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

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*.

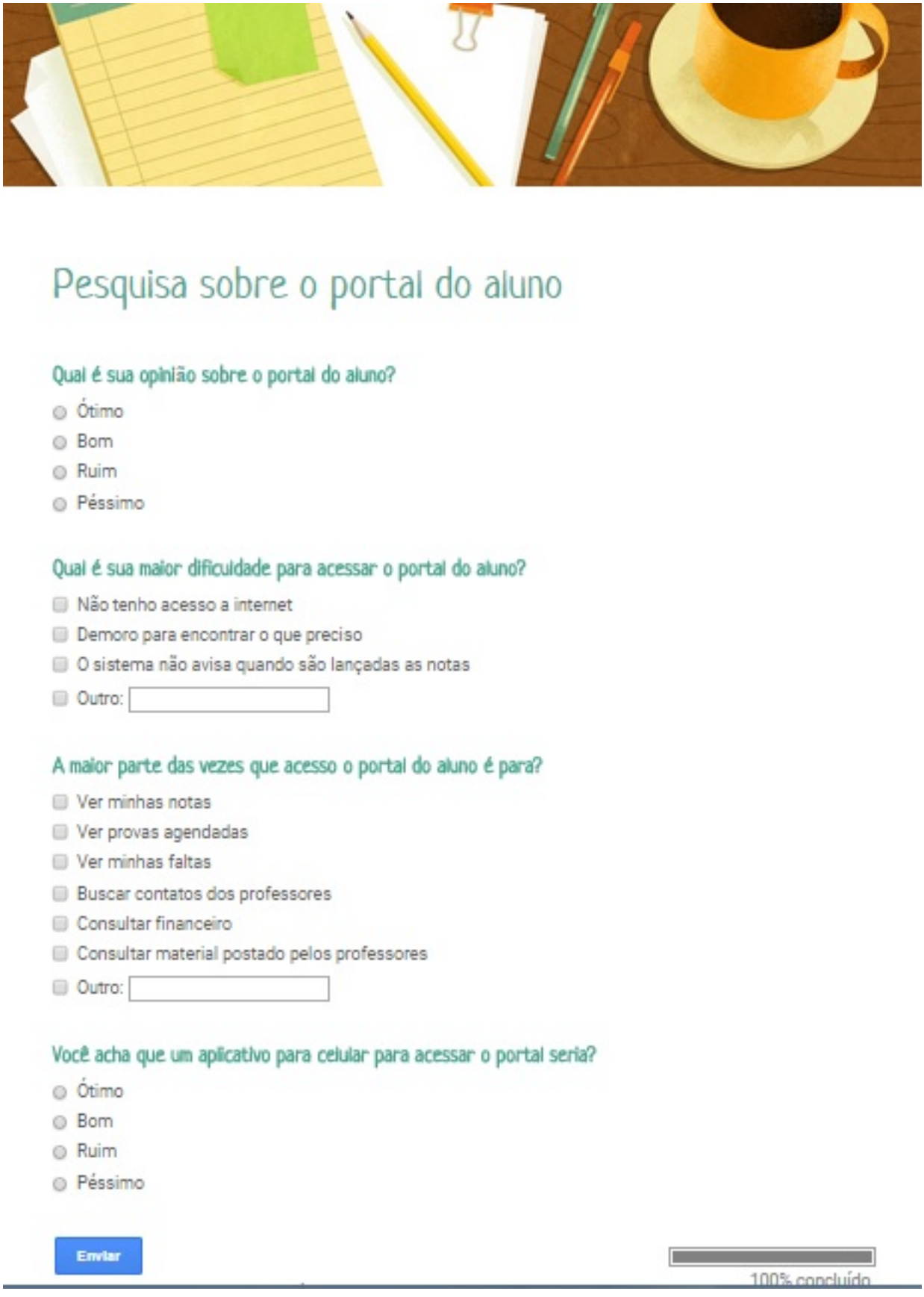

Figura 1 – Quetionário Aplicado. **Fonte:**Elaborado pelos autores.

1.4 Procedimentos e Resultados

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

1.4.1 Google Cloud Messaging(GCM)

No momento em que é lançada alguma nota, falta ou prova agendada, o *web service* precisa transmitir esta informação para o aplicativo Android. Para que esta comunicação aconteça foi utilizado o serviço da Google chamado de Google Cloud Messaging (GCM).

Neste contexto, o servidor *web* envia uma mensagem para o GCM com os elementos que precisa passar para a aplicação *mobile*. A partir daí, a entrega dos dados para os dispositivos móveis fica por conta da Google.

Para que o GCM apresentasse o resultado esperado, foi preciso acessar o *site* Google Developers Console através do endereço https://console.developers.google.com e construir um novo projeto. Para criá-lo, bastou clicar no botão ***Create Project*** que está na página inicial, conforme pode se ver na Figura 2. Logo após, foi adicionado um nome ao projeto e clicado no botão Criar, como mostra a Figura 3.

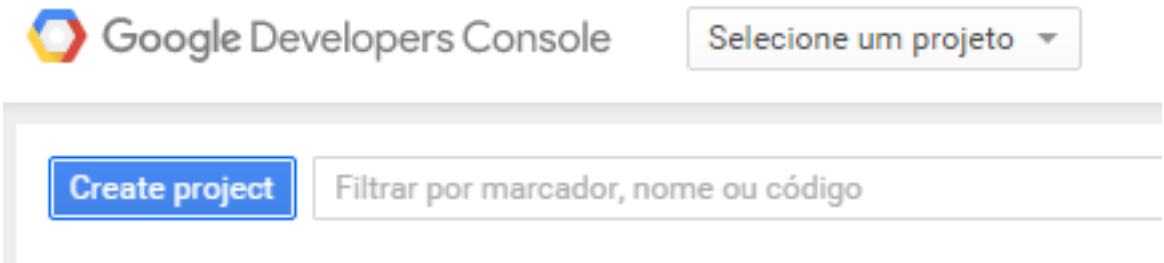

Figura 2 – Criando um novo projeto. **Fonte:**Elaborado pelos autores.

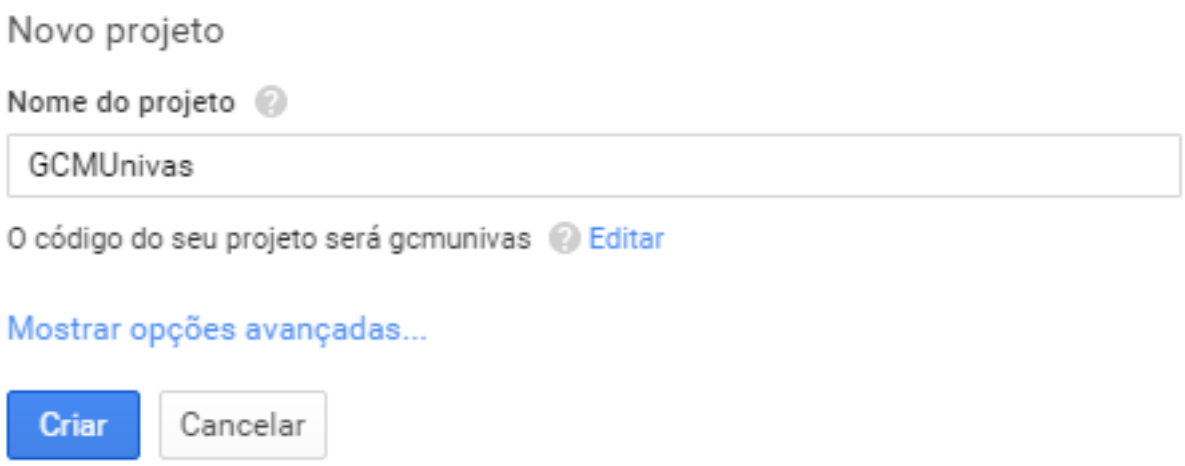

Figura 3 – Inserindo o nome do projeto. **Fonte:**Elaborado pelos autores.

Ao criar o projeto foi aberta uma tela para sua configuração, visível na Figura 4.

Figura 4 – Tela de configuração do projeto. **Fonte:**Elaborado pelos autores.

O primeiro dado que se obteve foi o número do projeto, também chamado de *Sender ID*. Este código serve para que a Google reconheça a aplicação que enviou a mensagem. Para visualizar este identificador, foi preciso clicar nos detalhes do projeto na página inicial, como se vê na Figura 5.

