

**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	Google Cloud Messaging(GCM)	5
1.4.2	Aplicativo	11
1.4.3	Web service	27
	REFERÊNCIAS	53
	APÊNDICES	55

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 agendadas.

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*.



Pesquisa sobre o portal do aluno

Qual é sua opinião sobre o portal do aluno?

- ☐ Ótimo
- ☐ Bom
- ☐ Ruim
- ☐ Péssimo

Qual é sua maior dificuldade para acessar o portal do aluno?

- ☐ Não tenho acesso a internet
- ☐ Demoro para encontrar o que preciso
- ☐ O sistema não avisa quando são lançadas as notas
- ☐ Outro:

A maior parte das vezes que acesso o portal do aluno é para?

- ☐ Ver minhas notas
- ☐ Ver provas agendadas
- ☐ Ver minhas faltas
- ☐ Buscar contatos dos professores
- ☐ Consultar financeiro
- ☐ Consultar material postado pelos professores
- ☐ Outro:

Você acha que um aplicativo para celular para acessar o portal seria?

- ☐ Ótimo
- ☐ Bom
- ☐ Ruim
- ☐ Péssimo

Enviar

100% concluído

Figura 1 – Questionário 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.

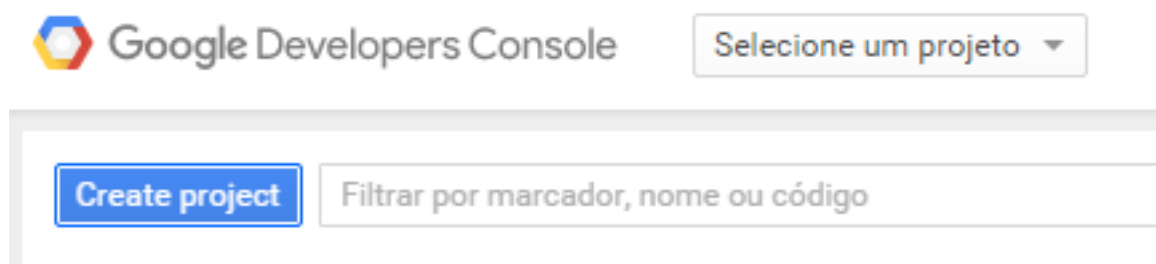


Figura 2 – Criando um novo projeto. **Fonte:**Elaborado pelos autores.

Novo projeto

Nome do projeto ?

O código do seu projeto será gcmunivas ? [Editar](#)

[Mostrar opções avançadas...](#)

Criar

Cancelar

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.

