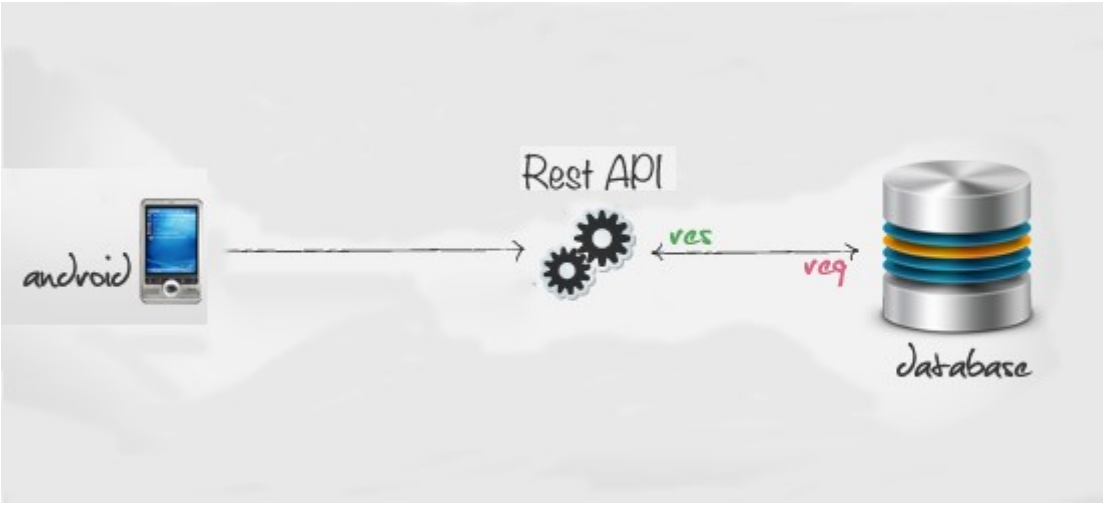


Figura 9 – Estrutura de comunicação dos módulos do projeto.

Fonte: Adaptado de http://dselva.co.in/blog/wp-content/uploads/2017/03/REST-API-in-PHP.png.

As requisições HTTP do aplicativo, no dispositivo móvel, é interpretada pelo *Controller* do Spring MVC, a partir dessa interpretação de qual tipo de requisição está sendo solicitada é que o *Controller* inicia o respectivo método de acesso a base de dados.

```java
@RestController
@RequestMapping("/vacinacoes")
public class VacinacaoRestController {

    @Autowired
    private VacinacaoRepository vacinacaoRepo;
    @Autowired
    private DoseRepository doseRepo;

    @PostMapping("/new")
    public Vacinacao inserirVacinacao(@RequestBody Vacinacao vacinacao){
        vacinacao = vacinacaoRepo.save(vacinacao);
        inserirDosesDaVacinacao(vacinacao);

        return vacinacao;
    }

    @GetMapping
    public List<Vacinacao> listarVacinacoes(){
        return vacinacaoRepo.findAll();
    }

    @GetMapping("/{id}")
    public Vacinacao buscarVacinacaoPorId(@PathVariable("id") Long idVacinacao){
        return vacinacaoRepo.findOne(idVacinacao);
    }

    @GetMapping("/carteira/{id}")
    public List<Vacinacao> buscarVacinacoesPelaCarteira(@PathVariable("id") Long idCarteira){
        return vacinacaoRepo.buscarVacinacoesPorCarteira(idCarteira);
    }

    private void inserirDosesDaVacinacao(Vacinacao vacinacao){
        if(vacinacao.getDoses() != null && vacinacao.getDoses().size() > 0){
            for(Dose dose : vacinacao.getDoses()){
                dose.setVacinacao(vacinacao);
            }

            doseRepo.save(vacinacao.getDoses());
        }
    }
}
```

Figura 10 – Declaração de um Controller da API.

Fonte: Print Srcreen da classe “VacinacaoRestController” no Eclipse.

A classe exibida na figura 10 é o *controller*, ou controlador, responsável por tratar as requisições HTTP referentes a entidade Vacinação que é a entidade principal do projeto. Na declaração do controlador foi utilizado a *annotation* “RestController”, pois ela já desempenha o mesmo papel das *annotation* “Controller” e “ResponseBody” combinadas, além disso o mapeamento das chamadas a esse controlador foi definido com contexto “/vacinacoes”, através da *annotation* “RequestMapping” para diferenciar dos demais controladores das outras entidades.

