DIEGO D'LEON NUNES
DIÓGENES APARECIDO REZENDE

APLICATIVO PARA CONSULTA DE NOTAS

UNIVERSIDADE DO VALE DO SAPUCAÍ
POUSO ALEGRE – MG
2015

SUMÁRIO

1	**QUADRO METODOLÓGICO**	**2**
1.1	**Tipo de pesquisa**	**2**
1.2	**Contexto de pesquisa**	**2**
1.3	**Instrumentos**	**3**
1.4	**Procedimentos e Resultados**	**5**
1.4.1	**Modelagem**	**5**
1.4.2	**Google Cloud Messaging**	**5**
1.4.3	**Aplicativo**	**6**
1.4.4	**Web service**	**17**
REFERÊNCIAS		**25**

1 QUADRO METODOLÓGICO

Neste capítulo serão apresentados os métodos adotados para se realizar esta pesquisa, tais como tipo de pesquisa, contexto, procedimentos, entre outros.

1.1 Tipo de pesquisa

Marconi e Lakatos (2002, p.15) definem pesquisa como "uma indagação minuciosa ou exame crítico e exaustivo na procura de fatos e princípios". Gonçalves (2008), por sua vez, conclui que uma pesquisa constitui-se em um conjunto de procedimentos visando alcançar o conhecimento de algo.

Segundo Marconi e Lakatos (2002, p.15), uma pesquisa do tipo aplicada "caracteriza-se por seu interesse prático, isto é, que os resultados sejam aplicados ou utilizados, imediatamente, na solução de problemas que ocorrem na realidade".

Dessa maneira, este projeto enquadra-se no tipo de pesquisa aplicada, pois desenvolveu-se um produto real com intuito de resolver um problema específico, no caso um aplicativo para plataforma Android que permita aos alunos da universidade do Vale do Sapucaí, consultarem suas notas, faltas e provas agendas.

1.2 Contexto de pesquisa

Para que os alunos possam saber suas notas, faltas e provas agendadas, é necessário aos discentes acessarem o portal do aluno para consultá-las.

O *software* desenvolvido nesse trabalho, é um aplicativo para dispositivos móveis com sistema operacional Android, o qual tem por finalidade facilitar aos alunos o acesso as suas informações escolares mais procuradas.

Os alunos acessarão o aplicativo com mesmo usuário e senha do portal do aluno, e quando houver o lançamento de alguma nota ou prova agendada, o estudante será notificado em seu dispositivo. Ao clicar na notificação o sistema lhe apresentará a informação correspondente.

1.3 Instrumentos

Os instrumentos de pesquisa existem para que se possam levantar informações para realizar um determinado projeto.

Pode-se dizer que um questionário é uma forma de coletar informações através de algumas perguntas feitas a um público específico. Segundo Gunther (2003), o questionário pode ser definido como um conjunto de perguntas que mede a opinião e interesse do respondente.

Neste trabalho foi realizado um questionário simples, apresentado na Figura 1, contendo quatro perguntas e enviado para *e-mails* de alguns alunos da universidade. O foco desse questionário era saber o motivo pelo qual os usuários mais acessavam o portal do aluno e se tinham alguma dificuldade em encontrar o que procuravam. Obteve-se um total de treze respostas, no qual pode-se perceber que a maioria dos entrevistados afirmaram ter dificuldades para encontrar as informações de que necessitam, e que gostariam de ser notificados quando houvesse alguma atualização de notas. Sobre o motivo do acesso, cem por cento dos discentes responderam que entram no sistema *web* para consultar os resultados das avaliações.

Outro instrumento utilizado para realizar esta pesquisa foram as reuniões, ou seja, reunir-se com uma ou mais pessoas em um local, físico ou remotamente para tratar algum assunto específico. Para Ferreira (1999), reunião é o ato de encontro entre algumas pessoas em um determinado local, com finalidade de tratar qualquer assunto.

Durante a pesquisa, foram realizadas reuniões entre os participantes com o objetivo de discutir o andamento das tarefas pela qual cada integrante responsabilizou-se a fazer e traçar novas metas. Também foram utilizadas referências de livros, revistas, manuais e *web sites*.