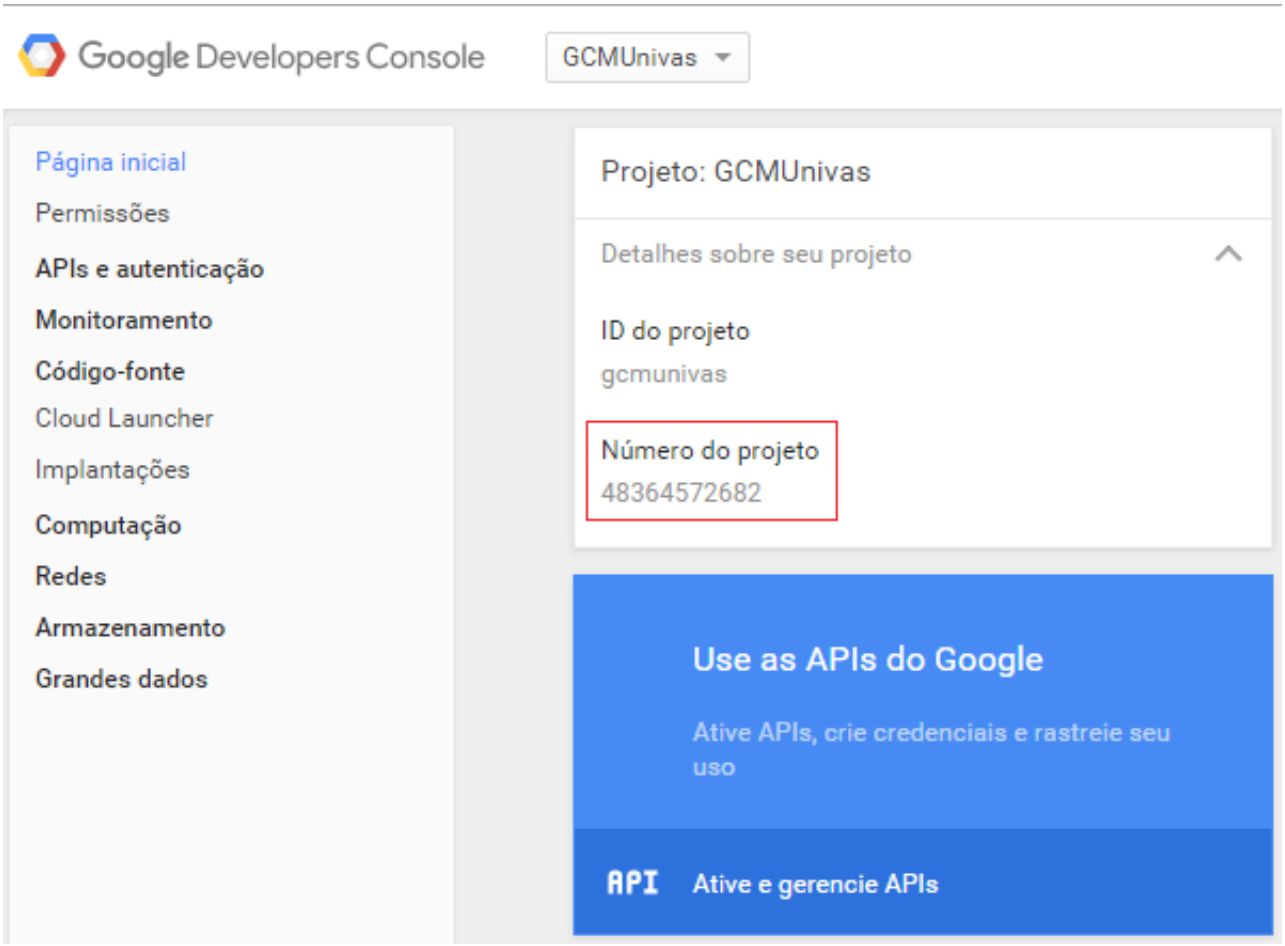

Figura 5 – Número do projeto. **Fonte:**Elaborado pelos autores.

O próximo passo, foi habilitar a API GCM para trabalhar com o projeto. Para essa etapa, foi necessário navegar até a aba API's e autenticação, selecionando a opção APIs, conforme indica a Figura 6. Na tela presente aparecem os serviços fornecidos pela Google. Neste caso optou-se por Cloud Messaging for Android, também ilustrado na Figura 6.

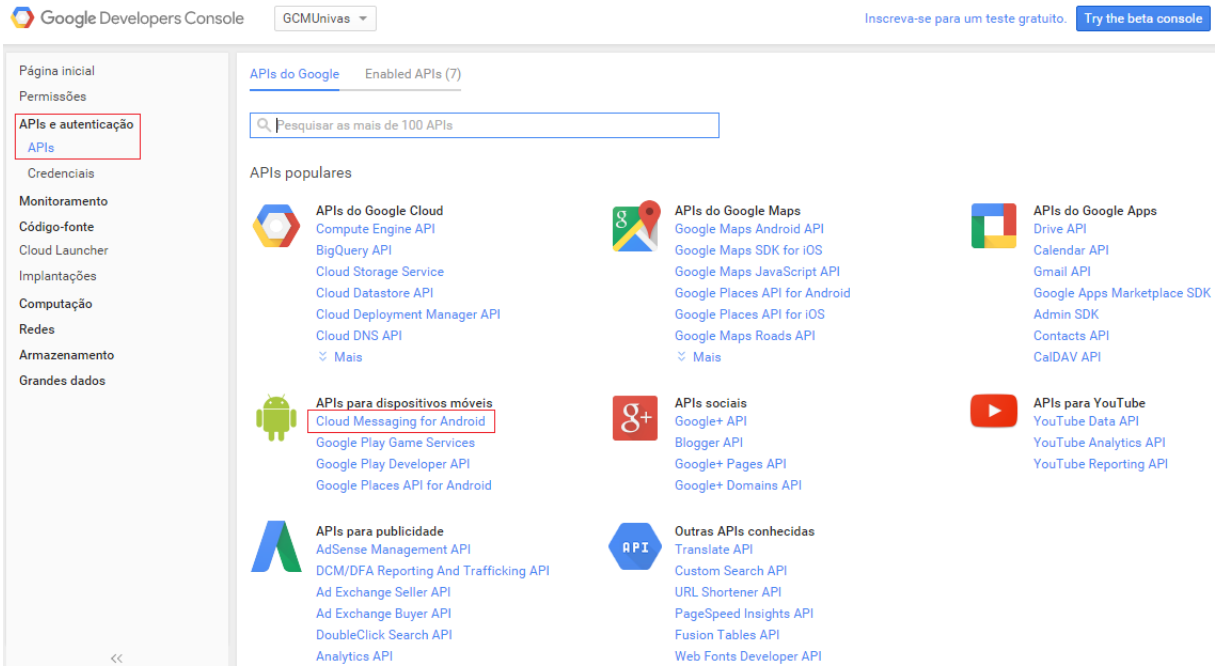

Figura 6 – Habilitando GCM para Android. **Fonte:**Elaborado pelos autores.

Ao selecionar Cloud Messaging for Android, foi apresentada a tela com a opção de ativar o GCM ao projeto, como é visível na Figura 7.

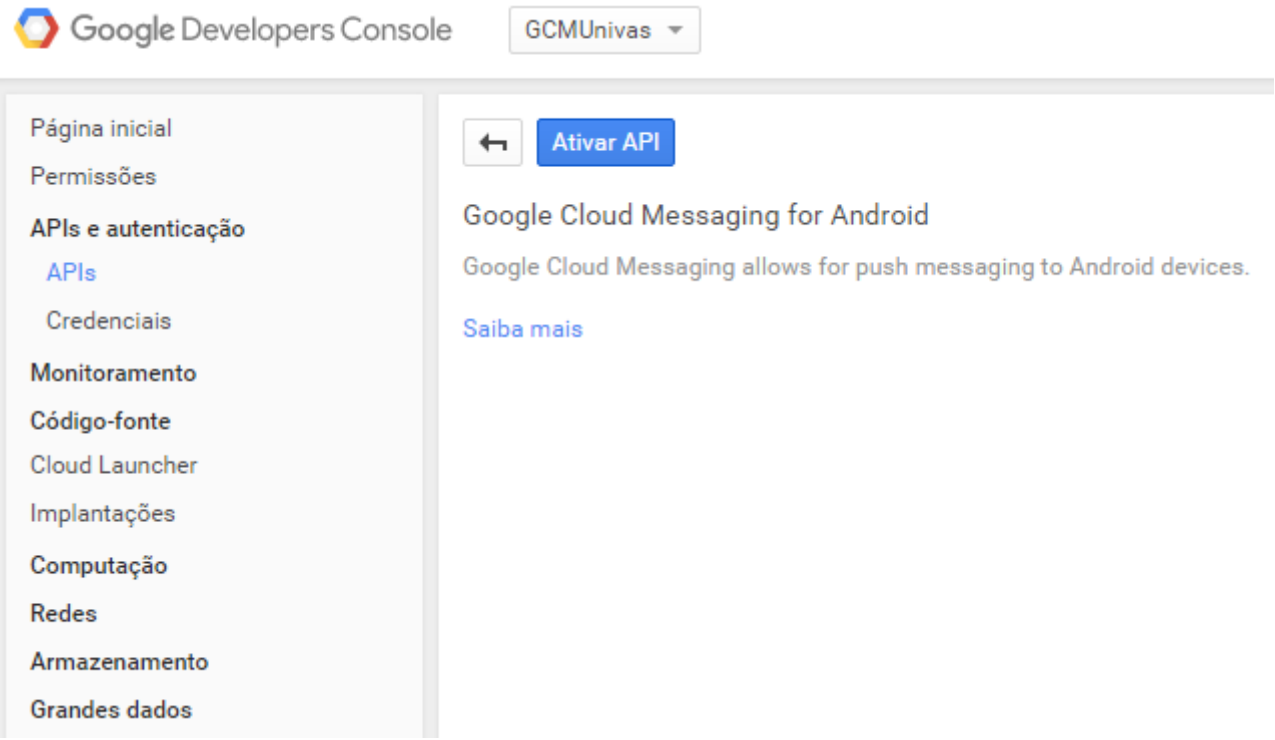

Figura 7 – Botão para ativar o GCM ao projeto. **Fonte:**Elaborado pelos autores.

Para concluir a configuração foi preciso acessar a aba APIs e autenticação, escolhendo a alternativa Credenciais, como mostra a Figura 8. Na página apresentada, foi selecionado a opção Chave de API, igualmente exibido na Figura 8.

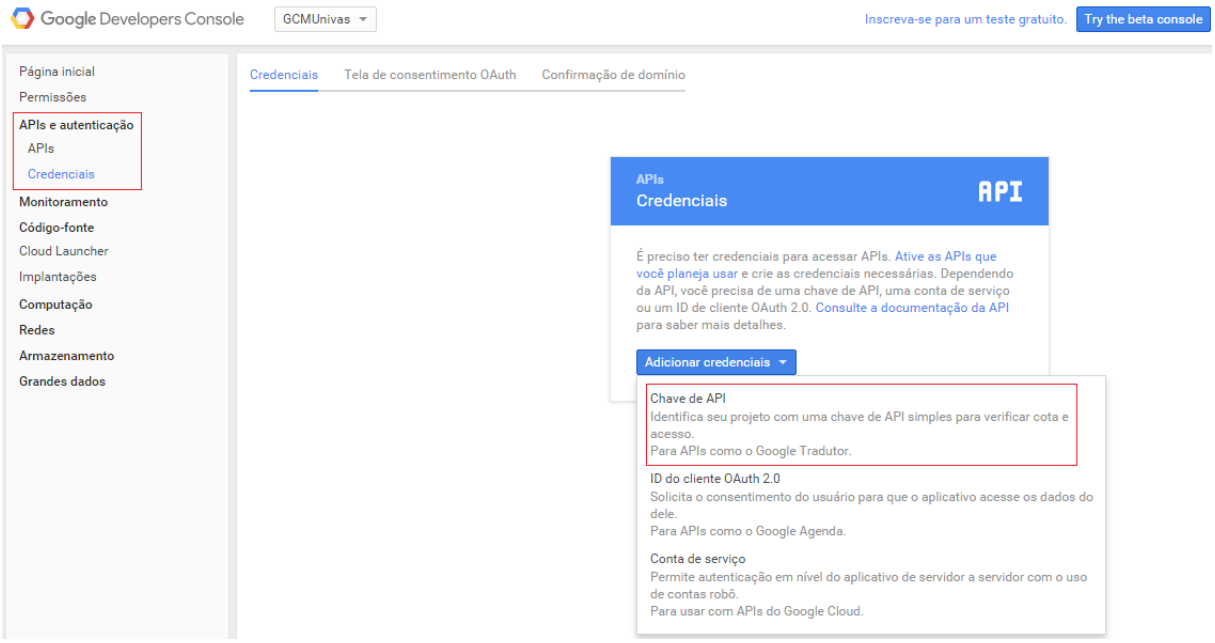

Figura 8 – Criando as credenciais. **Fonte:**Elaborado pelos autores.