Ainda dentro do escopo do controlador exibido na figura 10, temos os métodos que realizam as ações conforme a requisição HTTP enviada, como exemplo temos o caso do método “inserirVacinacao” que realiza ações referentes a requisições do tipo POST, realizadas a partir do contexto “/vacinacoes/new”, sendo tudo isso definido apenas com a *annotation* “PostMapping”. O mesmo vale para o método “buscarVacinacaoPorId” que tem sua representação nas requisições HTTP mapeadas pela *annotation* “GetMapping” sendo essa uma configuração de requisição HTTP do tipo GET, essa mesma lógica vale para os demais métodos.

O método “inserirVacinacao” tem como parâmetro esperado um objeto do tipo “Vacinacao” que virá em forma de JSON de todo o objeto, isso só é possível pelo uso da *annotation* “RequestBody”, evitando o excesso de código gerado nos casos em que o método responsável pelo POST recebe cada atributo da entidade como um parâmetro diferente atrelado a *annotation* “Field”.

Falando especificamente do estilo arquitetural REST, segundo Leonard Richardson (QCon Talk, 2008), existem quatro níveis de maturidade da arquitetura REST (Figura 11) que ajudam a explicar as características específicas de um sistema web.

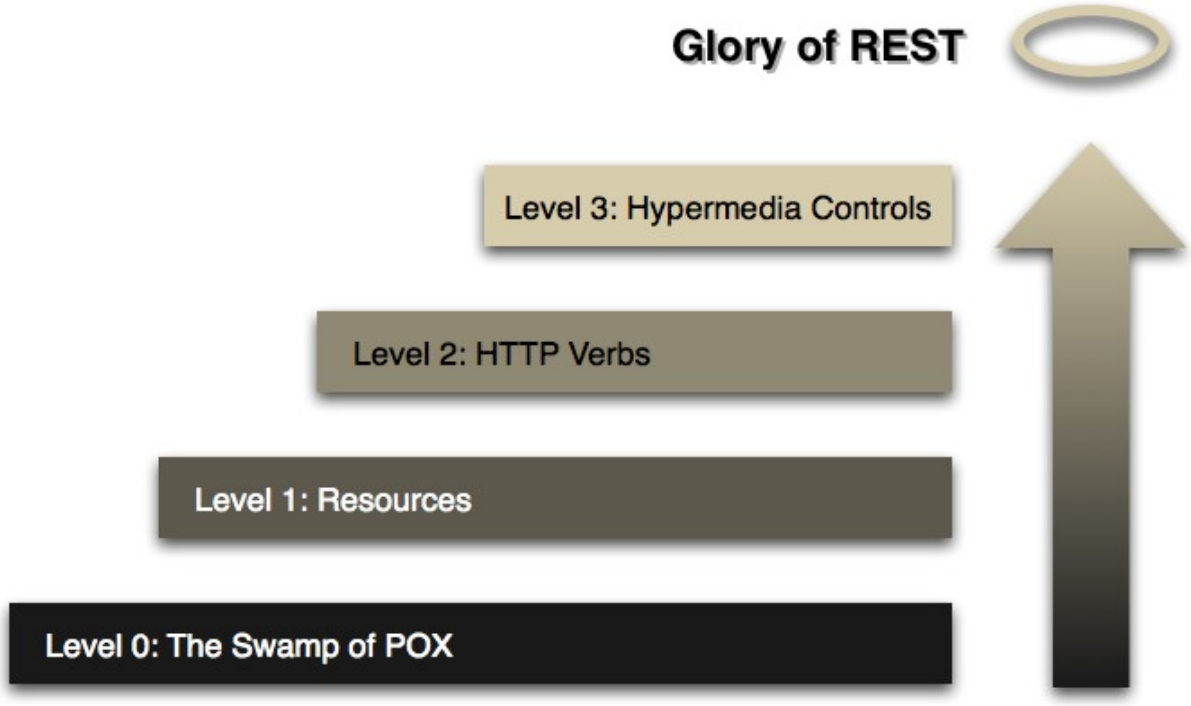


Figura 11 – Os níveis de maturidade do REST.