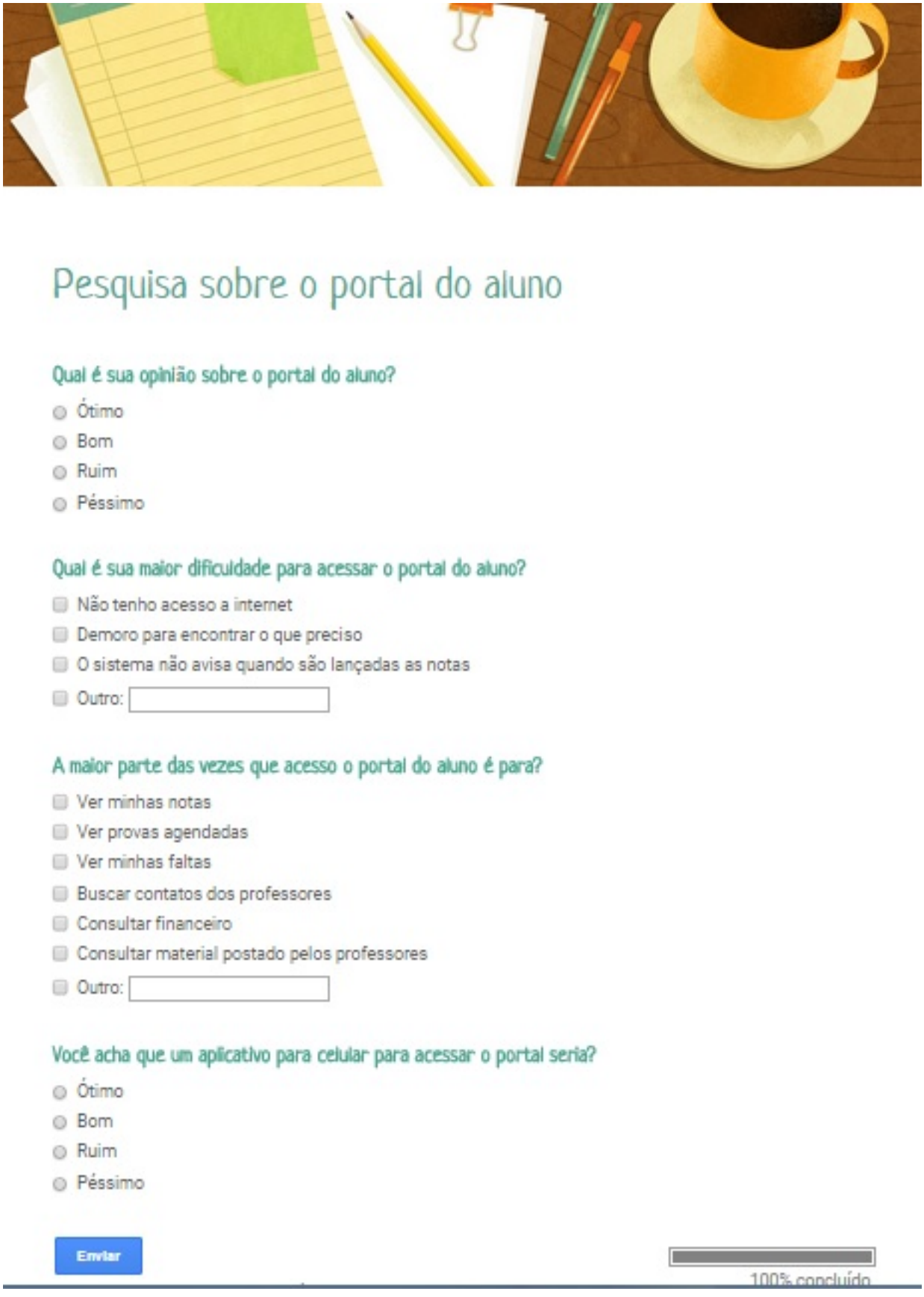

Figura 1 – Quetionário Aplicado. **Fonte:**Elaborado pelos autores.

1.4 Procedimentos e Resultados

Após estudar as teorias de desenvolvimento de *software* e integração entre *web service* e aplicativos *Android*, iniciou-se o período de modelagem do sistema.

1.4.1 Modelagem

Para atender o objetivo proposto por esta pesquisa, necessitou-se antes modelar o *software* através dos diagramas de UML.

1.4.2 Google Cloud Messaging

O envio dos dados do *web service* para o aplicativo *Android*, é feito através de um serviço da *Google* conhecido como GCM.

Para que o serviço apresente o resultado esperado, foi preciso acessar o *site* da *Google Developers Console* e criar um novo projeto. Ao criá-lo, foi necessário ir na aba API's e ativar a opção *Google Cloud Messaging for Android*.

Com a criação do projeto, a *Google* oferece um número que identificará o *software*, também chamado de `Sender ID`, conforme mostra a Figura 2.

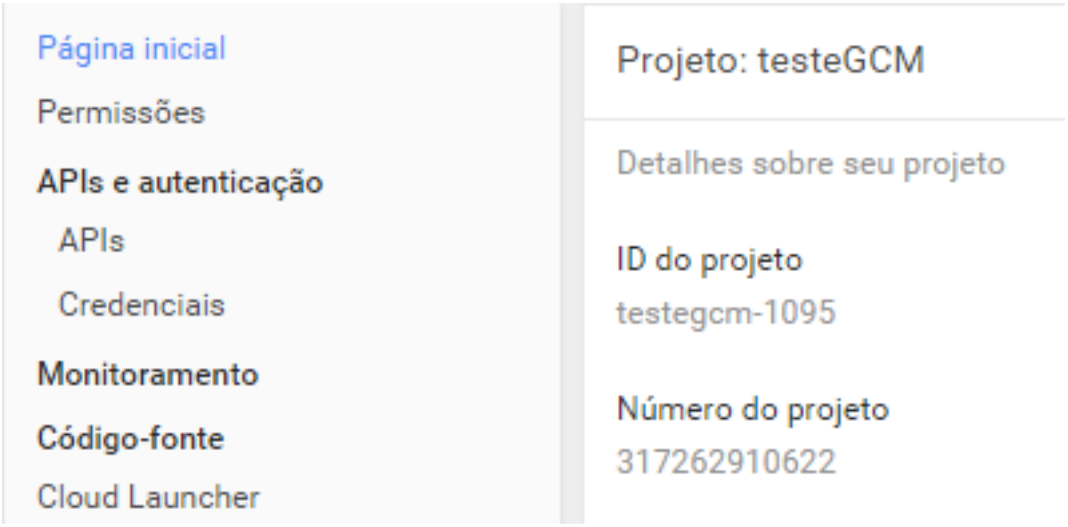

Figura 2 – `Sender ID` do GCM. **Fonte:**Elaborado pelos autores.

Por fim, acessou-se a aba Credenciais para indicar o IP do servidor. Ao informa-lo, o serviço gerou uma chave pública a qual foi inserida no *web service*. Na Figura 3, é possível ver o código de acesso criado.

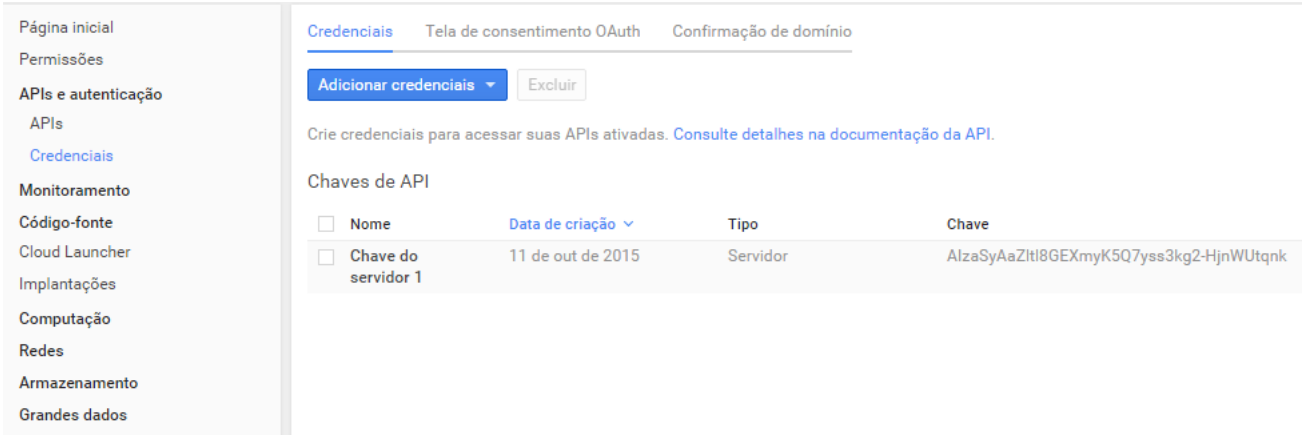

Figura 3 – Geração da credencial do GCM. **Fonte:**Elaborado pelos autores.

1.4.3 Aplicativo

Para iniciar a construção do aplicativo, fez-se necessário a instalação e configuração do ambiente de desenvolvimento. Primeiramente, realizou-se o *download* da IDE *Android Studio*, versão 1.1.0 e do *Android* SDK, versão 24.0.2, ambos no site *Developers Android*.

Contudo, ao executar o emulador do *Android* o sistema apresentava a seguinte mensagem: *"emulator: Failed to open the HAX device!"*. Depois de algum tempo pesquisando, percebeu-se que era necessário instalar um programa chamado *Intel Hardware Accelerated Execution Manager* (HAXM), responsável por aumentar a performance do emulador.