Ao escolher Chave de API, foi exibida uma tela ao qual se escolheu a opção Chave de Servidor, demostrado na Figura 9.

Figura 9 – Escolhendo a opção Chave de Servidor. **Fonte:**Elaborado pelos autores.

Ao decidir-se por Chave de Servidor, foi apresentada uma tela onde é criada a chave pública, também conhecida por *Sender Auth Token*. Esta identificação é transmitida no cabeçalho das mensagens enviadas do servidor ao GCM. Para que esse código fosse gerado foi fundamental adicionar um nome e o IP do web service, como mostra a Figura 10. Ao clicar no botão Criar, a Google apresentou a chave gerada, como ilustra a Figura 11.

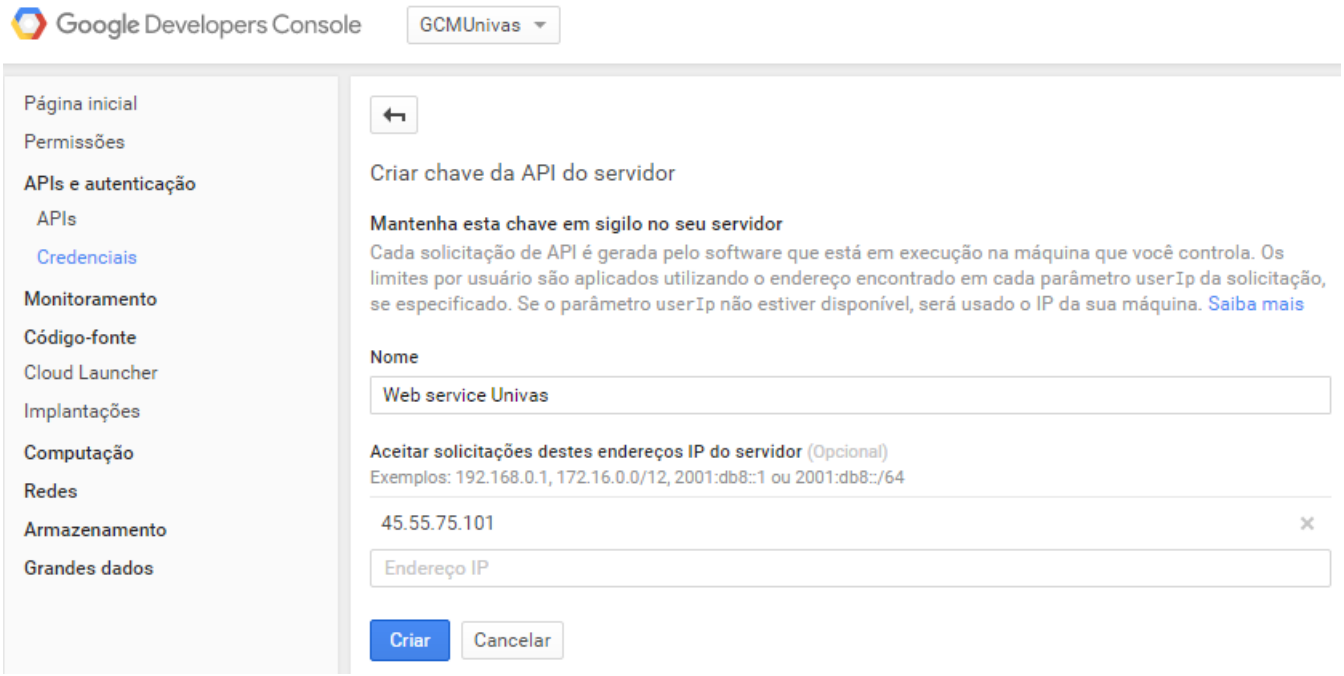

Figura 10 – Inserindo dados do servidor. **Fonte:**Elaborado pelos autores.

Figura 11 – Chave de API gerada. **Fonte:**Elaborado pelos autores.

1.4.2 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 através do endereço https://developer.android.com/intl/pt-br/sdk/index.html.

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), que permite a execução emulador Android mais rápido.

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, foi criado um repositório no controlador de versão Github, o qual pode ser acessado através do endereço https://github.com/diegodnunes12/AppTCC e compartilhado entre os participantes do projeto.

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.

O *layout* desta *activity* chama-se o `relativeLayout`, o qual permite adicionar um elemento em relação ao outro. Desta forma o *widget* `listView` utiliza o comando `android:layout_below="` para se posicionar após o componente com id *headerView* e a instrução `android:layout_above="@+idfo` indicando que ela deve preceder o *widget* com id *footerView*. Na Figura 12, pode ser visto o código XML dos *widgets* desta tela.

```xml
<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:background="@android:color/white"
    >

    "@drawable/logo1" />

    <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" />

</RelativeLayout>
```

Figura 12 – Código XML dos widgets do arquivo `fragment_navigation_drawer.xml`. **Fonte:**Elaborado pelos autores.

A classe `NavigationDrawerFragment` representa o painel de navegação. Nela se destaca o método `onCreateView()`, responsável por criar o *layout* de navegação. Na Figura 13,

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* da tela principal.

```
@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(new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> parent, View view, int position, long 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 13 – Método `onCreateView()`. **Fonte:**Elaborado pelos autores.

O próximo passo foi criar o banco de dados do aplicativo para salvar 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()` que cria a estrutura do banco de dados e outro conhecido por `onUpgrade()`, usado se for necessário atualizar a estrutura do banco de dados.

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 1 (um), o sistema chamará o método `onCreate()`. Se for preciso atualizar a estrutura do banco, o atributo versão deve ser incrementado em 1 (um), de modo que ao executar o software o sistema operacional perceba a mudança, chamando o método `onUpgrade()`. Na Figura 14 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 14 – Classe `DatabaseHelper`. **Fonte:**Elaborado pelos autores.

Em seguida foi criada a classe responsável por executar as consultas SQL, denominada `DatabaseExecute`. Nela foram inseridos os métodos responsáveis por inserir, alterar e buscar os dados dos alunos no banco de dados local do aplicativo. Na Figura 15, pode se observar o método que possibilita a inserção dos eventos ocorridos. Esses eventos podem ser notas, faltas ou provas agendadas.

```
public void insertEventos(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!");
    }else{
        Log.d(TAG, " Erro ao salvar o Evento!");
    }
}
```

Figura 15 – Método de inserção de eventos. **Fonte:**Elaborado pelos autores.

Este método recebe um objeto da classe `EventTO` com os elementos necessários para inserir o evento no banco de dados. 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()` que 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, que devolve um *Cursor* para navegar pelos resultados. Por fim, é composto um objeto do tipo `EventTO` e inserido na lista. Na Figura 16 é apresentado o método `getResults()`.

```
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 16 – Método `getResults()`. **Fonte:**Elaborado pelos autores.

Foram inseridos mais dois métodos semelhantes ao `getResults()`, chamados de `getFouls()` e `getAgendas()` para recuperar as faltas e provas agendadas respectivamente. O que diferencia-os é a consulta SQL, já que no `getFouls()` foram buscados os dados onde o tipo_evento = 'FALTAS' e no `getAgendas()` onde o tipo_evento = 'PROVA_AGENDADA'.

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

Logo após, criou-se uma classe chamada de `HttpUtil` para ler informações recebidas do *web service*. Nela foram inseridos dois métodos, sendo um chamado `getJsonDisciplinas()` para receber as informações referentes as disciplinas cursadas e outro denominado `getJsonEventos()` para captar os dados de eventos como notas, faltas e provas agendadas.

Os dois métodos são semelhantes, no entanto, o `getJsonEventos()` recebe os dados e transforma-os em objetos da classe `EventTO` enquanto o método `getJsonDisciplinas()` converte os elementos em objetos da classe `DisciplineTO`. Na Figura 17 é possível ver o método `getJsonEventos()` incumbido de ler as informações de eventos.

```
public void getJsonEventos(final String url){
    new Thread(new Runnable() {
        @Override
        public void run() {

            AlunoEventos 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, AlunoEventos.class);

                for (int i = 0; i< retorno.getEventos().size(); i++){
                    DatabaseExecute execute = new DatabaseExecute(helper);
                    EventTO to = new EventTO();
                    to.set_id((long) retorno.getEventos().get(i).getId_evento());
                    to.setValor_evento(retorno.getEventos().get(i).getValor());
                    to.setDescricao_evento(retorno.getEventos().get(i).getDescricao());
                    to.setId_disciplina(retorno.getEventos().get(i).getId_disciplina());
                    to.setData_evento(retorno.getEventos().get(i).getData());
                    to.setNota(retorno.getEventos().get(i).getNota());
                    to.setTipo_evento(retorno.getEventos().get(i).getTipoEvento());

                    execute.insertEventos(to);
                    Log.d("exec", "Eventos");
                }
                content.close();

            } catch (URISyntaxException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }).start();
}
```

Figura 17 – Método `getJsonEventos()`. **Fonte:**Elaborado pelos autores.

Neste método 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 utilizada a biblioteca Gson para convertê-las para o formato da classe `EventTO`. Após a leitura, o objeto da classe `EventTO` é enviado para a classe `DatabaseExecute`, a fim de realizar a inserção os dados no banco.

Para usufruir da biblioteca Gson, foi fundamental adicioná-la como uma dependência do projeto. Para isso, foi preciso ir ao Menu do Android Studio, clicando em **File** e depois em **Project Structure**. Com a janela da estrutura do projeto aberta, foi selecionada a aba **Dependencies** e depois foi escolhido o ícone de mais (+) para adicionar novas dependências, conforme mostra a Figura 18.

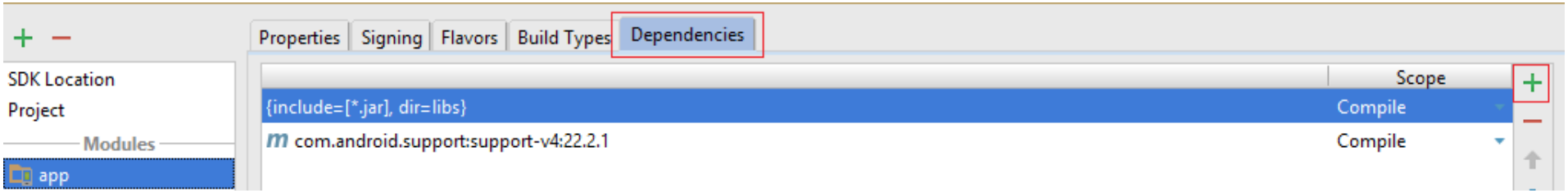

Figura 18 – Adicionando uma dependência ao projeto. **Fonte:**Elaborado pelos autores.