Fonte: https://martinfowler.com/articles/images/richardsonMaturityModel/overview.png.

O nível zero de maturidade é caracterizado por utilizar um protocolo de transporte, como o HTTP, sem utilizá-lo para indicar o estado da aplicação. Além disso apenas um endereço de acesso, ou *endpoint*, é utilizado atrelado a apenas um tipo de método do protocolo de transporte, o POST do HTTP, por exemplo. O nível um de maturidade já se caracteriza por utilizar vários *endpoints*, onde cada um deles fornecem acesso a recursos diferentes, porém esses acessos ainda estão amarrados a apenas um tipo de método do protocolo de transporte. Quanto ao nível dois da maturidade do REST, temos que é caracterizado por não estar amarrado a apenas um método do protocolo de transporte, ou seja, não só usa o POST por exemplo, faz uso do GET quando for preciso recuperar recursos ou mesmo do DELETE quando for necessário removê-los, além disso esse nível de maturidade faz uso dos códigos de status, ou *response codes*, para controle dos status da comunicação. O último nível de maturidade REST, o nível 3, diz respeito ao HATEOAS, que é uma derivação do REST onde é possível gerar o conteúdo dos recursos REST de tal modo que esse recurso inclua *endpoints* nele indicando os demais acessos possíveis para manipular esse recurso, por exemplo. Dessa maneira o nível três consegue indicar qual o próximo passo a ser tomado em termos de interação com o REST.

A API REST desenvolvida nesse projeto atingiu, não completamente, o nível 2 de maturidade do REST justificado pelos seguintes aspectos: utiliza o protocolo de transporte HTTP, como também contém vários *endpoints* de acesso aos recursos, além de utilizar vários métodos do protocolo de transporte com pode ser visto na Figura 10. O único aspecto do segundo nível que não foi utilizado foi o dos códigos de status para controle do status de comunicação com a API, mas essa utilização foi feita de maneira indireta, por isso pode-se considerar que a aplicação atingiu o segundo nível de maturidade.

A última característica para ser abordada da API REST é o fato dela ter sido gerado com JPA, esse framework foi escolhido para o mapeamento das entidades do projeto em detrimento as tabelas do banco, ou seja, o mapeamento objeto relacional, bem como a realização dos CRUDs na base de dados.

```java
@Entity
public class Vacinacao extends EntidadeGenerica{

    @OneToOne
    @NotNull
    private Vacina vacina;

    @OneToOne
    @NotNull
    private Carteira carteira;

    private String lote;

    private Date dataRenovacao;

    @Transient
    private List<Dose> doses;
```

Figura 12 – Mapeamento objeto relacional da classe “vacinacao”.

Fonte: Print Screen da classe “Vacinacao” no Eclipse.

O Spring JPA (ou Sring Data JPA) é um projeto que está inserido em um projeto maior do Spring chamado de Spring Data. Quanto ao Spring JPA, ele traz diversas facilidades para o desenvolvedor, onde uma dessas facilidades é o fato de já existir por padrão uma implementação de um repositório de persistências de dados associado a uma interface chamada “JpaRepository”, a partir desse recurso o projeto só precisa criar novas interfaces das entidades do projeto que estendam a interface “JpaRepository” do Spring JPA, pois a implementação dos métodos fica a cargo do Spring. Esse foi o recurso utilizado para todas as entidades da API.

```java
@Repository
public interface VacinacaoRepository extends JpaRepository<Vacinacao, Long>, VacinacaoRepositoryCustom{


}
```

Figura 13 – Interface de persistência da entidade “Vacinacao”.

Fonte: Print Screen da classe “VacinacaoRepository”.

Apesar do Spring JPA fornecer o excelente recurso da utilização da interface “JpaRepository”, houveram casos em que surgiu a necessidade necessita de utilização de uma consulta customizada, com aspectos específicos do projeto, ou com determinados parâmetros não suportada pela implementação fornecida pelo Spring, nesses casos uma nova interface e implementação foram desenvolvidas para atender à necessidade imposta pelo projeto. As figuras 14 e 15 exibem a interface e implementação, respectivamente, criadas para o desenvolvimento de consultas customizadas. Na figura 13 é possível visualizar o repositório da entidade “Vacinacao” estendendo tanto a implementação genérica do Spring “JpaRepository” quanto a customizada “VacinacaoRepositoryCustom”.

```java
public interface VacinacaoRepositoryCustom {

    public List<Vacinacao> buscarVacinacoesPorCarteira(Long idCarteira);

}
```

Figura 14 – Interface de persistência customizada.