No entanto, ao instalá-lo ocorria o seguinte erro: "*this computer meets the requirements for haxm but intel virtualization technology (VT-x) is not turned on*". A solução foi acessar a BIOS da máquina e habilitar o assistente de *hardware* para virtualização. Daí em diante, foi possível executar no emulador as aplicações feitas no *Android Studio*.

Com o ambiente já configurado, criou-se um repositório no controlador de versão *Github*, cujo todos os participantes possuem acesso, para que ambos tenham a versão mais recente do aplicativo em seu dispositivo.

A partir de então, passou-se a desenvolver o *software*. A princípio, foi construída uma `activity`, que é acessível ao aluno logo que a aplicação se inicia. Essa `activity` é do tipo

Navigation Drawer Layout, ou seja, é um painel que permite inserir as opções de navegação do aplicativo, semelhante a um menu.

Ao criar essa `activity`, o *Android Studio* gera automaticamente a classe `NavigationDrawerFragm` e um arquivo XML na pasta `layout`, chamado `fragment_navigation_drawer.xml`.

No arquivo `fragment_navigation_drawer.xml` foram inseridos três *widgets*, sendo dois do tipo `textView`, para o cabeçalho com a logomarca da Univás e para o rodapé com o seguinte texto: "Univás - Pouso Alegre - MG"e um *widget* do tipo `listView` que contém a lista com as opções que o *software* oferece ao aluno. Na Figura 4, podem ser vistos os *widgets* no arquivo `fragment_navigation_drawer.xml`

```
<TextView
    android:id="@+id/headerView"
    style="?android:attr/textAppearanceLarge"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:drawableLeft="@drawable/logo1"
    android:gravity="center_vertical"
    android:padding="25dp"
    android:text=""
    android:layout_alignParentTop="true"
    android:layout_alignParentStart="true" />
<TextView
    android:id="@+id/footerView"
    style="?android:attr/textAppearanceMedium"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:gravity="center"
    android:padding="20dp"
    android:text="Univás - Pouso Alegre - MG"
    android:textStyle="bold" />
<ListView
    android:id="@+id/navigationItems"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_above="@+id/footerView"
    android:layout_below="@+id/headerView"
    android:background="#cccc"
    android:choiceMode="singleChoice"
    android:divider="@android:color/transparent"
    android:dividerHeight="0dp" />
```

Figura 4 – sem legenda. **Fonte:**Elaborado pelos autores.

A classe `NavigationDrawerFragment` representa o painel de navegação. Nela se destacam os métodos `onCreateView()`, responsável por criar o *layout* de navegação e o `selectItem()`, encarregado em identificar qual item foi escolhido pelo usuário. Na figura 5, vê-se o método `onCreateView()`, informando ao sistema operacional o *layout* a ser chamado e adicionando a um `array` de *String* as alternativas de navegação que serão exibidos no `listView` do arquivo `fragment_navigation_drawer.xml`.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    View view = inflater.inflate(
            R.layout.fragment_navigation_drawer, container, false);

    mDrawerListView = (ListView) view.findViewById(R.id.navigationItems);
    mDrawerListView.setOnItemClickListener((parent, view, position, id) -> {
            selectItem(position);
    });
    mDrawerListView.setAdapter(new ArrayAdapter<String>(
            getActionBar().getThemedContext(),
            android.R.layout.simple_list_item_activated_1,
            android.R.id.text1,
            new String[]{
                    "Home",
                    "Notas",
                    "Faltas",
                    "Provas Agendadas",
                    "Sair"
            }));
    mDrawerListView.setItemChecked(mCurrentSelectedPosition, true);
    return view;
}
```

Figura 5 – sem legenda. **Fonte:**Elaborado pelos autores.

O próximo passo, foi criação de uma `activity` do tipo `blank activity` com finalidade de listar as notas. Ao criá-la com o nome de `ListResultsActivity`, o *Android Studio* gera dentro da pasta *layout* o arquivo XML referente a ela, chamado de `activity_list_results.xml`. Neste, foi inserido apenas o *widget* `expandableListView`, que está incumbido de apresentar a lista de disciplinas cujo o discente está cursando e ao clicar em alguma dessas matérias serão apresentadas as notas referentes aos exercícios realizados desta disciplina. Na Figura 6 é possível ver o *layout* com uma lista do tipo `expandableListView`.

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" android:layout_width="match_parent"
    android:layout_height="match_parent" android:paddingLeft="16dp"
    android:paddingRight="16dp"
    android:paddingTop="16dp"
    android:paddingBottom="16dp"
    tools:context="univas.edu.com.university.ListResultsActivity">

    <ExpandableListView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/expandableListView2"
        android:layout_alignParentBottom="true"
        android:layout_alignParentStart="true"
        android:layout_alignParentTop="true" />

</RelativeLayout>
```

Figura 6 – sem legenda. **Fonte:**Elaborado pelos autores.

Depois, fez-se necessário a criação de uma classe encarregada por organizar e controlar todas as atualizações dos itens de uma lista. Essa classe recebeu o nome de `ListResultsAdapter` e estende da classe `BaseExpandableListAdapter`, nativa do *Android*.

Os procedimentos acima citados foram necessários também para as opções de faltas e provas agendadas. Após construídos todos os *layouts*, foi fundamental criar o banco de dados, no qual o aplicativo salva as informações recebidas do *web service*. Para que isso fosse possível, elaborou-se uma classe denominada `DatabaseHelper` que estende da classe `SQLiteOpenHelper` do *Android*, com dois métodos, um chamado `onCreate()` e outro conhecido por `onUpgrade()`.

Foi preciso criar um atributo que mantém a versão do banco de dados. Essa informação serve para que o *Android* consiga saber qual dos dois métodos devem ser executados. Ao iniciar a aplicação pela primeira vez, estando a versão em um, o sistema chamará o método `onCreate()`. Se for preciso atualizar a estrutura do banco, o atributo versão deve ser incrementado em um, de modo que ao executar o *software* o sistema operacional perceba a mudança, chamando o método `onUpgrade()`. Na figura 7 é apresentado a classe `DatabaseHelper`.

```
public class DatabaseHelper extends SQLiteOpenHelper {

    private static final String BANCO_DADOS = "univasDB_version1";
    private static int VERSAO = 1;

    public DatabaseHelper(Context context) { super(context, BANCO_DADOS, null, VERSAO); }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE disciplinas (_id LONG PRIMARY KEY, nome TEXT);");

        db.execSQL("CREATE TABLE eventos (_id LONG PRIMARY KEY, id_disciplina LONG, " +
                " tipo_evento TEXT, descricao_evento TEXT," +
                " data_evento TEXT, valor_evento INTEGER, nota INTEGER,"  +
                " FOREIGN KEY (id_disciplina) REFERENCES disciplinas (_id) );");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int i, int i2) {

    }
}
```

Figura 7 – sem legenda. **Fonte:**Elaborado pelos autores.

A fim de estabelecer uma conexão entre o aplicativo e *web service* foi preciso conceder a permissão de acesso à internet no `AndroidManifest.xml` com o seguinte comando: `<uses-permission android:name="android.permission.INTERNET"/>`.