Na tela em que foi aberta localizou-se a biblioteca Gson com o endereço da Google, logo após selecionou-a e clicou no botão Ok para adicioná-la, como mostra a Figura 19.

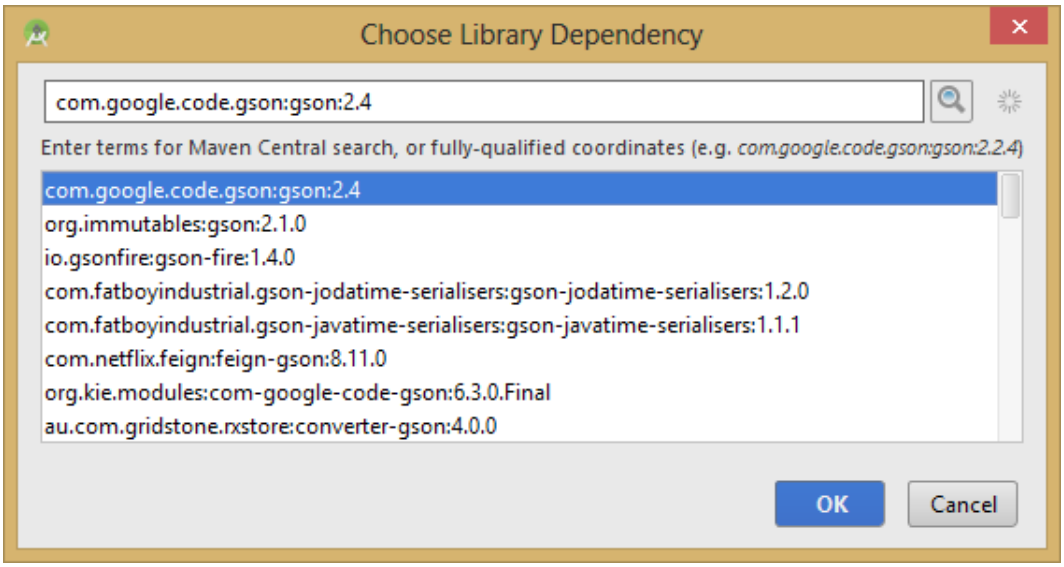

Figura 19 – Adicionando a biblioteca Gson ao projeto. **Fonte:**Elaborado pelos autores.

1.4.3 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.

1.4.3.1 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 do aplicativo, 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, de seu 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.

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

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.

1.4.3.2 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 20.

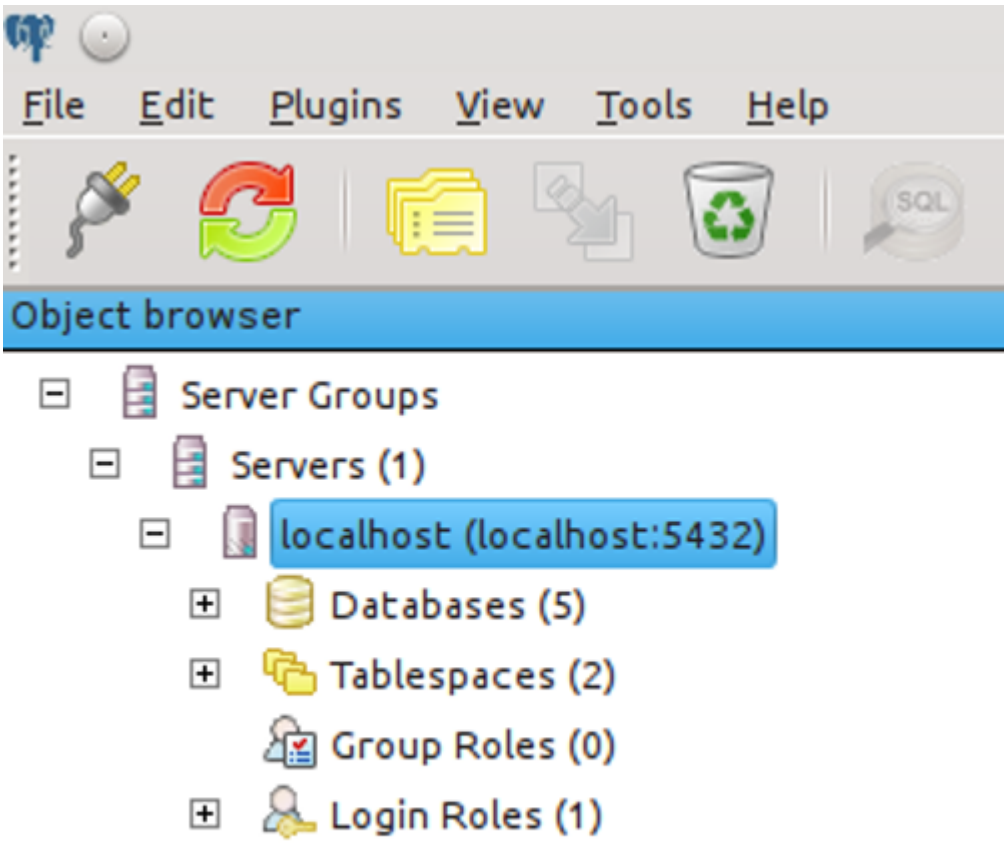

Figura 20 – Servidor de banco de dados local no PgAdmin. **Fonte:**Elaborado pelos autores.

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, apresentada na Figura 21.

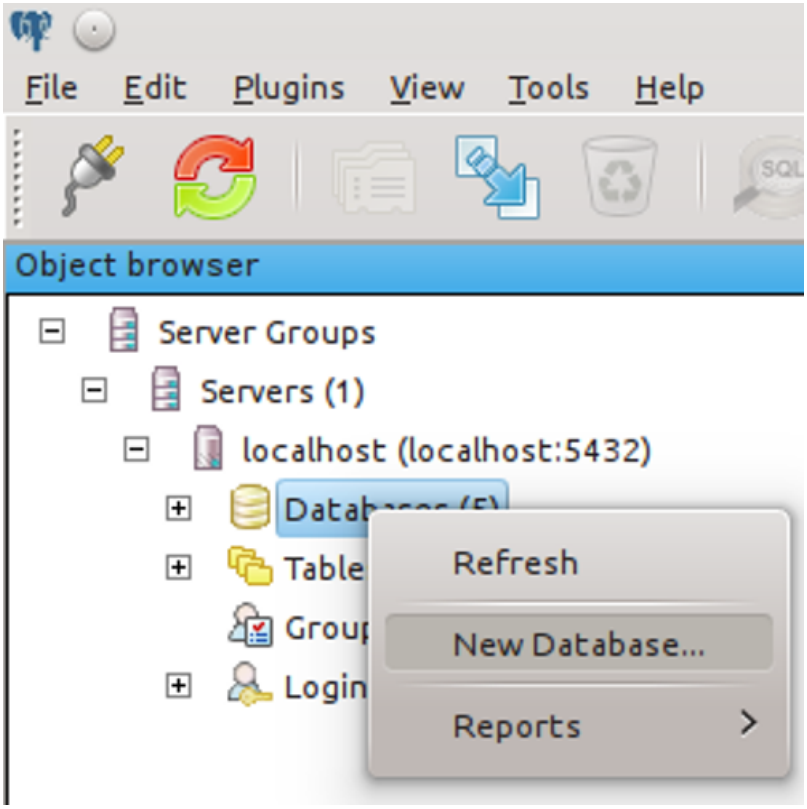

Figura 21 – Opção *New Database...*. **Fonte:**Elaborado pelos autores.

Em seguida foi necessário preencher o dados da janela apresentada, como está apresentado na Figura 22.

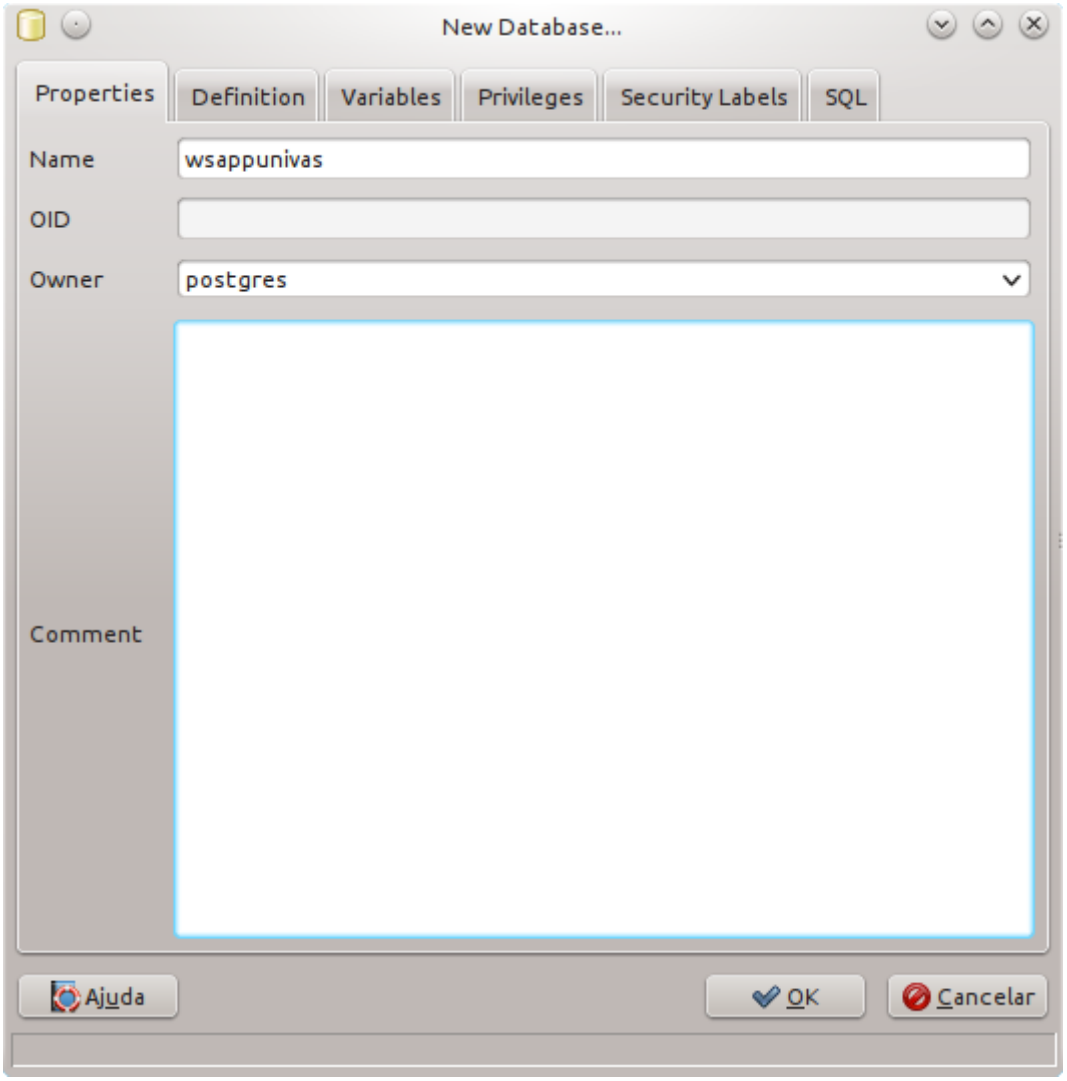

Figura 22 – Tela *New Database.* . . . **Fonte:**Elaborado pelos autores.