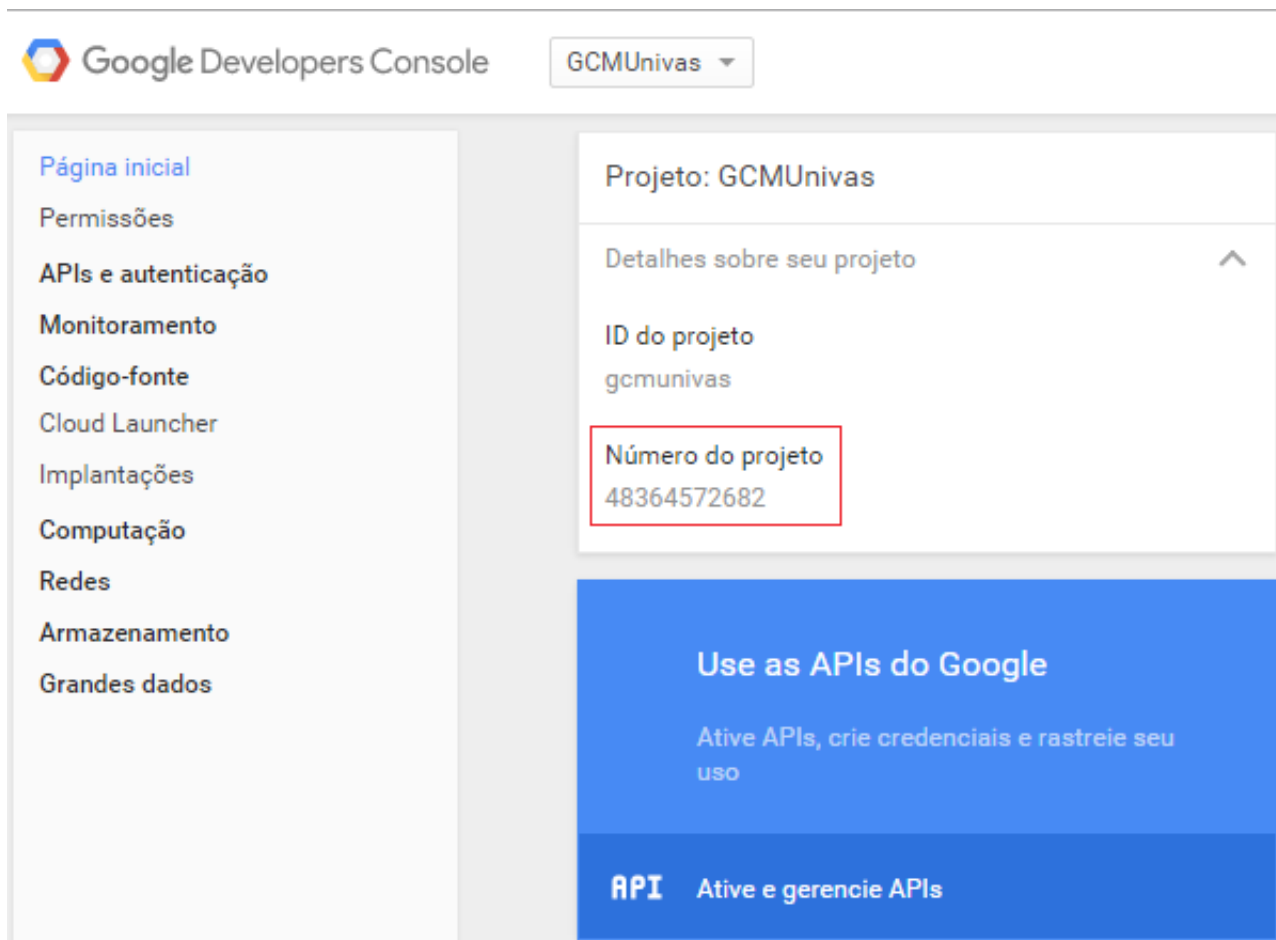


Figura 5 – Número do projeto. **Fonte:**Elaborado pelos autores.

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.