Logo após, criou-se uma classe chamada de `HttpUtil` para ler informações recebidas *web service*. Nela foram inseridos dois métodos, um para identificar as informações referentes as disciplinas cursadas e outro para captar os dados de eventos como notas, faltas e provas agendadas. Na Figura 8 é possível ver o método incumbido de interpretar os elementos das matérias.

```
public HttpUtil(DatabaseHelper helper) { this.helper = helper; }

public void getJsonDiscipline(final String url){
    new Thread((Runnable) () -> {
            AlunoDisciplina retorno = null;
            try {
                HttpClient httpClient = new DefaultHttpClient();
                HttpGet request = new HttpGet();
                request.setURI(new URI(url));
                HttpResponse response = null;
                try {
                    response = httpClient.execute(request);
                } catch (IOException e) {
                    e.printStackTrace();
                }
                InputStream content = null;
                try {
                    content = response.getEntity().getContent();
                } catch (IOException e) {
                    e.printStackTrace();
                }
                Reader reader = new InputStreamReader(content);
                Gson gson = new Gson();
                retorno = gson.fromJson(reader, AlunoDisciplina.class);
                for (int i = 0; i< retorno.getDisciplinas().size(); i++){
                    DatabaseExecute execute = new DatabaseExecute(helper);
                    DisciplineTO to = new DisciplineTO();
                    to.set_id(retorno.getDisciplinas().get(i).getId());
                    to.setNome(retorno.getDisciplinas().get(i).getNome());
                    execute.insertDiscipline(to);
                }
                content.close();
```

Figura 8 – sem legenda. **Fonte:**Elaborado pelos autores.

Nos métodos de leitura dos dados foi preciso criar uma `thread` separada da `thread` principal do sistema, evitando travar a aplicação enquanto recebe as informações vindas do *web service*. Estes dados estão em formato JSON e foi utilizado a biblioteca Gson para convertê-las no formato esperado. Para utilizá-la foi fundamental adicioná-la como uma dependência do projeto, conforme mostra a Figura 9.

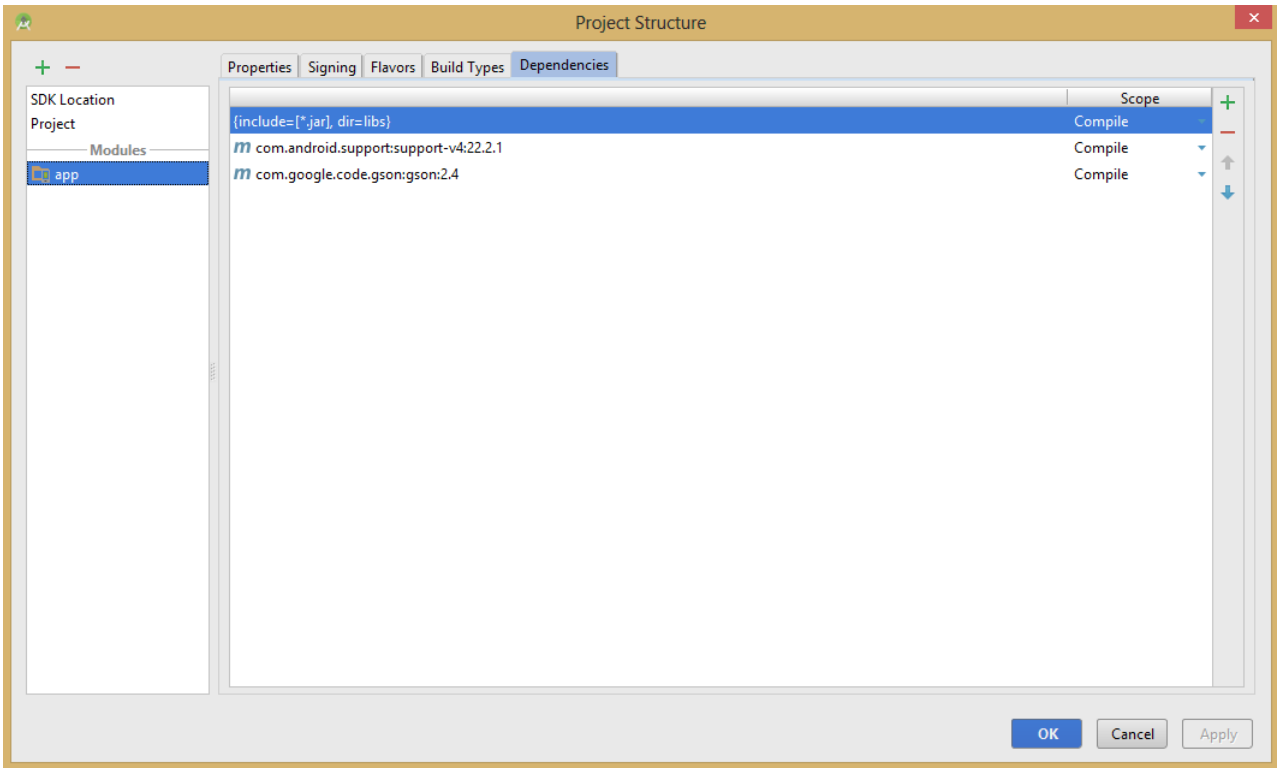

Figura 9 – sem legenda. **Fonte:**Elaborado pelos autores.

Ao receber as informações do servidor *web* é preciso salvá-las no banco de dados do aplicativo. Para isso desenvolveu-se uma classe chamada `DatabaseExecute`, encarregada de inserir, alterar e buscar os dados dos estudantes no banco de dados. Na Figura 10, pode se ver o método pelo qual são inseridos os eventos ocorridos. Esses eventos podem ser notas, faltas ou provas agendadas.

```
public void insertEvents(EventTO to){
    SQLiteDatabase db = helper.getWritableDatabase();

    ContentValues values = new ContentValues();
    values.put("_id", to.get_id());
    values.put("id_disciplina", to.getId_disciplina());
    values.put("tipo_evento", to.getTipo_evento());
    values.put("descricao_evento", to.getDescricao_evento());
    values.put("data_evento", to.getData_evento());
    values.put("valor_evento", to.getValor_evento());
    values.put("nota", to.getNota());

    long result = db.insert("eventos", null, values);

    if(result != -1 ){
        Log.d(TAG, " Evento salvo com sucesso! Tipo evento= " + to.getTipo_evento() + " idDisciplina= "+ to.getId_disciplina());
    }else{
        Log.d(TAG, " Erro ao salvar o Evento!");
    }
}
```

Figura 10 – sem legenda. **Fonte:**Elaborado pelos autores.

O método recebe um objeto do tipo `EventTO`. Para que seja possível a inserção dos dados, Monteiro (2012) afirma, que é necessário recuperar a referência da classe `SQLiteDatabase`, através do método `getWritableDatabase()`, logo após é instanciada a classe `ContentValues`, onde é informado o campo da tabela e o valor desejado. Ao concluir é chamado o `insert` da classe `SQLiteDatabase` informando o nome da tabela e o objeto da classe `ContentValues`.