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 usuário, o responsável pelo banco de dados, que para este caso foi usuário padrão do SGDB, que é o `postgres`. Além destas configurações mais nenhuma foi necessária. O banco de dados 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 e de acordo com a evolução do projeto. 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. Para proceder com a criação de um novo projeto deste tipo no Eclipse, é necessário acessar na IDE, a opção **File -> New-> Dynamic Web Project** como pode ser visto na figura 23.

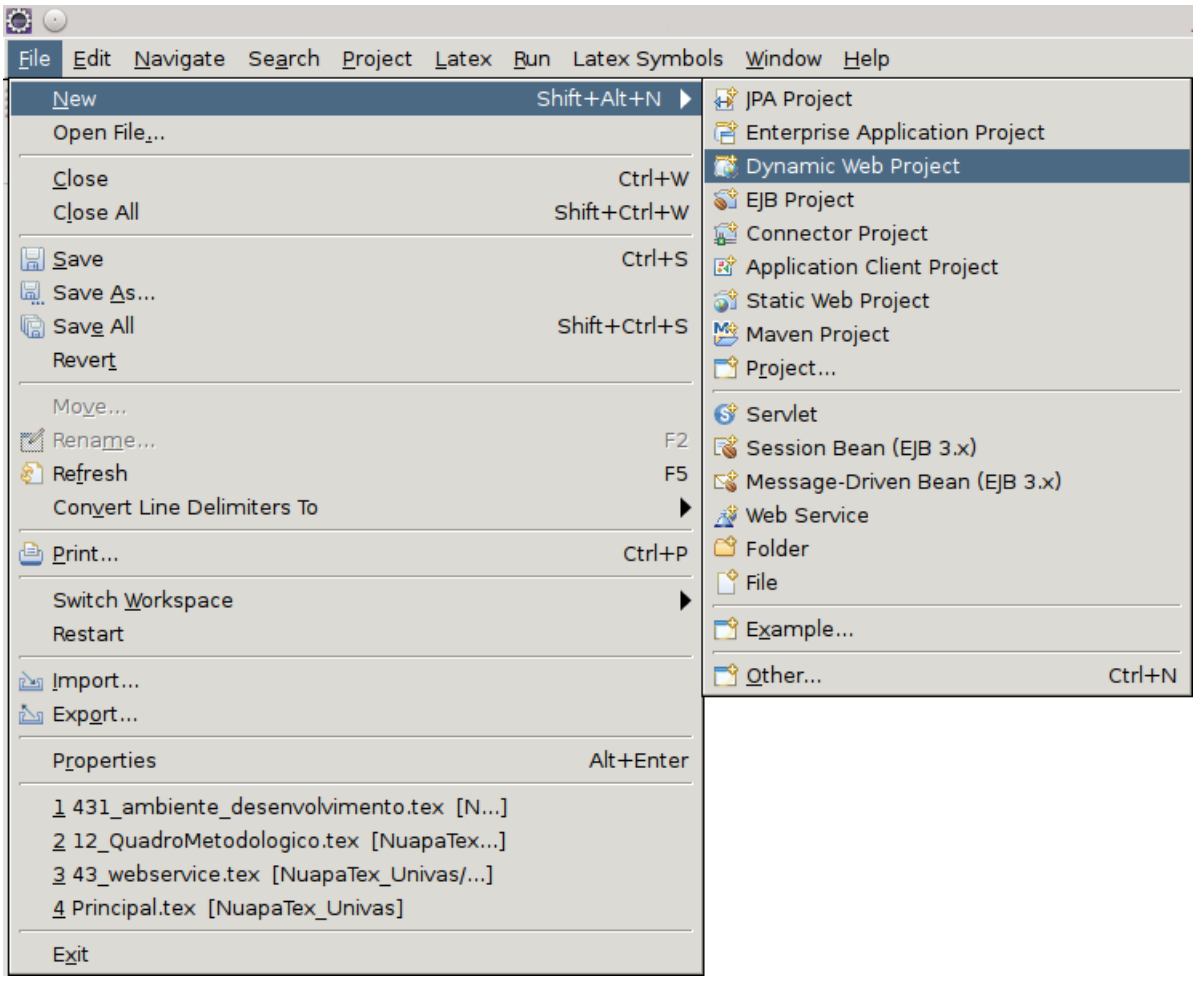

Figura 23 – Tela *New Database.* . . . **Fonte:**Elaborado pelos autores.

Em seguida foi apresentada uma tela para o preenchimento de alguns dados requeridos para a criação do projeto. Destas informações somente foi preenchido o nome do projeto. As outras informações continuaram sendo as que vem por padrão da IDE. A janela apresentada e as informações preenchidas podem ser vistas na Figura 24.

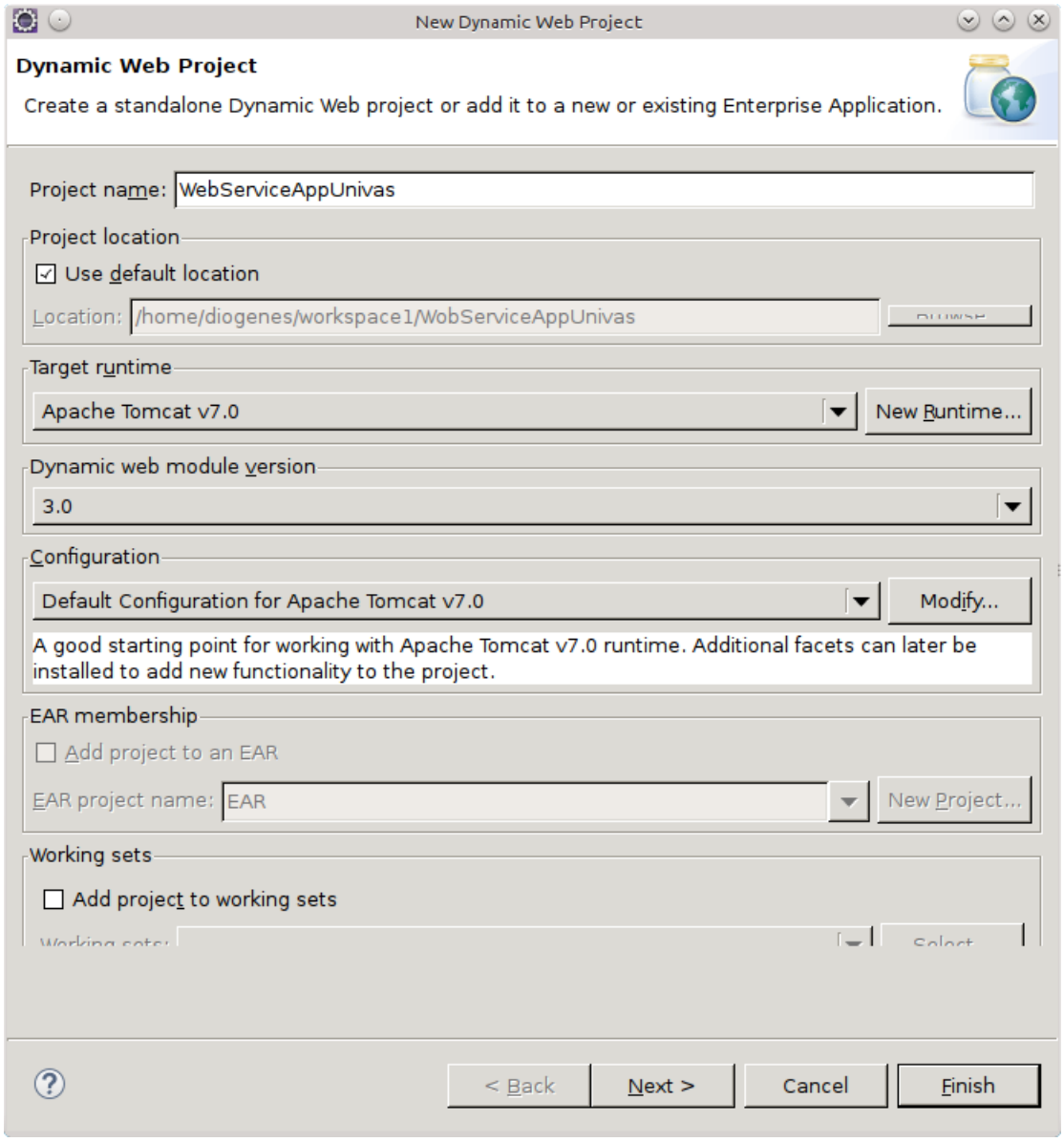

Figura 24 – Tela para criação de um novo projeto no Eclipse. **Fonte:**Elaborado pelos autores.

Na próxima janela apresentada, que têm por função configurar a pasta de códigos do projeto manteve-se a configuração apresentada pela IDE, como mostra a Figura 25.