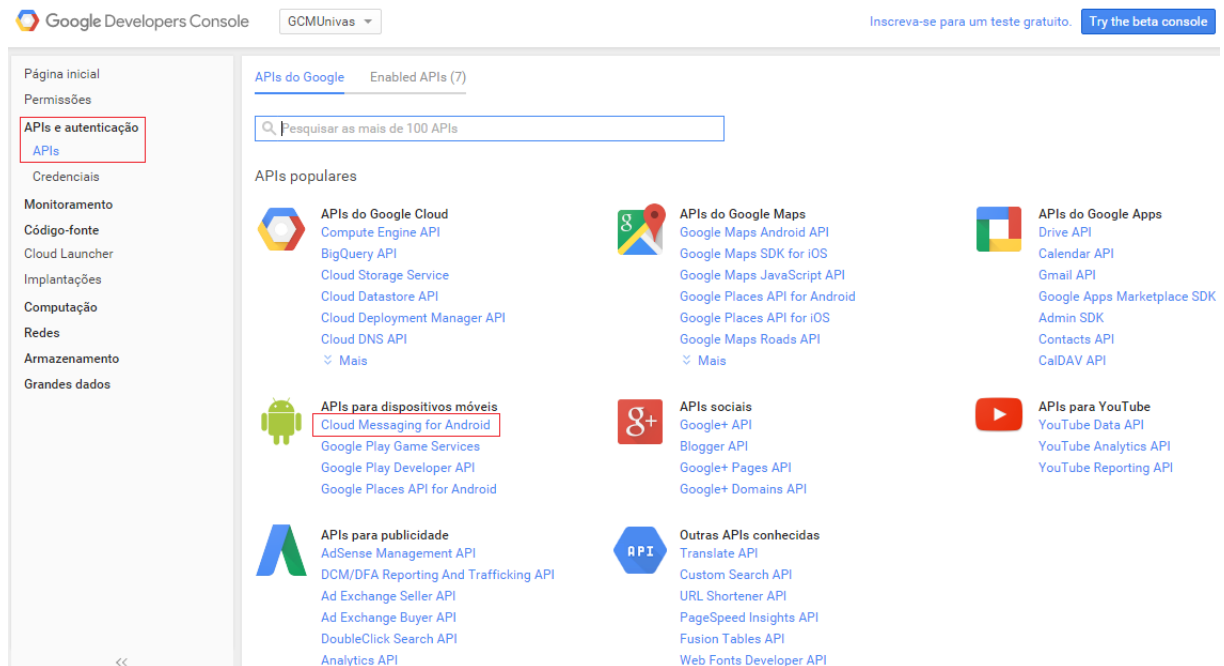


Figura 6 – Habilitando GCM para Android. **Fonte:**Elaborado pelos autores.

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.

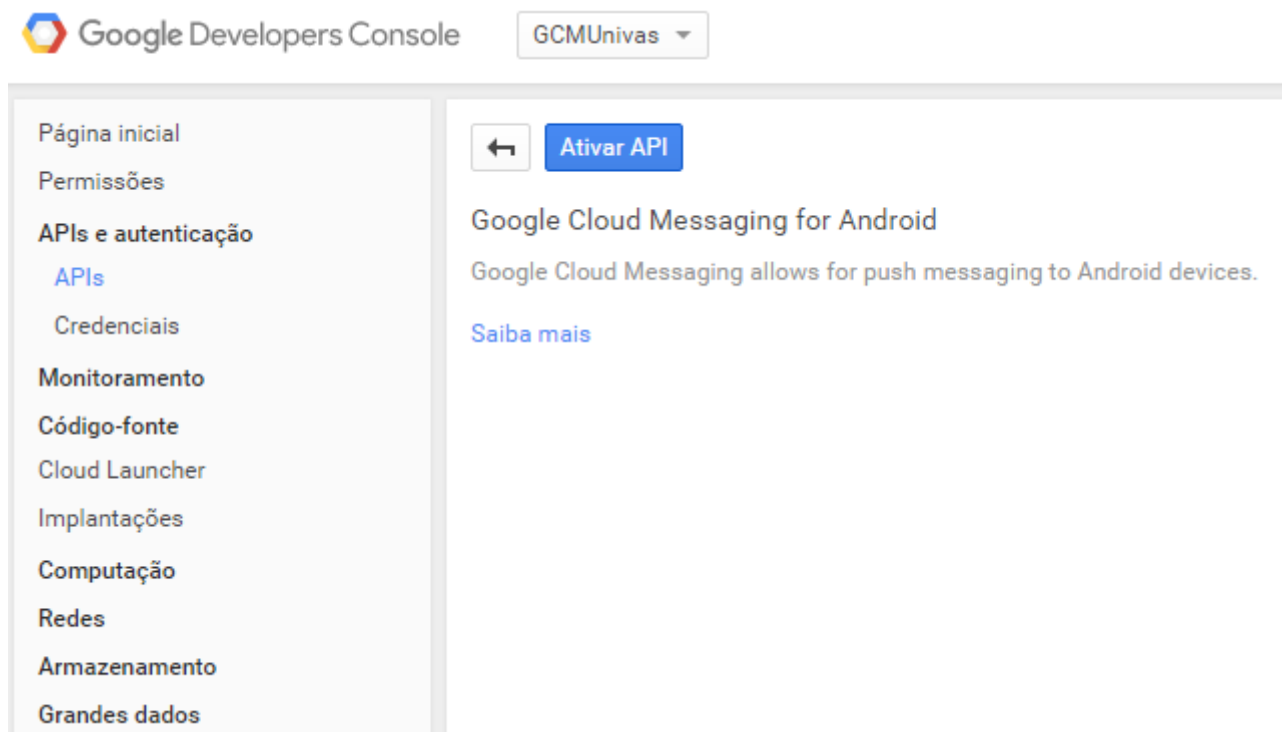


Figura 7 – Botão para ativar o GCM ao projeto. **Fonte:**Elaborado pelos autores.

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.