Fonte: Print Screen da classe “VanicaoRepositoryCustom” no Eclipse.

```java
@Repository
public class VacinacaoRepositoryImpl implements VacinacaoRepositoryCustom{

    @PersistenceContext
    EntityManager entityManager;

    public List<Vacinacao> buscarVacinacoesPorCarteira(Long idCarteira){

        String sql = "SELECT * FROM public.vacinacao WHERE carteira_id = ?";

        Query query = entityManager.createNativeQuery(sql);
        query.setParameter(1, idCarteira);

        return query.getResultList();
    }
}
```

Figura 15 – Implementação da interface “VacinacaoRepositoryCustom”.

Fonte: Print Screen da classe “VacniacaoRepositoryImpl” no Eclipse.

Essas interfaces de persistências criadas para as entidades do projeto são incorporadas aos controladores, das respectivas entidades, através da injeção de dependência realizada na instanciação da variável usada para realização das operações CRUD nos métodos dos controladores. Na figura 10 é possível perceber que os repositórios das entidades “Vacinacao” e “Dose” estão sendo incorporados ao controlador por meio da *annotation* “Autowired” que faz a injeção de dependência nas variáveis vacinacaoRepo e doseRepo.

Além de todos os recursos Spring utilizados pela API já citados, ela também utiliza o servidor de aplicação Tomcat que o Spring traz embutido no artefato “.war” (ou “.jar”) da aplicação web, possibilitando que com apena uma *annotation, “SpringBootApplication”,* o servidor web possa ser iniciado através da execução do método main da classe principal do projeto, eliminando também a necessidade de download e configuração do servidor de aplicação de maneira manual. A figura 16 mostra a estrutura base da classe que contém o método “main” da aplicação.

```java
@SpringBootApplication
public class VacinappServerApplication {

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

Figura 16 – Estrutura da classe que contém o método “main” da aplicação.

Fonte: Print Screen da classe “VacinappServerApplication” na IDE Eclipse.

Todos os aspectos da API REST foram detalhados nessa seção, logo as próximas serão dedicadas ao detalhamento da construção do aplicativo, cujo nome é o mesmo do projeto, desenvolvido para o sistema Android na plataforma mobile.

4 O LADO CLIENTE

O software “cliente” desse projeto foi desenvolvido na linguagem JAVA, assim como na API, e foi criado a partir do projeto base que é gerado pela IDE Android Studio. Por padrão a IDE cria os novos projetos como tipo Gradle (recurso semelhante ao Maven) e com uma tela de navegação, chamada de Activity, com padrão definido no momento da criação, que no caso do “Vacinapp” foi uma Activity vazia.

4.1 O APLICATIVO EM DETALHES

O desenvolvimento do aplicativo “Vacinapp” foi desde o início voltado para a comunicação com a API REST criada nesse trabalho, por essa razão toda a estrutura do aplicativo foi desenvolvida para esse fim. O primeiro aspecto levado em conta na construção do aplicativo foi o mapeamento das entidades, pois nesse tipo de cenário onde uma aplicação deseja se comunicar com uma API, é necessário que as entidades criadas na aplicação sejam compatíveis com o JSON retornado pela API, que consequentemente é reflexo das entidades dela.