Para listar os resultados dos exames realizados pelos discentes no painel de notas é utilizado o método `getResults()`, da classe `DatabaseExecute`. Ele retorna uma lista de objetos da classe `EventTO`. De acordo com Monteiro (2012), para conseguir recuperar as informações armazenadas no banco de dados é preciso adquirir a instância de leitura da classe `SQLiteDatabase` através do método `getReadableDatabase()`. Por meio dele pode-se realizar a consulta e recebe um `Cursor` para navegar pelos resultados. Por fim, é composto um objeto do tipo `EventTO` e inserido na lista. Na Figura 11 é apresentado o método `getResults()`.

Para que as informações possam aparecer na tela, a classe `ListResultsActivity` deve informar ao sistema operacional o *layout* a ser chamado através do método `setContentView()`. Por fim, deve recuperar uma instância do *widget* que apresentará os dados, no caso `expandableListView`, e passar para a classe com função de `Adapter` a lista de disciplinas cursadas e as notas, para que ela possa fazer a organização das informações. Como pode-se ver na Figura 12.

```
public List<EventTO> getResults(){
    List<EventTO> notasTO = new ArrayList<>();

    SQLiteDatabase db = helper.getReadableDatabase();
    Cursor cursor =
            db.rawQuery("SELECT _id, id_disciplina, descricao_evento, valor_evento, nota  FROM" +
                    " eventos WHERE tipo_evento = 'PROVA_APLICADA'",
                    null);
    cursor.moveToFirst();

    for(int i = 0; i<cursor.getCount();i++){
        EventTO nota = new EventTO();

        nota.set_id(cursor.getLong(0));
        nota.setId_disciplina(cursor.getLong(1));
        nota.setDescricao_evento(cursor.getString(2));
        nota.setValor_evento(cursor.getInt(3));
        nota.setNota(cursor.getInt(4));

        notasTO.add(nota);
        cursor.moveToNext();
    }

    cursor.close();

    return notasTO;
}
```

Figura 11 – sem legenda. **Fonte:**Elaborado pelos autores.

```
public class ListResultsActivity extends Activity {

    private DatabaseHelper helper;
    private DatabaseExecute execute;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_list_results);
        helper = new DatabaseHelper(this);

        execute = new DatabaseExecute(helper);

        ExpandableListView listView = (ExpandableListView) findViewById(R.id.expandableListView2);
        listView.setAdapter(new ListResultsAdapter(this, execute.getDisciplines(), execute.getResults()));
    }
```

Figura 12 – sem legenda. **Fonte:**Elaborado pelos autores.

Na classe `ListResultsAdaper` as informações referentes as matérias são inseridas em um vetor de *string* enquanto as notas são inseridas em uma matriz, conforme ilustra a Figura 13. Ao findar esse processo, utiliza-se o método `getGroupView()` para apresentar os nomes das disciplinas e o método `getChildView()` para mostrar as notas da matéria desejada, como demostra a Figura 14.

```
public ListResultsAdapter(Context context, List<DisciplineTO> disciplinas, List<EventTO> notas){
    this.context = context;
    this.disciplinas = disciplinas;
    this.notas = notas;
    nomesMateria = new String[disciplinas.size()];
    valoresMateria = new String[disciplinas.size()][notas.size()];

    for (int i = 0; i < disciplinas.size(); i++){
       Long idMateria = disciplinas.get(i).get_id();
       String nomeMateria = disciplinas.get(i).getNome();
       nomesMateria[i] = nomeMateria;
        int position = 0;
        for (int y = 0; y < notas.size(); y++) {
            String descricao = notas.get(y).getDescricao_evento();
            int valor = notas.get(y).getValor_evento();
            int nota = notas.get(y).getNota();
            Long idDisciplina = notas.get(y).getId_disciplina();

            if (idMateria == idDisciplina){
                valoresMateria [i][position] = "Descrição: " + descricao + " Valor: " + valor + " Nota: " + nota;
                position++;
            }
        }

    }
```

Figura 13 – sem legenda. **Fonte:**Elaborado pelos autores.

```
@Override
public View getGroupView(int groupPosition, boolean isExpanded,
                         View convertView, ViewGroup parent) {

    TextView textViewCategorias = new TextView(context);
    textViewCategorias.setText(nomesMateria[groupPosition]);
    textViewCategorias.setPadding(30, 5, 0, 5);
    textViewCategorias.setTextSize(20);
    textViewCategorias.setTypeface(null, Typeface.BOLD);

    return textViewCategorias;
}

@Override
public View getChildView(int groupPosition, int childPosition,
                         boolean isLastChild, View convertView, ViewGroup parent) {

    TextView textViewSubLista = new TextView(context);
    textViewSubLista.setText(valoresMateria[groupPosition][childPosition]);
    textViewSubLista.setPadding(10, 5, 0, 5);

    return textViewSubLista;
}
```

Figura 14 – sem legenda. **Fonte:**Elaborado pelos autores.

Nesse contexto, criou-se uma classe denominada `GcmControllerUnivas`, que verifica se o aparelho em que o aplicativo está instalado é compatível com os requisitos do GCM. Caso o dispositivo esteja com as configurações recomendadas, é executado o método `registerInBackground()`, para que possa gerar a chave de registro na *Google*. Ao receber essa chave é chamado o método `sendRegistrationIdToBackend()`, encarregado de enviar esse código para o *web service*. Na Figura 15, vê-se o método `registerInBackground()`, recebendo na variável `regid`, a

chave de registro do aparelho, realizado pela classe nativa `GoogleCloudMessaging`, ao ser passado o id do projeto gerado na `Google Developers Console`.

```
private void registerInBackground() {

    new AsyncTask<Void, Void, String>() {
        @Override
        protected String doInBackground(Void... params) {
            String msg = "";
            try {
                if (gcm == null) {
                    gcm = GoogleCloudMessaging.getInstance(context);
                }
                regid = gcm.register(SENDER_ID);
                msg = "Dispositivo registrado, registro ID=" + regid;

                //...
                sendRegistrationIdToBackend(regid);

                //...
                storeRegistrationId(context, regid);
            } catch (IOException ex) {
                msg = "Error :" + ex.getMessage();
                //...
            }
            return msg;
        }
```

Figura 15 – sem legenda. **Fonte:**Elaborado pelos autores.

Desta forma, quando o *web service* envia uma informação ao GCM, é transmitido junto aos dados o `Registration ID`, gerado pelo método *registerInBackground()*, possibilitando ao serviço da *Google*, identificar a qual dispositivo deve conduzir a mensagem.