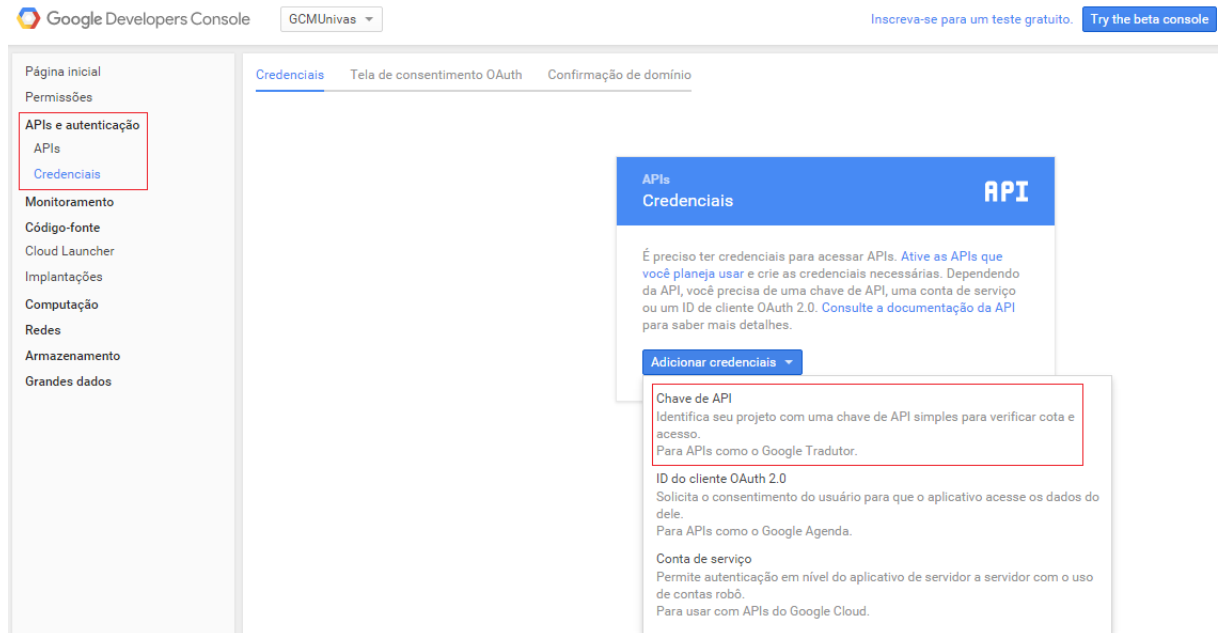


Figura 8 – Criando as credenciais. **Fonte:**Elaborado pelos autores.