```
[
    {
        "id": 1,
        "nome": "Antitetânica",
        "renovavel": true,
        "numeroDoses": 3,
        "doencas": null,
        "newEntity": false
    },
    {
        "id": 2,
        "nome": "Febre Amarela",
        "renovavel": false,
        "numeroDoses": 1,
        "doencas": null,
        "newEntity": false
    },
    {
        "id": 3,
        "nome": "Tríplice Viral",
        "renovavel": false,
        "numeroDoses": 2,
        "doencas": null,
        "newEntity": false
    }
]
```


Figura 17 – Lista com registros da entidade Vacina, da API REST, no formato JSON.

Fonte: Print Screen do JSON, retornado da API REST, na ferramenta Postman.

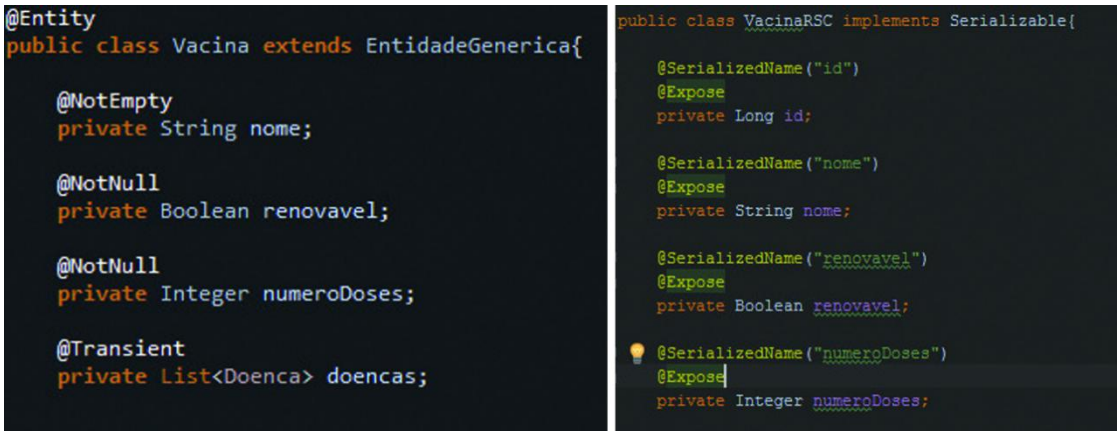


Figura 18 – Comparativo entre a entidade “Vacina” (lado esquerdo) da API com a entidade equivalente do aplicativo (lado direito).

Fonte: Print Screen da classe “Vacinacao” no Eclipse e no Android Studio.

Uma vez que as entidades do aplicativo estão de acordo com as da API, o segundo aspecto a ser tratado é o da comunicação entre ambos os softwares. Nesse quesito foram analisadas duas tecnologias o AsyncTask e o framework Retrofit, onde inicialmente foi realizada uma tentativa de implementação da comunicação com a API através do AsyncTask para só então tentar o Retrofit.

Fazendo um comparativo entre as duas tecnologias, temos que em termos de complexidade, o mesmo objetivo desejado foi alcançado com bem menos linha de código, ou seja, com mais simplicidade e elegância no Retrofit em comparação com o AsyncTask, já no quesito desempenho, nenhum teste de carga foi realizado

no projeto “Vacinapp”, porém no artigo “Android Async HTTP Clients: Volley vs Retrofit” do blog Instruct Tech Blog, referenciado em diversos outros portais de tecnologia, os testes apontam o Retrofit como mais rápido e eficiente.

Para utilizar o Retrofit, além da inclusão de sua dependência no arquivo de configuração do Gradle (build.gradle) é preciso configurar o projeto Android para acesso à internet, incluindo uma linha de configuração (Figura 19) no arquivo de propriedades do projeto (AndroidManifest.xml).

```
<uses-permission android:name="android.permission.INTERNET" />
```

Figura 19 – Comando que configura o projeto Android para acesso a internet.

Fonte: Print Screen da linha de configuração retirada no Eclipse.

Em relação a forma de utilização básica necessária para que o Retrofit possa realizar a comunicação com a API desejada, duas coisas são necessárias, sendo uma dela a criação de uma interface que tem a função de definir os *endpoints* que serão acessados na API (Figura 20), bem como o resultado esperado da funcionalidade que será acessada. E quanto a outra coisa necessária é a instanciação do Retrofit no fluxo do sistema que de fato irá acessar a API.

A estratégia adotada no aplicativo do projeto “Vacinapp”, para atender o segundo pré requisito para utilização do Retrofit, foi a criação de uma classe que tem apenas um método genérico de instanciação do Retrofit. Dessa maneira evitou-se repetição de código nos locais onde a conexão com a API foi realizada (Figura 21).