Ao receber os registros enviados pelo GCM o `broadcastReceiver` chama a classe que apresenta ao usuário a notificação, no entanto, antes de notificá-lo a informação recebida é enviada à classe `HttpUtil` que faz a leitura dos dados e os salva no banco de dados. Ao findar esse processo é analisada se a informação recebida é de notas, faltas ou provas agendadas e chama o método `sendNotification()` que recebe um objeto de `EventTO` e o tipo do evento.

Por fim, adiciona em uma lista de `String` as informações que devem ser apresentadas aos estudantes, passando-as para a `activity` responsável em exibí-las. A notificação, por sua vez, é exposta através do comando retratado na Figura 16, onde identifica-se os atributos da notificação como a ícone que aparecerá, o título e a mensagem. O método `notify()`, é responsável por fazer a notificação aparecer.

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this).setSmallIcon(R.drawable.notification_univas)
.setContentTitle("Univas informa")
.setAutoCancel(true)
.setStyle(new NotificationCompat.BigTextStyle()
.bigText(msg))
.setContentText(msg);

mBuilder.setContentIntent(contentIntent);
mNotificationManager.notify(NOTIFICATION_ID, mBuilder.build());
```

Figura 16 – sem legenda. **Fonte:**Elaborado pelos autores.

1.4.4 Web service

Nesta seção serão descritos os procedimentos realizados para o desenvolvimento do *web service* responsável por prover os dados necessários ao aplicativo. Além disso serão descritas as configuração necessárias para a montagem do ambiente de desenvolvimento e sua posterior implantação.

1.4.4.1 Montagem do Ambiente de Desenvolvimento

No que diz respeito à contrução do *web service*, foi necessária a instalação e configuração de um ambiente de desenvolvimento compatível com as necessidades apresentadas pelo

software.

A princípio foi instalado o Servlet Container Apache Tomcat em sua versão de número 7. Esse Servlet Container foi instalado pois implementa a API da especificação Servlets 3.0 do Java. Isso era necessário pelo fato que o *framework* Jersey usa *servlets* para disponibilizar serviços REST. Além disso o Apache Tomcat foi escolhido, para que o *web service* pudesse fornecer os serviços necessários para o consumo, na arquitetura REST, que sugere o uso do protocolo HTTP[1] para troca de mensagens, pois além da funcionalidade com Servlets, o Apache Tomcat também é um servidor HTTP.

O Apache Tomcat foi instalado, por meio do *download* de um arquivo compactado no site oficial do mesmo. A instalação consiste apenas em extrair os dados do arquivo em uma pasta da preferência do desenvolvedor. Esta abordagem permitiu a integração do Apache Tomcat com o IDE[2] Eclipse, que foi usada para o desenvolvimento. Com isto foi possível controlar e monitorar, o servidor de aplicações através da IDE. Além da configuração necessária para integrar o servidor à IDE, nenhuma outra configuração foi necessária.

Como ferramenta para desenvolvimento, foi usada a IDE Eclipse na versão 4.4, que é popularmente conhecida como Luna. O processo de instalação e configuração da IDE, se assemelha bastante ao processo de instalação do Apache Tomcat, pois somente é necessário fazer o download do arquivo compactado que é fornecido na página do projeto, e descompactá-lo no local preterido pelo desenvolvedor.

Para armazenar os dados gerados e/ou recebidos, foi necessário fazer a intalação do Sistema Gerenciador de Banco de Dados(SGBD) PostGreSql na sua versão de número 9.4. Como está sendo usado um sistema operacional baseado em GNU/Linux como ambiente de desenvolvimento, o PostGreSql foi instalado através do gerenciador de pacotes da distribuição.

1.4.4.2 Desenvolvimento

Com o ambiente de desenvolvimento pronto, começou de fato o desenvolvimento. Primeiramente foi necessário criar o banco de dados no SGDB. Este por sua vez foi criado com a ajuda do PgAdmin que é um software gráfico para administração do SGDB, e que fornece uma interface gráfica de apoio para o PotgreSql. Para criar era necessário ja estar com o PgAdmin aberto e conectado a um servidor de banco de dados que neste caso era em servidor local como pode ser visto na Figura 17.

[1] HTTP - Hypertext Transfer Protocol
[2] IDE - Integrated Development Environment

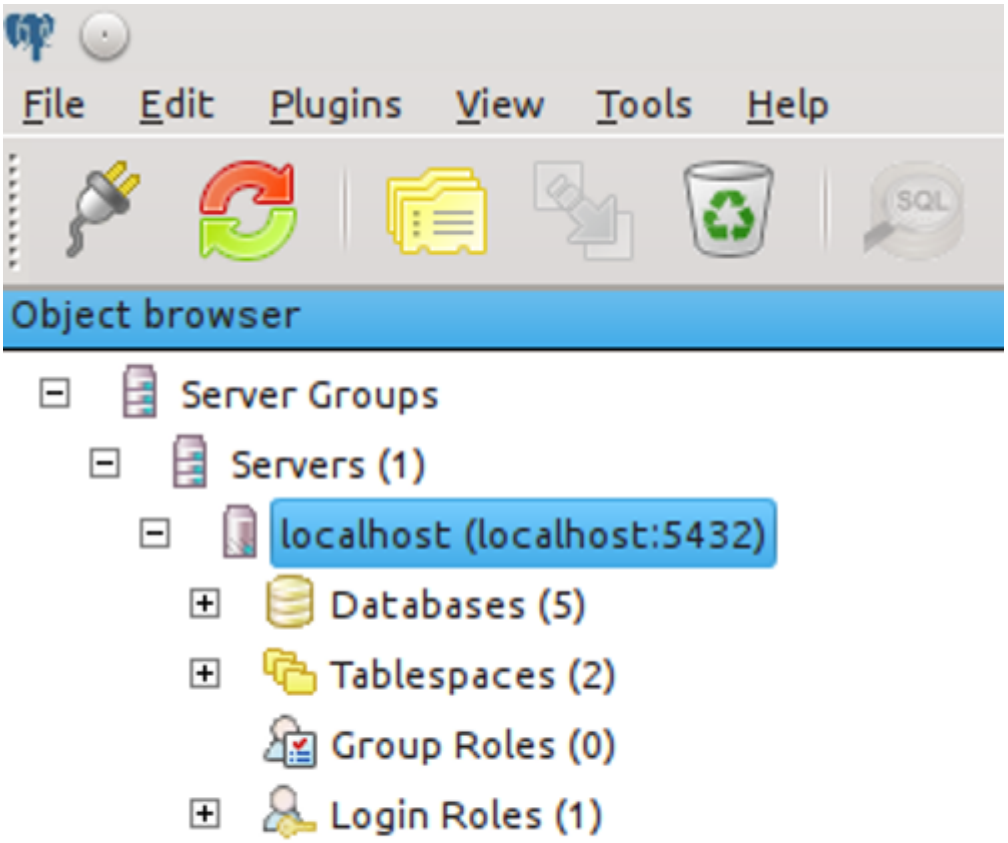

Figura 17 – Servidor de banco de dados local no PgAdmin. **Fonte:**Elaborado pelos autores.

Para a efetiva criação do banco de dados era necessário clicar com o botão direito do *mouse*, sobre a opção *Databases>New Database...* no PgAdmin. Em seguida foi necessário preencher o dados da janela apresentada, como está apresentado na Figura 18.

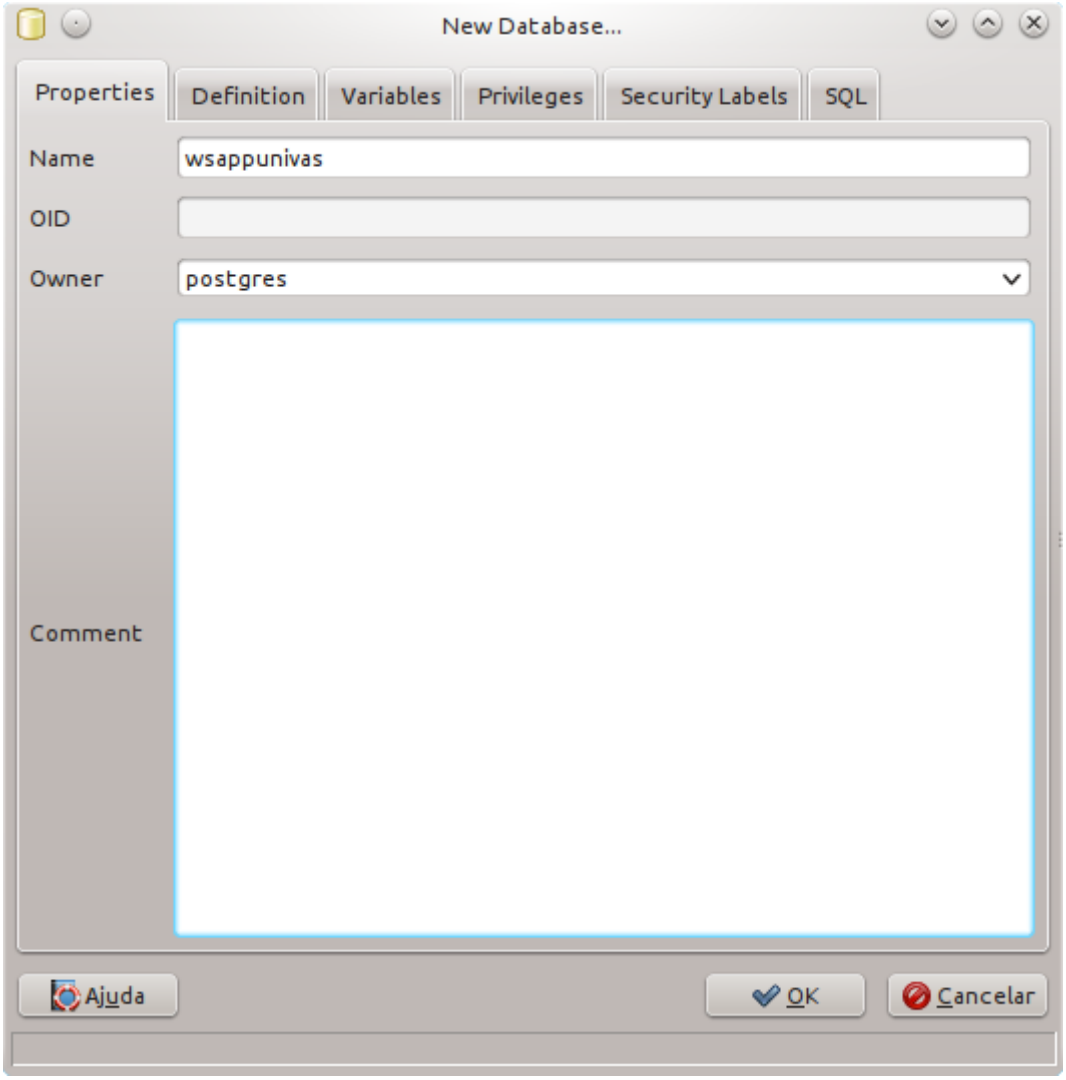

Figura 18 – Tela *New Database.*... **Fonte:**Elaborado pelos autores.