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

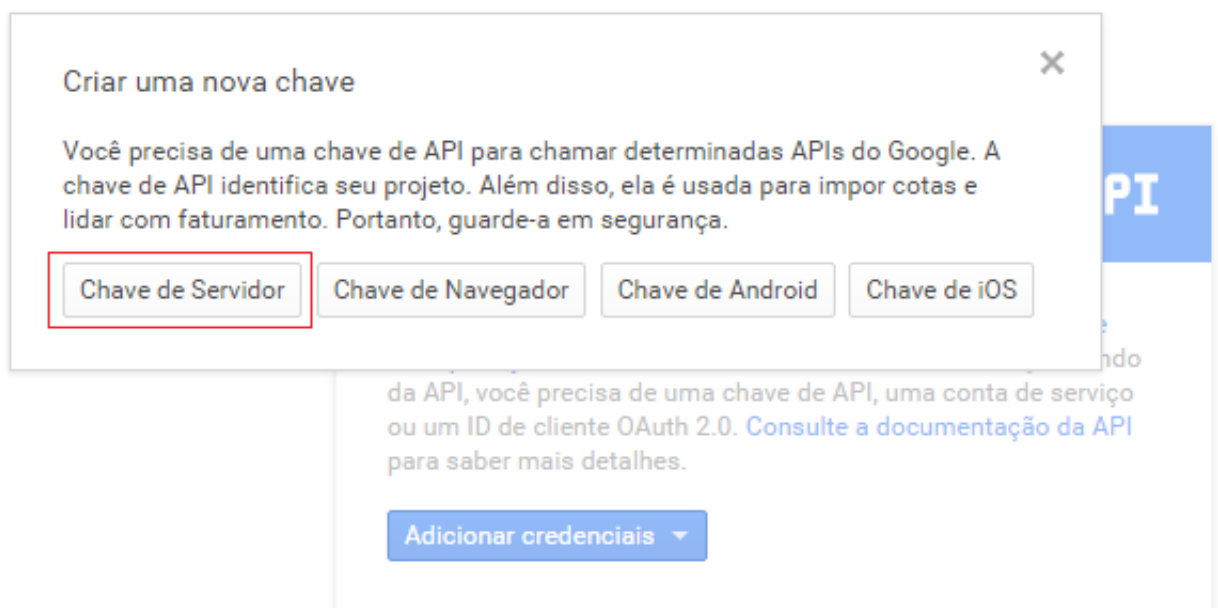
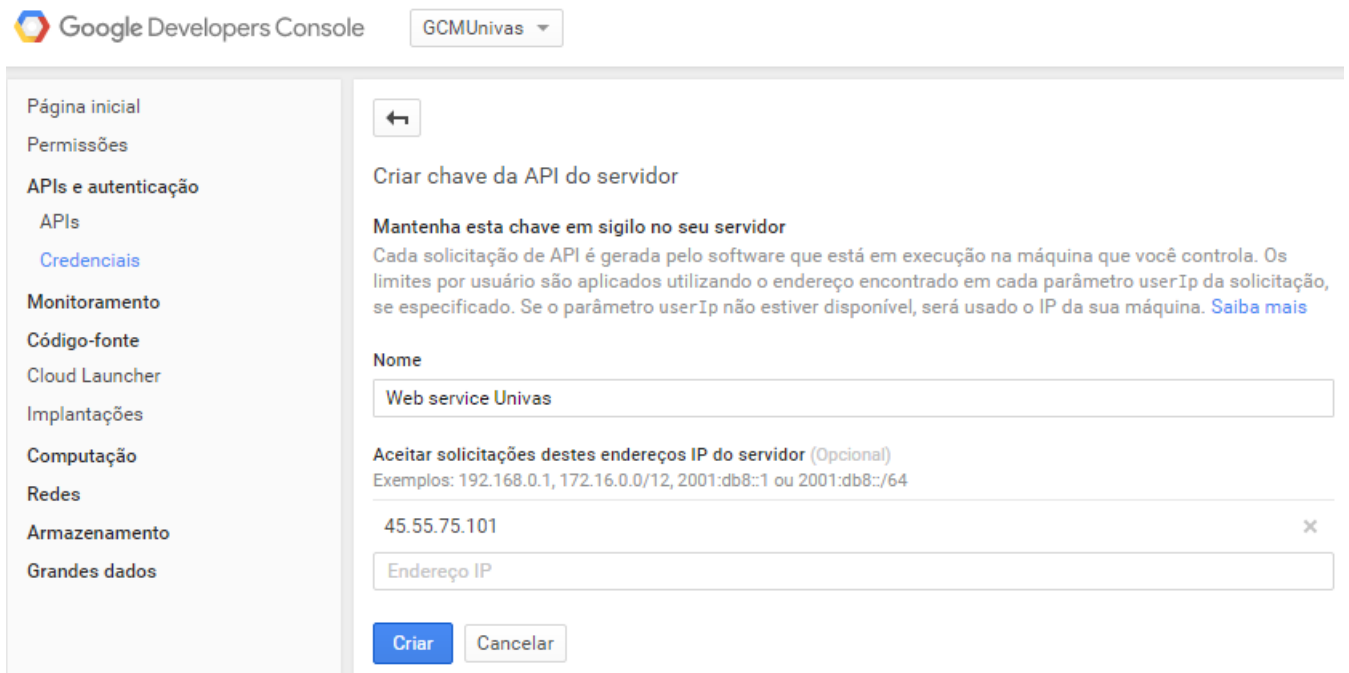