```
public interface APIDoencaInterface {

    @GET("/doencas")
    Call<List<DoencaRSC>> getDoencasServ();

    @POST("/doencas/new")
    Call<DoencaRSC> insertDoenca(@Body DoencaRSC doenca);

    @GET("/doencas/nome/like/{nome}")
    Call<List<DoencaRSC>> buscarDoencaPorNomeLike (@Path("nome") String nome);

    @GET("/doencas/vacina/{id}")
    Call<VacinaRSC> buscarVacinaPorDoenca(@Path("id") Long idDoenca);

}
```

Figura 20 – Interface de definição dos *endpoints*.

Fonte: Print Screen da interface “APIDoencaInterface” no Eclipse.

```
public class RetrofitService {

    public static <T> T criarRetrofitService(final Class<T> classe, final String urlAPI){
        Gson gsonDate = new GsonBuilder()
                .setDateFormat("yyyy-MM-dd")
                .create();
        final Retrofit retrofit = new Retrofit.Builder()
        .baseUrl(urlAPI)
        .addConverterFactory(GsonConverterFactory.create(gsonDate))
        .build();

        T service = retrofit.create(classe);

        return service;
    }

}
```

Figura 21 – Classe para instanciação genérica do Retrofit.

Fonte: Print Screen da classe RetrofitService no Eclipse.

Uma característica importante e relevante, que impacta diretamente os projetos de aplicativos que fazem acesso a internet, do sistema Android é o de que ele não realiza as operações de acesso à internet diretamente na Thread principal do sistema (UI Thread), pois ela é destinada apenas para lidar com as ações realizadas pelos usuários na interface (Activity) do sistema. Por conta dessa característica do Android, além de instanciar o Retrofit para iniciar o acesso a API, foi preciso também definir que esse acesso fosse assíncrono e realizado numa thread diferente da principal. O Retrofit possibilita atender as duas condições utilizando o as “chamadas” a API através da interface Call e seu método *enqueue* que realiza a as “chamadas” de maneira assíncrona notificando o retorno através do Callback, juntamente com o Handle do próprio Android (Figura 22) que é utilizado justamente para o controle de threads.

```
RetrofitService apiService = new RetrofitService();
APIDoencaInterface apiDoenca = apiService.criarRetrofitService(APIDoencaInterface.class, urlAPI);
Call<List<DoencaRSC>> doencasCall = apiDoenca.getDoencasServ();

doencasCall.enqueue(new Callback<List<DoencaRSC>>() {
    @Override
    public void onResponse(Call<List<DoencaRSC>> call, Response<List<DoencaRSC>> response) {
        if(response.isSuccessful()) {
            List<DoencaRSC> doencas = response.body();
            for(DoencaRSC doenca : doencas) {
                arrayAdDoenca.add(doenca.getNome());
                mapaDoenca.put(doenca.getNome(), doenca);
            }
            mainHandler.post(new Runnable() {
                @Override
                public void run() {
                    actViewDoencas.setAdapter(arrayAdDoenca);
                }
            });