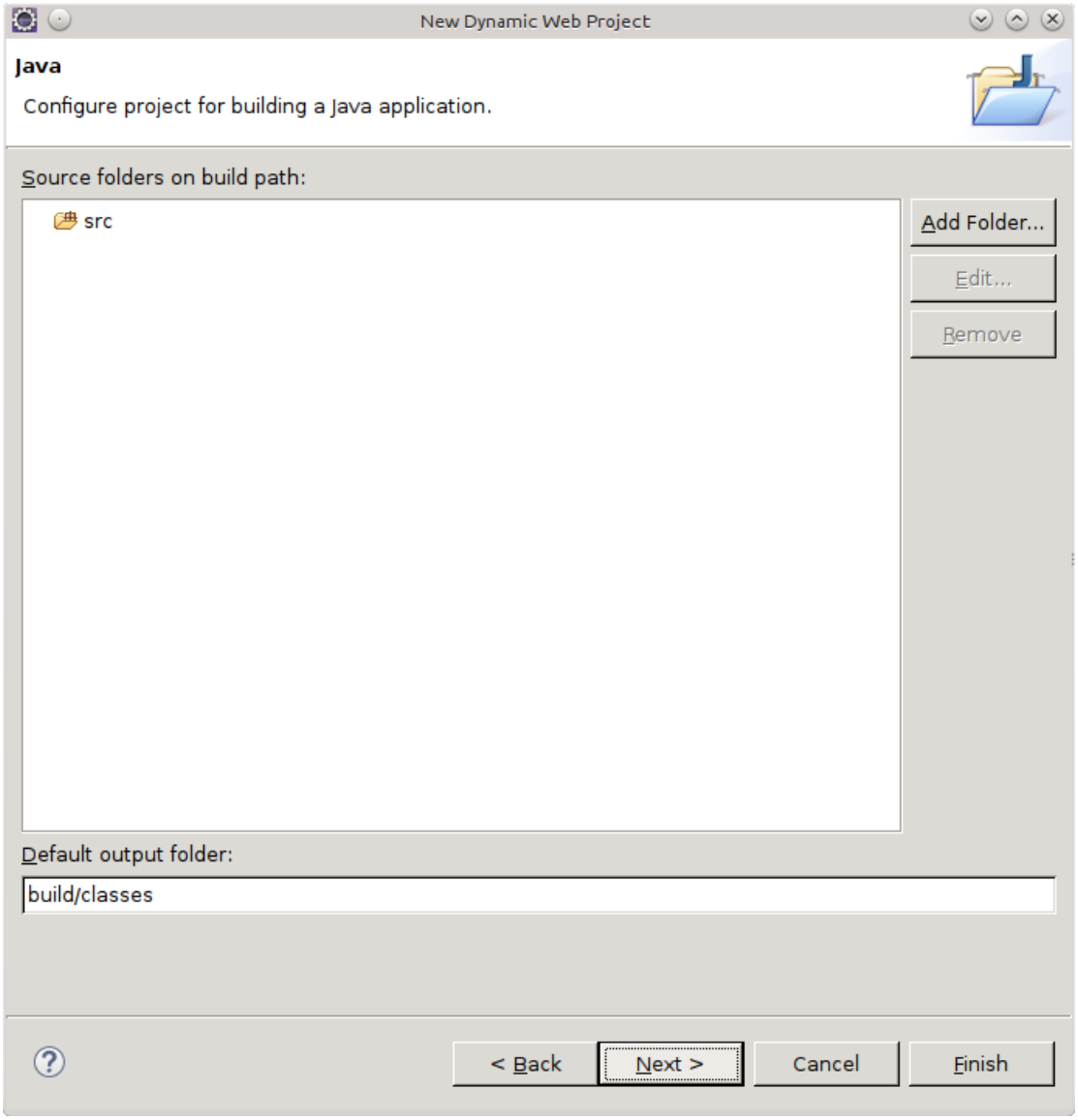

Figura 25 – Tela para criação de um novo projeto no Eclipse. **Fonte:**Elaborado pelos autores.

Na sequencia, na tela que foi apresentada era necessário preencher o campo **Context root:** com o contexto principal da aplicação web que acabou mantendo o próprio nome da aplicação. Além disso foi marcado a opção **Generate web.xml deployment descriptor**, para que ao criar o projeto, a própria IDE criasse o arquivo `web.xml`, arquivo responsável por algumas configurações da aplicação web. Esta tela esta apresentada na Figura 26.

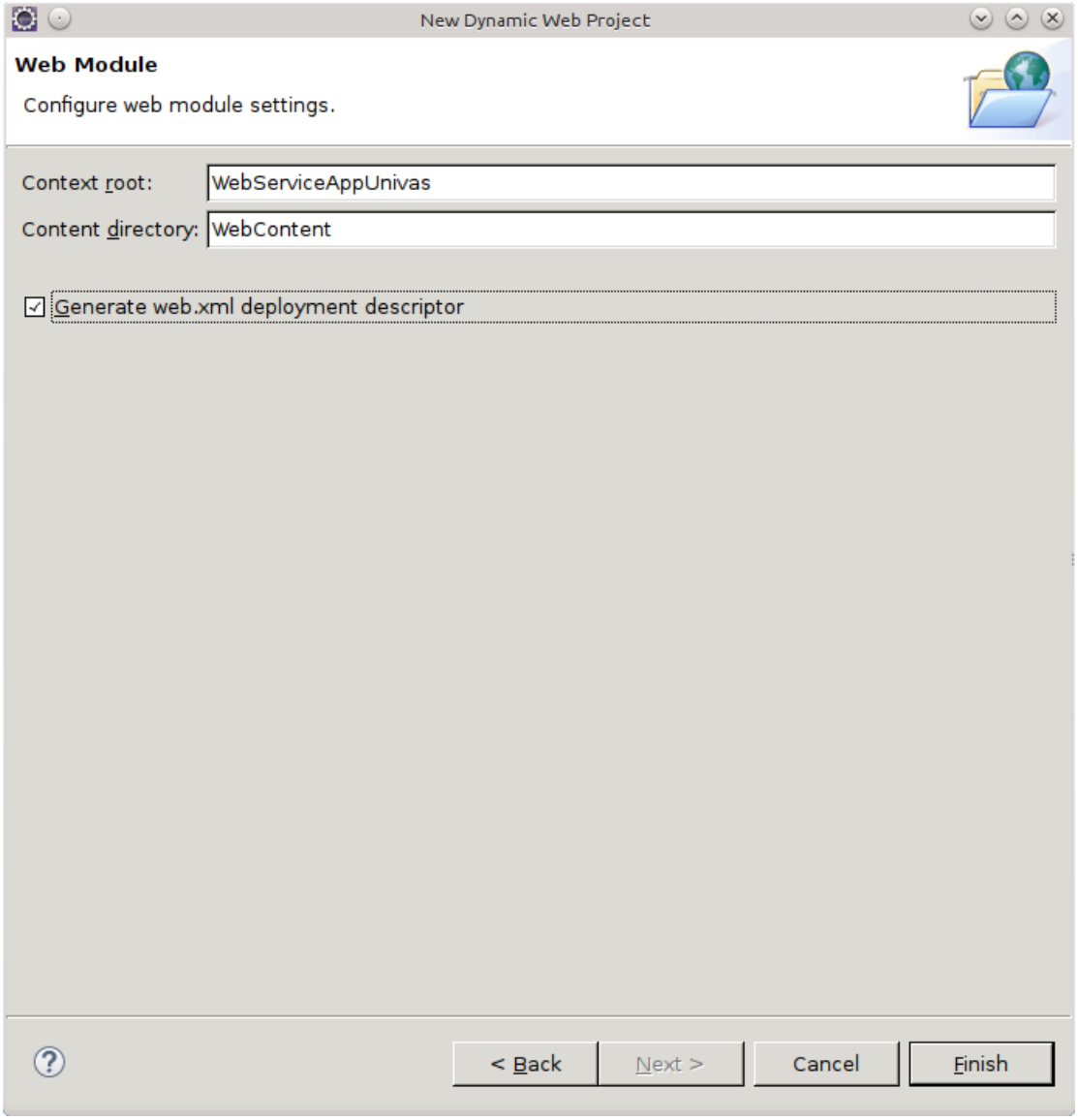

Figura 26 – Tela para criação de um novo projeto no Eclipse. **Fonte:**Elaborado pelos autores.

Após este passo foi concluído a criação do projeto, e já era 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. Para a criação do pacote foi necessário clicar com o botão direito do mouse sobre o projeto e acessar a opção **New -> Package**, como pode ser visto na Figura 27.

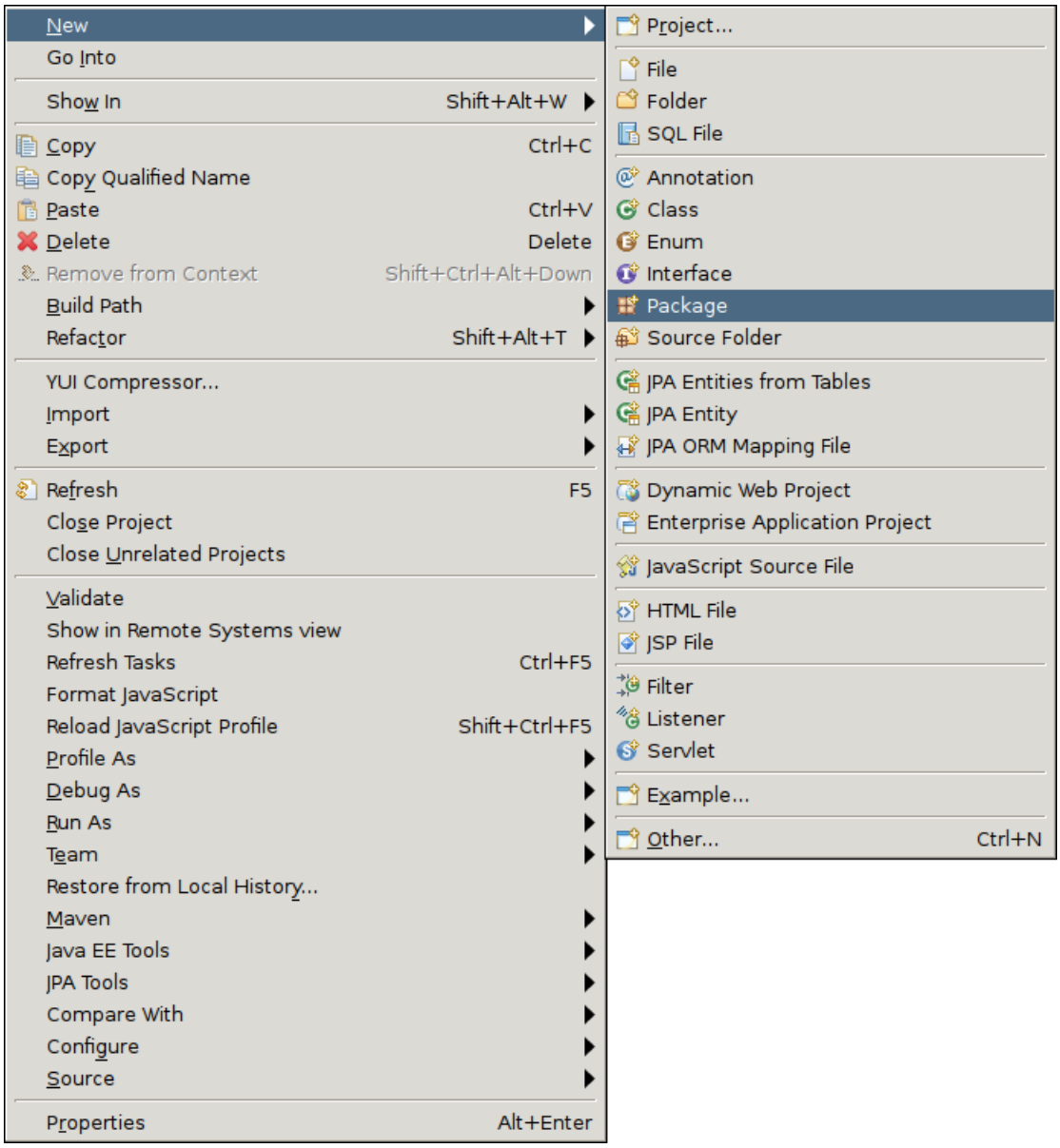

Figura 27 – Tela para criação de um novo projeto no Eclipse. **Fonte:**Elaborado pelos autores.