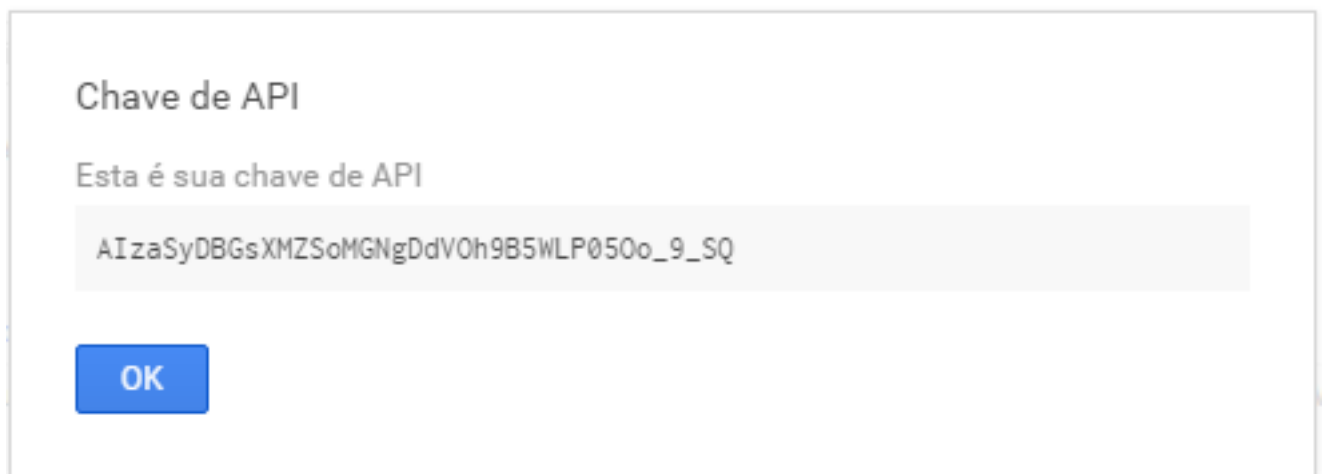
Figura 9 – Escolhendo a opção Chave de Servidor. **Fonte:**Elaborado pelos autores.

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.



The screenshot shows the 'Google Developers Console' interface. On the left is a sidebar with navigation links: 'Página inicial', 'Permissões', 'APIs e autenticação', 'APIs', 'Credenciais', 'Monitoramento', 'Código-fonte', 'Cloud Launcher', 'Implantações', 'Computação', 'Redes', 'Armazenamento', and 'Grandes dados'. The main content area is titled 'Criar chave da API do servidor' (Create server API key). It includes a warning: 'Mantenha esta chave em sigilo no seu servidor' (Keep this key secret on your server) and explains that API requests are generated by software on the machine you control. Below this, there is a 'Nome' (Name) field containing 'Web service Univas'. Then, there is a section 'Aceitar solicitações destes endereços IP do servidor (Opcional)' (Accept requests from these server IP addresses (Optional)) with examples of IP addresses and a field containing '45.55.75.101'. At the bottom are 'Criar' (Create) and 'Cancelar' (Cancel) buttons.

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



The screenshot shows a dialog box titled 'Chave de API' (API Key). It contains the text 'Esta é sua chave de API' (This is your API key). Below this, the generated key is displayed in a text box: 'AIzaSyDBGsXMZSoMGNgDdV0h9B5WLP050o_9_SQ'. At the bottom left is an 'OK' button.