        }else{
            Toast.makeText(getApplicationContext(), "Erro ao acessar API", Toast.LENGTH_SHORT);
        }
    }
```

Figura 22 – Exemplo de acesso a API em um thread diferente da principal, utilizando o Retrofit.

Fonte: Print Screen do fragmento de código no Eclipse.

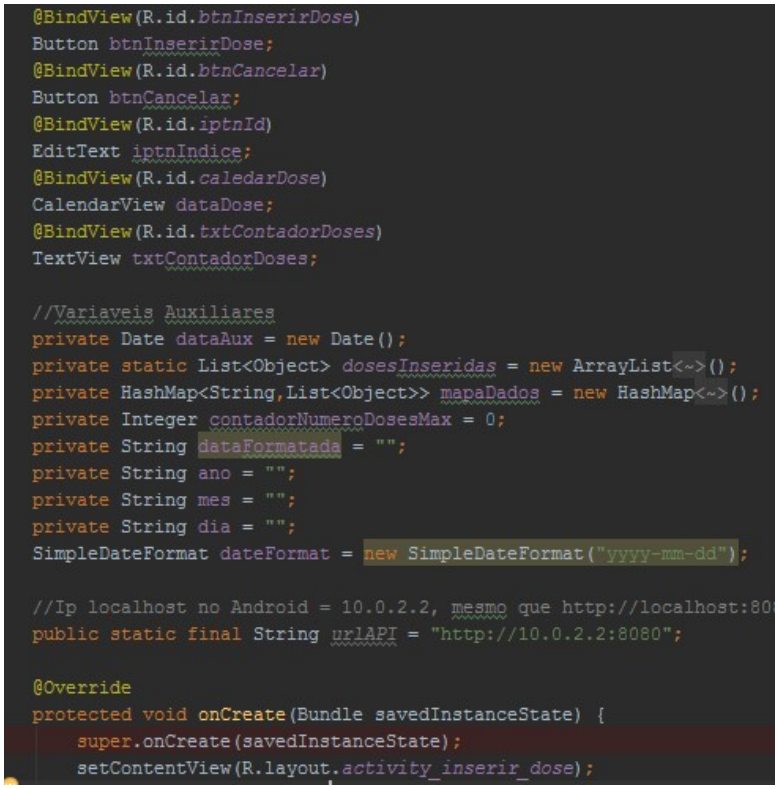
Por padrão as Activities do Android sempre têm um arquivo XML atrelado a uma classe, onde o esse XML é responsável pelo design da Activity e definição dos itens existentes nela (Widgets), enquanto que a classe realiza os devidos processamentos, tais como navegação entre Activities, ações em botões, construção de textos para exibição na Activity e etc. Para que os componentes visuais, ou widgets, definidos no XML possam ser reconhecidos na classe atrelada ao XML, normalmente é preciso fazer repetidamente um código de associação do item presente no XML com uma variável definida na classe, como pode ser visto na figura 23.

```
Button btnInserirDose;
Button btnCancelar;
EditText iptnIndice;
CalendarView dataDose;
TextView txtContadorDoses;

//Variaveis Auxiliares
private Date dataAux = new Date();
private static List<Object> dosesInseridas = new ArrayList<Object>();
private HashMap<String,List<Object>> mapaDados = new HashMap<String, List<Object>>();
private Integer contadorNumeroDosesMax = 0;


//List<DoseRSC> dosesInseridas = new ArrayList<DoseRSC>();

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_inserir_dose);
    btnInserirDose = (Button) findViewById(R.id.btnInserirDose);
    btnCancelar = (Button) findViewById(R.id.btnCancelar);
    iptnIndice = (EditText) findViewById(R.id.iptnId);
    dataDose = (CalendarView) findViewById(R.id.caledarDose);
    txtContadorDoses = (TextView) findViewById(R.id.txtContadorDoses);
```

Figura 23 – Associação dos widgets com as variáveis da classe.

Fonte:https://github.com/lpara/vacinapp/blob/63c9e5635bfcf9be475b31dc8b43c790d5bcee4f/app/src/main/java/br/com/lpara/vacinapp/principais/InserirDoseActivity.java.

Para evitar a repetição de código na associação entre elementos visuais e as variáveis da classe, o aplicativo do projeto “Vacinapp” utilizou o framework Butter Knife que possibilita a associação dos elementos visuais com variáveis da classe através de anotações, além disso os métodos de ações realizadas quando os usuários do aplicativo apertam os botões contidos nas Activities puderam ser mapeados também com anotações, que antes obrigatoriamente eram mapeados no XML. A figura 24 exibe um trecho de código onde é possível evidenciar a utilização do Butter Knife no projeto.



```
@BindView(R.id.btnInserirDose)
Button btnInserirDose;
@BindView(R.id.btnCancelar)
Button btnCancelar;
@BindView(R.id.iptnId)
EditText iptnIndice;
@BindView(R.id.caledarDose)
CalendarView dataDose;
@BindView(R.id.txtContadorDoses)
TextView txtContadorDoses;

//Variaveis Auxiliares
private Date dataAux = new Date();
private static List<Object> dosesInseridas = new ArrayList<~>();
private HashMap<String,List<Object>> mapaDados = new HashMap<~>();
private Integer contadorNumeroDosesMax = 0;
private String dataFormatada = "";
private String ano = "";
private String mes = "";
private String dia = "";
SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-mm-dd");

//Ip localhost no Android = 10.0.2.2, mesmo que http://localhost:80
public static final String urlAPI = "http://10.0.2.2:8080";