Como pode ser visto foram preenchidos os campos nome e usuário . O campo nome se refere ao nome do banco de dados que foi definido com `wsappunivas`, e o usuário responsável pelo banco, que para este caso foi usuário padrão do SGDB, que é o `postgres`. Além destas configurações mais nenhuma foi necessária. O banco foi criado, porém sua estrutura não foi definida, pois como será visto mais adiante o Hibernate, possui um mecanismo, que com algumas configurações, permite a estruturação do banco de dados, de acordo com o mapeamento objeto-relacional. Isto permitirá mudanças na estrutura do banco de dados e suas tabelas, e até mesmo eventuais correções.

Em seguida foi criado um projeto do tipo Dynamic Web Project no Eclipse.

com a ajuda do plugin Maven. Esse Projeto foi criado usando o Maven pois depende de uma quantidade considerável de *frameworks*, e uma das principais funcionalidades deste plugin, é ajudar na resolução das dependências de um projeto Java.

Para tal projeto foi necessário a configuração do `POM.xml` que é o arquivo utilizado pelo Maven. Nele estão contidas as configurações relativas à compilação do projeto bem como suas dependências. Na figura a seguir pode ser visto o conteúdo do arquivo `POM.xml`.

Com a estrutura do projeto devidamente criada foi possível iniciar os trabalhos com a camada de persistência de dados do projeto. Para este propósito, primeiramente foi criado um pacote, onde ficaram contidas as classes que representam as entidades do ORM. O pacote recebeu o nome de "`br.edu.univas.restapiappunivas.model`", pois nele estão contidas as classes que fazem parte do modelo de negócios da aplicação. Este pacote foi criado visando a divisão das responsabilidades internas no projeto, além de contribuir positivamente com a organização do mesmo. Tal pacote e as classes que o compõe estão representandos na figura.

Com este pacote criado, ja era possível criar as classes do ORM. Foi criada primeiramente a classe `Student.java`. Para que esta classe pudesse ser reconhecida como um entidade e persistida ao banco de dados através do Hibernate, é necessário que esta classe tivesse a anotação `@Entity`. Com isso esta classe ja poderia ser entendida como uma entidade e poderia ser persistida no banco de dados, porém além dessa anotação outras foram usadas para que a persistência pudesse ocorrer de forma consistente.

Fazendo uso desse diagrama foi possível criar todas as classes Java que representam as entidades do mapeamento objeto-relacional. Essas classes foram criadas fazendo uso de anotações próprias do Hibernate, que é um *framework* que implementa a especificação JPA[3]. Essas classes fazem parte dos mecanismos de persistêcia de dados e são simplesmente t ou seja, objetos simples que contêm somente atributos privados e os métodos *getters* e *setters* que servem apenas para encapsular estes atributos. Uma das classes criadas, foi a classe `Aluno.java` que representa a tabela `alunos` no banco de dados e está representada.

Foram criadas outras classes Java com a mesma finalidade da anterior, porém com pequenas diferenças no que diz respeito à atributos, metodos e anotações. Estas classes representam, de maneira individual, as tabelas no banco de dados. Certos atributos dessas classes têm por finalidade representar as colunas de cada tabela. Já os atributos que armazenam instâncias de outras classes ou até mesmo conjuntos (coleções) de instâncias representam os relacionamentos entre as tabelas.

E por fim, para cada classe querepresenta uma entidade, foi necessário implementar os métodos `hashCode` e `equals`, para que estas pudessem facilmente ser comparadas e diferenciadas em relação aos seus valores, haja visto que cada instância destas classes representa um registro no banco de dados.

[3] JPA - Java Persistense API

Em seguida à criação das entidades, foi necessário configurar o arquivo `persistence.xml` que fica dentro do *classpath* do projeto Java ou seja, dentro da mesma pasta onde estão contidos pacotes do projeto. Este arquivo é extremamente importante, pois é nele que estão todas as configurações relativas à conexão com o banco de dados, configurações referentes ao Dialeto SQL que vai ser usado para as consultas e configurações referentes ao *persistence unit* que é o conjunto de classes mapeadas para o banco de dados. O arquivo `persistence.xml` está exposto no código 19.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="WsAppUnivas" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/wsappunivas" />
            <property name="javax.persistence.jdbc.user" value="postgres" />
            <property name="javax.persistence.jdbc.password" value="password" />

            <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
            <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.format_sql" value="true" />
            <property name="hibernate.temp.use_jdbc_metadata_defaults" value="false"></property>
            <property name="hibernate.hbm2ddl.auto" value="create" />
        </properties>
    </persistence-unit>
</persistence>
```