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/App-TCC> 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 *NavigationDrawerFragment* 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="@+id/headerView* para se posicionar após o componente com id *headerView* e a instrução *android:layout_above="@+id/footerView"* indicando que ela deve preceder o *widget* com id *footerView*. Na Figura 12, pode ser visto o código XML dos *widgets* desta tela.

```

<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.getId());
    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:

Logo após, criou-se uma classe chamada de HttpUtil para ler informações recebidas do *web service*. Nela foram implementados dois métodos diferentes, 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 EventT0. Após a leitura, o objeto da classe EventT0 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.

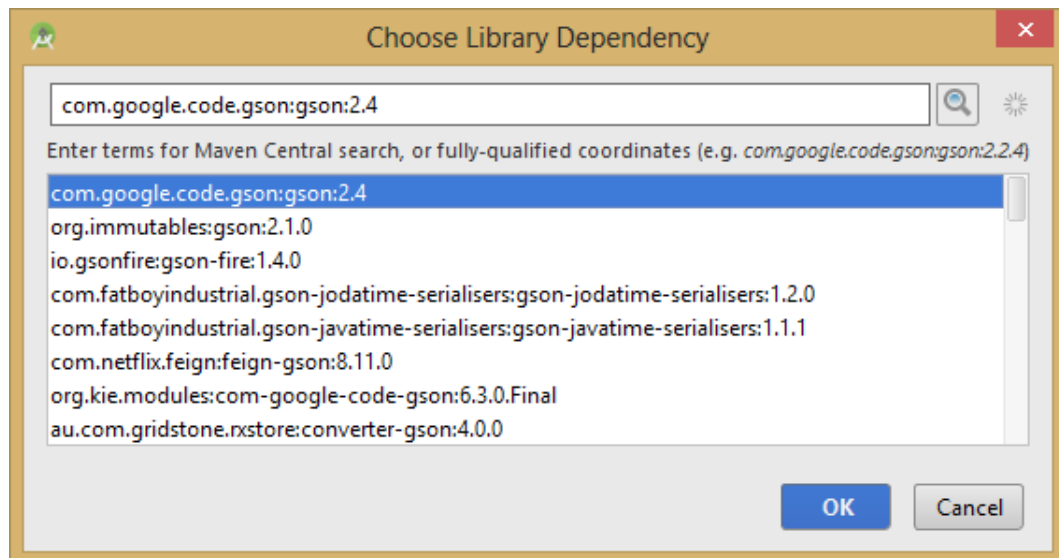


Figura 19 – Adicionando a biblioteca Gson ao projeto. **Fonte:**Elaborado pelos autores.

Depois, fez-se necessário construir uma classe que faz a ligação e organização dos dados vindos do banco de dados com a interface que listará as informações ao usuários. Essa classe recebeu o nome de `ListResultsAdapter` e estende da classe nativa do Android denominada `BaseExpandableListAdapter`.

Nesta classe foi criado um construtor que recebe a lista de disciplinas cursadas pelo aluno e uma lista com as notas de cada matéria. O nome das disciplinas foram inseridos em um *array* de *Strings*, já as notas foram inseridas em uma matriz, como apresenta a Figura 20.

```

public ListResultsAdapter(Context context, List<DisciplinaTO> disciplinas, List<EventoTO> 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;
        int somaNotas = 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++;
                somaNotas += nota;
            }
        }
        valoresMateria[i][position] = "MÉDIA: " + somaNotas;
    }
}

```

Figura 20 – Construtor da classe ListResultsAdapter. **Fonte:**Elaborado pelos autores.

Após adicionado os dados no *array* e na matriz é preciso exibí-los ao estudante. Para realizar essa tarefa foi utilizado o método `getGroupView()` para apresentar os nomes das disciplinas e o método `getChildView()` para mostrar as notas de cada matéria.

Na Figura 21, pode se ver o método `getGroupView()` criando um *widget* do tipo *textView* e inserindo nele o nome da matéria, os espaçamentos, o tamanho da fonte e informando que as palavras serão escritos em negrito.

```

@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;
}

```

Figura 21 – Método `getGroupView()`. **Fonte:**Elaborado pelos autores.

O método `getChildView()` segue a mesma lógica do método `getGroupView()`, como ilustra a Figura 22.

```
@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 22 – Método `getChildView()`. **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 gerou 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 23 é possível ver o código XML de 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 23 – Código XML do layout que apresentará a lista de notas. **Fonte:**Elaborado pelos autores.

Na classe `ListResultsActivity`, é preciso informar ao sistema operacional o *layout* a ser chamado através do método `setContentView()`. Também foi necessário passar para a classe `ListResultsAdapter` a lista de disciplinas que o discente está cursando e a lista com as notas referentes a cada matéria. Na Figura 24 é exibido o método `onCreate()` da classe `ListResultsActivity`.

```
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 24 – Método `onCreate()` da classe `ListResultsActivity`. **Fonte:**Elaborado pelos autores.

Estes procedimentos que foram realizados para as classes `ListResultsActivity` e `ListResultsAdapter` foram necessários para apresentar as notas dos exercícios resolvidos. Desta mesma forma foi preciso criar uma *activity* e um *adapter* tanto para faltas quanto para provas agendadas seguindo a mesma lógica.

No momento em que algum professor lançar notas, faltas ou provas agendadas no portal do aluno, é indispensável notificar ao estudante. Com esse intuito desenvolveu-se uma classe chamada de `GcmIntentServiceUnivas` que estende `IntentService`.

Esta classe recebe os dados em formato JSON, por isso ela transfere estas informações para o método `getJsonEventos()` da classe `HttpUtil`, o qual será responsável por ler os dados e realizar os procedimentos de gravação no banco de dados.

Ao salvar o evento é chamado o método `sendNotification()`, que receberá um objeto da classe `EventTO`. Ele realiza uma análise do tipo de evento, para saber qual *activity* deve ser executada quando o usuário clicar na notificação. Logo após foi adicionado os atributos da notificação, como o ícone, o título e a mensagem que irá aparecer ao usuário. Na Figura 25 é visível o método `sendNotification()`.