Em seguida foi apresentada a janela New Java Package, para a criação de um novo pacote mostrada na Figura 28. 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.

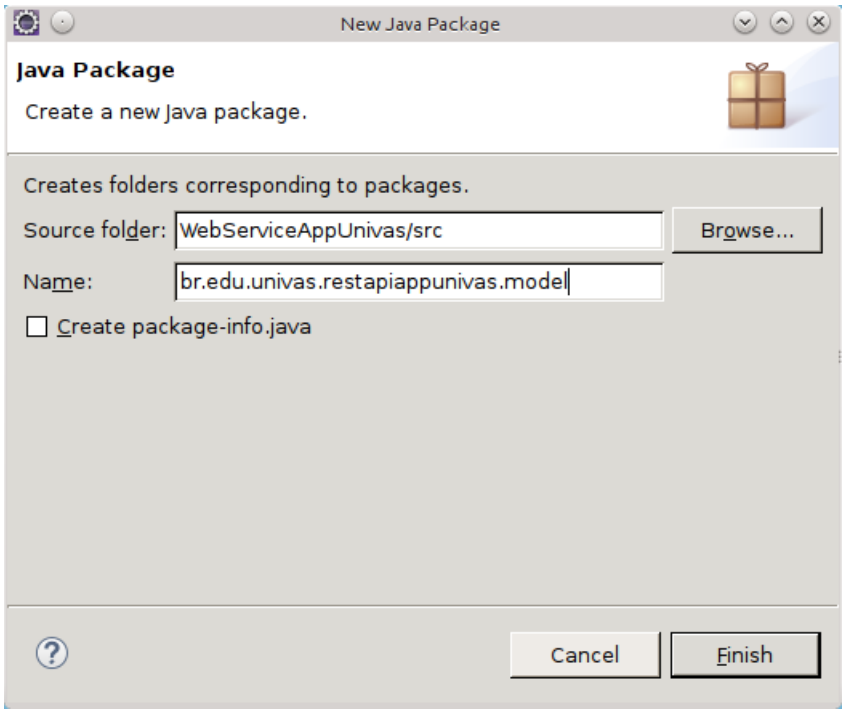

Figura 28 – Tela para criação de um novo projeto no Eclipse. **Fonte:**Elaborado pelos autores.

Com este pacote criado, ja era possível criar as classes do ORM. Foi criada primeiramente a classe `Student.java`. Para a criação desta classe foi necesário clicar com o botão direito do *mouse* sobre o projeto e navegar até a opção **New -> Class** como pode ser visto na Figura 29.

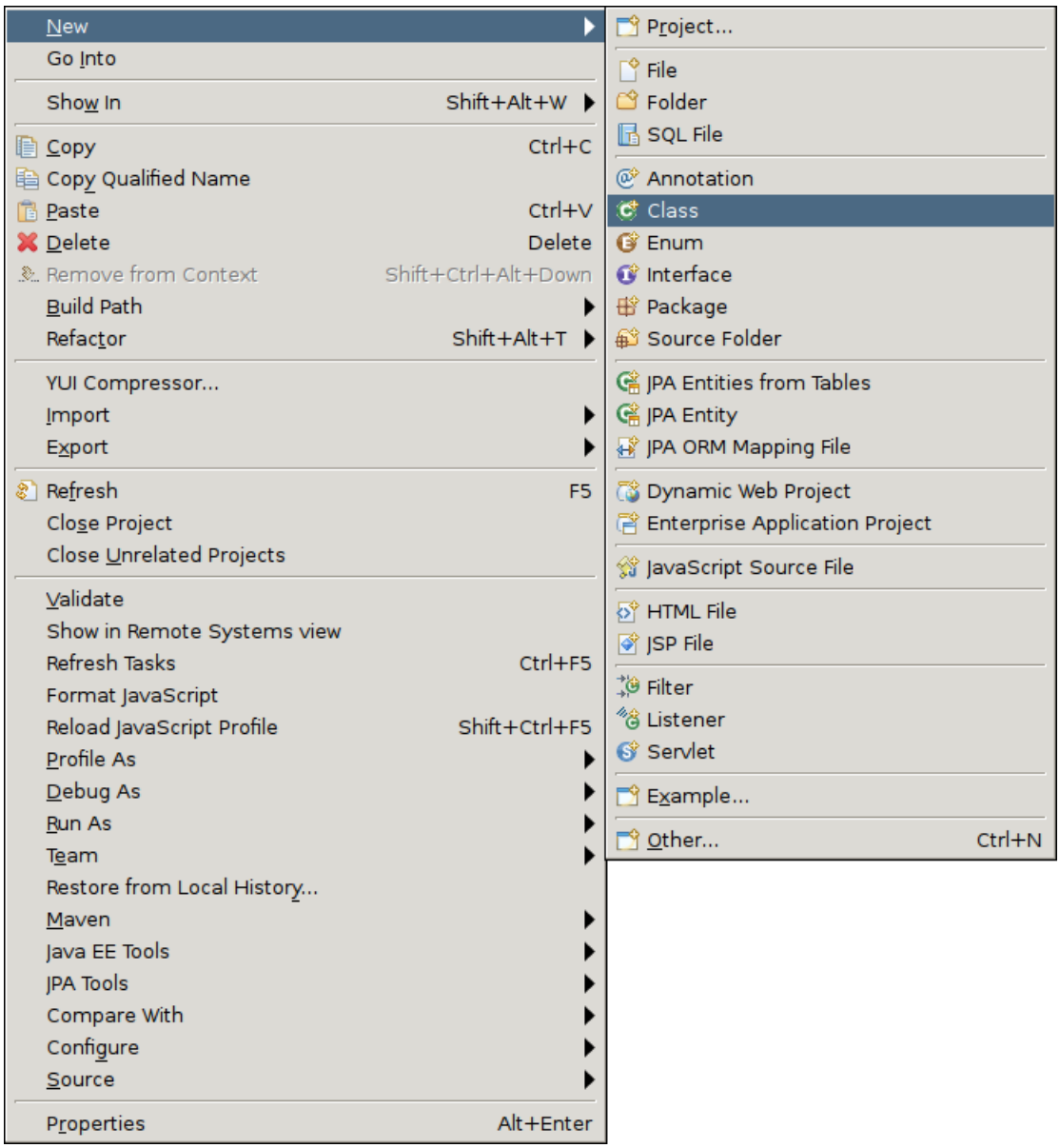

Figura 29 – Sem legenda. **Fonte:**Elaborado pelos autores.

Em seguida foi apresentada uma janela chamada New Java Class. Nesta janela somente foi necessário preencher o campo **Name:** que representa o nome da classe que está sendo criada.

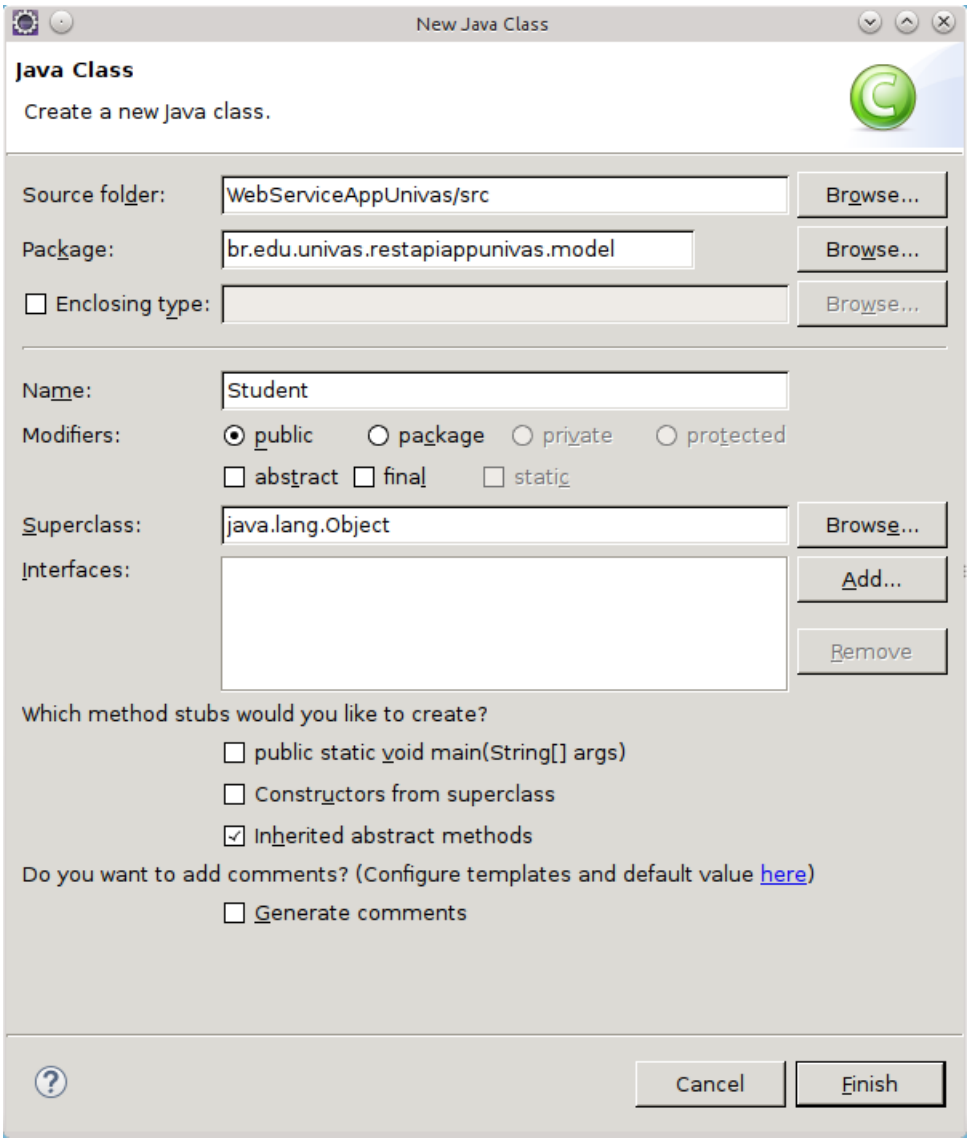

Figura 30 – Sem legenda. **Fonte:**Elaborado pelos autores.