Figura 19 – Arquivo `persistence.xml`. **Fonte:**Elaborado pelos autores.

Em seguida à confecção do `persistence.xml` foi criada a classe `JpaUtil` que está representada na Figura 20. Esta classe é responsável por criar uma `EntityManagerFactory` que é uma fábrica de instâncias de `EntityManager` que nada mais é que um *persistence unit* ou unidade de persistência. Essa classe tem a responsabilidade de prover um modo de comunicação entre a aplicação e o banco de dados. No entanto a classe `JpaUtil` cria uma única instância de `EntityManagerFactory`, que é responsável por disponibilizar e gerenciar as instâncias de `EntityManager` de acordo com a necessidade da aplicação.

```java
public class JpaUtil {
    private static EntityManagerFactory factory;

    static {
        factory = Persistence.createEntityManagerFactory("WsAppUnivas");
    }

    public static EntityManager getEntityManager() {
        return factory.createEntityManager();
    }

    public static void close() {
        factory.close();
    }

}
```

Figura 20 – Classe `JpaUtil`. **Fonte:**Elaborado pelos autores.

Em seguida à construção das classes que fazem a parte da persistência de dados, foi desenvolvido a parte de disponibilização de serviços RESTful, fazendo uso do *framework* Jersey. Com isso pode-se construir a classe que representa o primeiro serviço do *webservice*, que é a classe `Alunos`. Essa classe representa um contexto REST, e portanto, dispõe de alguns recursos. Esses recursos fazem a recuperação e a transmissão dos dados do *web service* para o aplicativo Android. Essa classe e seus respectivos métodos estão representada na Figura 21.

```java
@Path("/alunos")
public class AlunosService {

    /*
     * Busca um Aluno e a suas informações e eventos
     */
    @GET
    @Path(" /{ $cod } ")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Alunos getAlunoById(@PathParam("cod") Long idAluno) {
        Alunos alunos = new Alunos();
        AlunoCtrl ctrl = new AlunoCtrl();
        alunos.setAlunos(ctrl.getById(idAluno));

        return alunos;
    }
    /*
     * Busca os eventos de um Aluno pelo seu id
     */
    @GET
    @Path("/eventos/{ $cod }")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Eventos getAll(@PathParam("cod") Long idAluno) {
        Eventos eventos = new Eventos();
        EventoCtrl ctrl = new EventoCtrl();
        eventos.setEventos(ctrl.getById(idAluno));
        return eventos;
    }
}
```

Figura 21 – Classe `AlunosService`. **Fonte:**Elaborado pelos autores.

O *webservice* pode fazer a busca de alunos pelo `id` passado ou retornar uma coleção de eventos vinculados a um alunos, dependendo do recurso acessado. Os tipos de dados que o *webservice* consome e retorna é o JSON[4]. Não foi necessário fazer nenhuma implementação

[4] JSON - Javascript Object Notation

adicional relativa a este formato, pois o próprio *framework* Jersey faz o tratamento e a conversão dos tipos de entrada e saída de dados. No caso do saída de dados, faz a conversão de objetos Java para JSON. E no caso de entrada tranforma um JSON em objeto Java já conhecido pelo *web service*. Com isso concluiu-se o desenvolvimento do *web service* que fornece os dados para o aplicativo.

Para que fosse possível transmitir dados para o aplicativo, era necessário receber as informações do sistema acadêmico da referida instituição, haja vista que o *web service* é independente do mesmo. Para esse propósito é necessário contruir um módulo que faça a importação dos dados necessários para a base de dados do *web service*.

Este por sua vez terá a responsabilidade de fazer a importação dos dados periodicamente, e ainda tratar os tipos de dados recebidos para tipos aplicáveis ao banco de dados local. Além disso é preciso notificar o módulo responsável por invocar o serviço Google Cloud Messaging para que os dispositivos dos alunos aos quais houveram atualizações nos dados, fossem notificados e fizessem acesso ao *web service* para solicitar esses dados atualizados.

Os procedimentos acima citados foram os passos até agora realizados com o propósito de se alcançar os resultados esperados para essa pesquisa.

REFERÊNCIAS

FERREIRA, A. B. H. : **Novo Aurélio Século XXI:** o dicionário da língua portuguesa. 3ª. ed. Rio de Janeiro: Nova Fronteira, 1999.

GONçALVES, J. A. T. : **O que é pesquisa? Para que?** 2008. Disponível em: <http://metodologiadapesquisa.blogspot.com.br/2008/06/pesquisa-para-que.html>. Acesso em: 07 de Outubro de 2015.

GUNTHER, H. : **Como Elaborar um Questionário.** 2003. Disponível em: <http: //www.dcoms.unisc.br/portal/upload/com_arquivo/como_elaborar_um_questionario.pdf>. Acesso em: 15 de Abril de 2015.

MARCONI, M. A.; LAKATOS, E. M. : **Técnicas de pesquisas:** planejamento e execução de pesquisas, amostragens e técnicas de pesquisas, elaboração, análise e interpretação de dados. 5ª. ed. São Paulo: Atlas, 2002.

MONTEIRO, J. B. : **Google Android:** crie aplicações para celulares e tablets. São Paulo: Casa do Código, 2012.