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_inserir_dose);
```

Figura 24 – Associação dos widgets através das *annotations*.

Fonte: Print Screen do trecho de código da classe InserirDoseActivity no Eclipse.

A explicação da utilização do framework Butter Knife dentro do projeto encerra o detalhamento técnico do aplicativo desenvolvido no projeto “Vacinapp”, como também encerra o conteúdo programático desse relatório técnico.

5 CONSIDERAÇÕES FINAIS

A partir do tempo e esforço investido no desenvolvimento desse projeto, foi possível compreender o quão custoso e complexo é o desenvolvimento de um software comercial. Todas as decisões tomadas nesse projeto envolveram conversas com o orientador e foram acordadas para a definição final, apesar da proporção do projeto não ser tão grande quanto a de um software comercial, mas mesmo assim foi possível compreender que em meio ao desenvolvimento de sistemas é preciso sempre se questionar o caminho que se está sendo tomando, quais rumos aquele software irá tomar e se ele não pode ser melhorado em algum de seus aspectos.

Uma lição muito válida aprendida ao término do projeto foi a de que não se deve economizar tempo no processo de definição da arquitetura do sistema e de quais tecnologias serão utilizadas, essas escolhas tem uma grande influência no quão bom será o produto final desenvolvido. Além disso muitas vezes a variável com mais peso é o tempo final de desenvolvimento do projeto e é por isso que determinadas tecnologias podem surgir como um agente de otimização do tempo.

O software desenvolvido nesse projeto foi um MVP que posteriormente será complementado para que as tecnologias envolvidas sejam aprofundadas e novas tecnologias sejam agregadas a ele. Alguns exemplos de melhoras no projeto são:

Implementação do HATEOAS para elevação do grau de maturidade da arquitetura REST do sistema, incluir algum nível de segurança utilizando autenticação por certificado, token ou por OAuth, realizar otimização do código fonte de modo a aprimorar o funcionamento do aplicativo, realizar o versionamento de scripts com o Flyway também é uma opção.

REFERÊNCIAS

AFONSO, Alexandre. **Produtividade no Desenvolvimento de Aplicações Web com Spring Boot**. 3 ed. 2017. Disponível em <http://cafe.algaworks.com/livro-spring-boot/>. Acessado em: 28 Out. 2017.

AFONSO, Alexandre. **O que é Spring Boot ?**. Disponível em < http://blog.algaworks.com/spring-boot/>. Acessado em: Nov. 2017.

ANDROID Async HTTP Clients: Volley vs Retrofit. INSTRUCTURE TECH BLOG. Disponível em <http://instructure.github.io/blog/2013/12/09/volley-vs-retrofit/>. Acessado em: 22 Nov. 2017.

BUILDING a Rest API with Spring 4. Disponível em <http://www.baeldung.com/building-a-restful-web-service-with-spring-and-java-based-configuration>. Acessado em: 15 Nov. 2017.

COMMUNICATING with the UI Thread. Disponível em < https://developer.android.com/training/multiple-threads/communicate-ui.html>. Acessado em: Dez. 2017.

CONSUMING APIs with Retrofit. Disponível em < https://guides.codepath.com/android/consuming-apis-with-retrofit>. Acessado em: Nov. 2017.

CONSUMINDO APIs em Android com Retrofit e Butter Knife. Disponível em < http://www.luiztools.com.br/post/consumindo-apis-em-android-com-retrofit-e-butter-knife/?utm_content=buffer8556d&utm_medium=social&utm_source=twitter.com&utm_campaign=buffer>. Acessado em: 16 Dez. 2017.

FARIAS, Lucas. **Anotações: Meta-annotations, Stereotype Annotation e Composed Annotation**. Disponível em < https://medium.com/@luksrn/anota%C3%A7%C3%B5es-meta-annotations-stereotype-annotation-e-composed-annotations-ee31004a2330>. Acessado em: Out. 2017.

FOWLER, Martin. **Richardson Maturity Model steps toward the glory of REST**. Disponível em < https://martinfowler.com/articles/richardsonMaturityModel.html>. Acessado em: 19 Dez. 2017.

OSAHON, Gino. **Consuming REST API using Retrofit Library in Android**. Disponível em <https://android.jlelse.eu/consuming-rest-api-using-retrofit-library-in-android-ed47aef01ecb>. Acessado em: Nov. 2017.