Esta classe foi definida para representar as informações referente aos alunos. o código da classe pode ser visto na Figura 34.

```java
package br.edu.univas.restapiapp.model;
/**
 *imports omitidos
 */

@Entity
@Table(name = "student")
public class Student {

  @Id
  @SequenceGenerator(name = "id_student", sequenceName = "
     seq_id_student",
    allocationSize = 1)
  @GeneratedValue(generator = "id_student", strategy =
     GenerationType.IDENTITY)
  @Column(name = "id_student", nullable = false)
  private Long idStudent;

  @Column(name = "id_external", nullable = false)
  private Long idDatabaseExternal;

  @Column(length = 100, nullable = false)
  private String name;

  @Column(length = 100, nullable = false)
  private String email;

  @OneToMany(mappedBy="student", fetch = FetchType.EAGER)
  private List<Event> events;

  @OneToOne(optional = false, fetch = FetchType.LAZY)
  @JoinColumn(name = "id_user")
  private User user;

  /**
   * Omitidos todos Getters e Setters
   */

  @Override
  public int hashCode() {
    /**
     * Omitido
     */
  }

  @Override
  public boolean equals(Object obj) {
    /**
     * Omitido
     */
  }

}
```

Figura 31 – Sem legenda. **Fonte:**Elaborado pelos autores.

É válido lembrar esta classe possui anotações para que possa ser reconhecida como uma entidade do JPA, e assim persistida no banco de dados quando necessário. Além disso estas anotações possuem outras finalidades específicas. A seguir estão listadas todas as anotações que foram usadas na classe `Student.java` e nas outras classes que fazem parte do mapeamento objeto relacional da aplicação.

- `@Entity`: esta anotação foi necessária para que esta classe pudesse ser reconhecida como uma entidade do JPA e assim persistida no banco de dados;
- `@Table`: anotação que possui algumas configurações relativas a tabela no banco de dados, a qual esta entidade representa, no caso da classe' mostrada anteriormente é configurado o nome da tabela;
- `@Id`: esta anotação fica sobre o atributo que representa a chave primária no banco de dados;
- `@SequenceGenerator`: esta anotação define qual será o modo com que a chave primaria será incrementada.
- `@Column`: define algumas propriedades do campo da tabela do banco de dados, o qual o atributo que ele anota representa. Estas configuraçãoes podem são:
  - `name`: muda o nome do campo;
  - `length`: determina o tamanho em caracteres que o campo aceitará;
  - `nullable`: define se o preenchimento do campo é obrigatório;
  - `unique`: este atributo define se o campo aceitará valores únicos;
- `@OneToMany`: representa um relacionamento um-para-muitos no banco de dados. Anotam coleções de outras entidades;
- `@ManyToOne`: representa um relacionamento muitos-para-um no banco de dados. Este é a contraparte da anotação um-para-muitos;
- `@OneToOne`: representa um relacionamento um-para-um no banco de dados.

Esta classe faz parte do mecanismo de persistêcia de dados e é simplesmente um pojo ou seja, um objeto simples que contêm somente atributos privados e os métodos *getters* e *setters* que servem apenas para encapsular estes atributos. Além desta classe, foram criadas outras com

os mesmos propósitos. Estas classes tinham 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. As classes podem ser vistas na Figura 32.

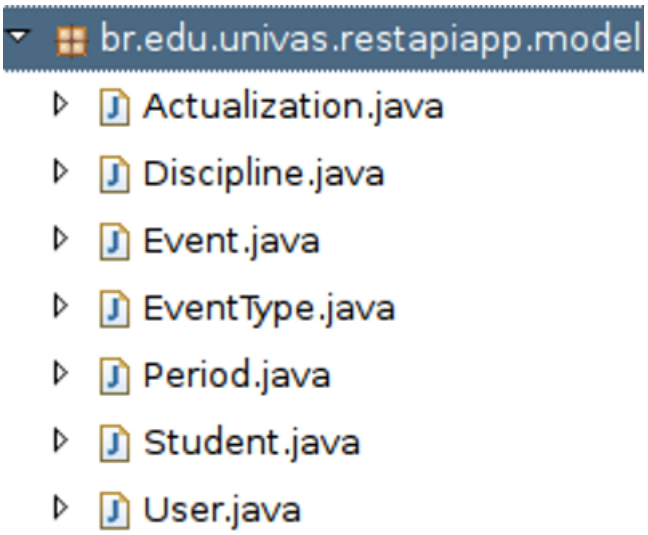

Figura 32 – Sem legenda. **Fonte:**Elaborado pelos autores.

E por fim, para cada classe que representa 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. A própria IDE provê uma forma facíl para criar este métodos, bastando para isso clicar com o botão direito do mouse sobre o código da classe e escolher a opção **Source -> Generate hashCode() and equals()...** como pode ser visto na Figura 33.

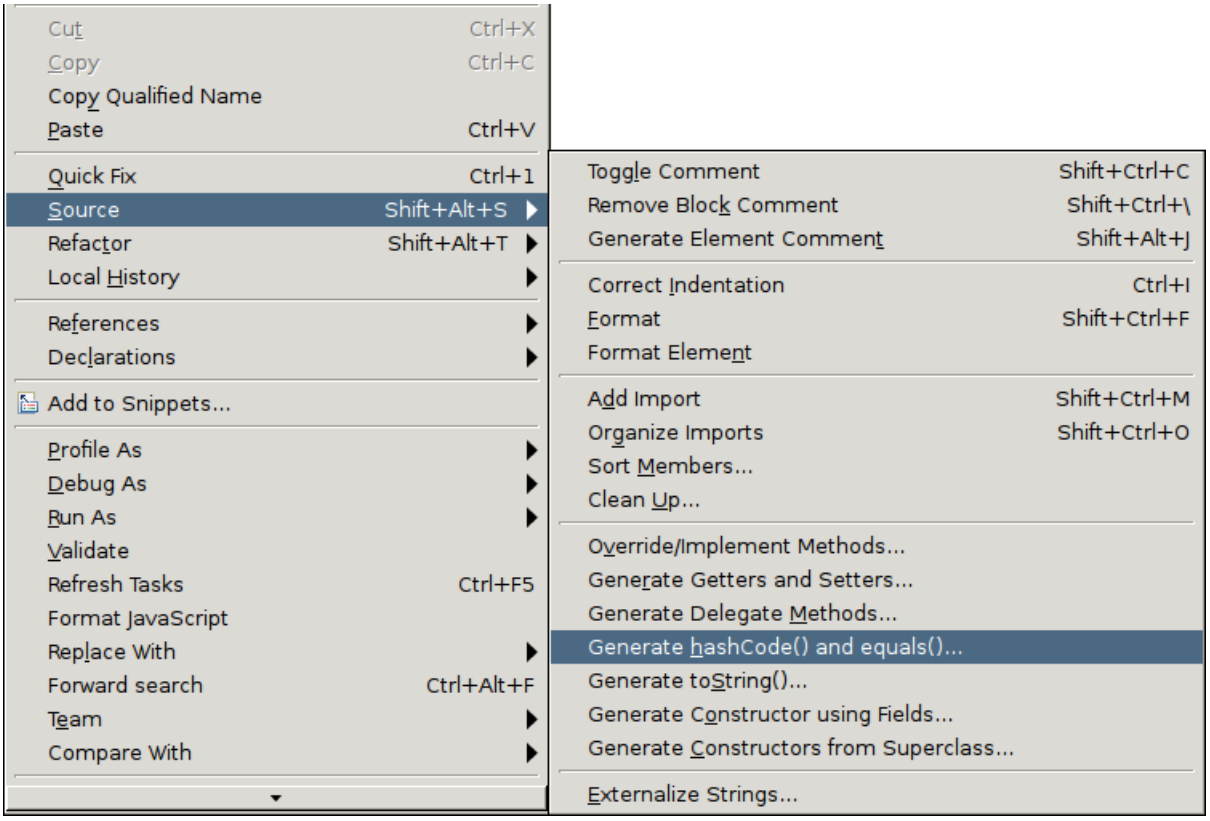

Figura 33 – Sem legenda. **Fonte:**Elaborado pelos autores.

Os métodos `hashCode` e `equals` da classe `Student.java` foram implementados usando o atributo id pode ser visto na imagem

```
1  ...
2
3  @Override
4  public int hashCode() {
5    final int prime = 31;
6    int result = 1;
7    result = prime * result
8        + ((idStudent == null) ? 0 : idStudent.hashCode());
9    return result;
10 }
11
12 @Override
13 public boolean equals(Object obj) {
14   if (this == obj)
15     return true;
16   if (obj == null)
17     return false;
18   if (getClass() != obj.getClass())
19     return false;
20   Student other = (Student) obj;
21   if (idStudent == null) {
22     if (other.idStudent != null)
23       return false;
24   } else if (!idStudent.equals(other.idStudent))
25     return false;
26   return true;
27 }
28 ...
```

Figura 34 – Sem legenda. **Fonte:**Elaborado pelos autores.

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 na Figura 35.

Em seguida à confecção do `persistence.xml` foi criada a classe `JpaUtil` que está representada na Figura 36. Esta classe é responsável por criar uma `EntityManagerFactory`. Este por sua vez é uma fábrica de instâncias de `EntityManager` 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.

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

Figura 35 – Arquivo `persistence.xml`. **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 .

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[3]. Não foi necessário fazer nenhuma implementação 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

[3] JSON - Javascript Object Notation

```java
package br.edu.univas.restapiappunivas.util;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

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 36 – Classe `JpaUtil.java`. **Fonte:**Elaborado pelos autores.

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.