```

private void sendNotification(EventTO to) {

    String msg;

    DisciplineTO disciplineTO = execute.getDispline(to.getId_disciplina());
    String nomeDisplina = disciplineTO.getNome();

    mNotificationManager = (NotificationManager)
        this.getSystemService(Context.NOTIFICATION_SERVICE);
    PendingIntent contentIntent;

    if(to.getTipo_evento().equals("PROVA AGENDADA")){
        List<String> dados = new ArrayList<>();
        dados.add(nomeDisplina);
        dados.add(to.getDescricao_evento());
        dados.add(String.valueOf(to.getValor_evento()));
        dados.add(to.getData_evento());
        Intent intent = new Intent(this, NotificationAgendasActivity.class);
        intent.putExtra("dados", (ArrayList<String>)dados);

        contentIntent = PendingIntent.getActivity(this, 0,
            intent, 0);

        msg = "Prova agendada dia" + to.getData_evento();
    }else{
        if(to.getTipo_evento().equals("FALTAS")){
            List<String> dados = new ArrayList<>();
            dados.add(nomeDisplina);
            dados.add(to.getData_evento());
            dados.add(String.valueOf(to.getValor_evento()));
            Intent intent = new Intent(this, NotificationFoulsActivity.class);
            intent.putExtra("dados", (ArrayList<String>)dados);

            contentIntent = PendingIntent.getActivity(this, 0,
                intent, 0);

            msg = to.getValor_evento() + " falta(s) recebidas";
        }else{
            List<String> dados = new ArrayList<>();
            dados.add(nomeDisplina);
            dados.add(to.getDescricao_evento());
            dados.add(String.valueOf(to.getValor_evento()));
            dados.add(String.valueOf(to.getNota()));
            Intent intent = new Intent(this, NotificationResultsActivity.class);
            intent.putExtra("dados", (ArrayList<String>)dados);

            contentIntent = PendingIntent.getActivity(this, 0,
                intent, 0);

            msg = "Nova nota " + to.getNota();
        }
    }

    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 25 – Método sendNotification(). **Fonte:**Elaborado pelos autores.

A notificação acontece toda vez que o GCM envia uma informação ao dispositivo. Para tratar essas ocorrências foi projetada uma classe para ser o BroadcastReceiver chamada de

GcmBroadcastReceiverUnivas. Ela estende da classe WakefulBroadcastReceiver nativa do Android e possui apenas um método chamado de onReceive(), o qual receberá a *intent* a ser chamada quando chegar algum dado. Na Figura 26, pode ser vista a implementação da classe GcmBroadcastReceiverUnivas, que através do método onReceive() inicializará a classe GcmIntentServiceUnivas.

```
public class GcmBroadcastReceiverUnivas extends WakefulBroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        // Explicitly specify that GcmIntentService will handle the intent.
        ComponentName comp = new ComponentName(context.getPackageName(),
            GcmIntentServiceUnivas.class.getName());
        // Start the service, keeping the device awake while it is launching.
        startWakefulService(context, (intent.setComponent(comp)));
        setResultCode(Activity.RESULT_OK);
    }
}
```

Figura 26 – Classe GcmBroadcastReceiverUnivas. **Fonte:**Elaborado pelos autores.

No entanto para configurar esta classe como um BroadcastReceiver, foi preciso adicioná-la na *tag* <receiver> do arquivo AndroidManifest.xml, como mostra a Figura 27.

```
<receiver
    android:name=".model.GcmBroadcastReceiverUnivas"
    android:permission="com.google.android.c2dm.permission.SEND" >
    <intent-filter>
        <action android:name="com.google.android.c2dm.intent.RECEIVE" />

        <category android:name="univas.edu.com.university.model" />
    </intent-filter>
</receiver>
```

Figura 27 – Configuração do BroadcastReceiver no AndroidManifest.xml. **Fonte:**Elaborado pelos autores.

Por fim, foi construída uma classe chamada de GcmControllerUnivas que tem por objetivo configurar o dispositivo para trabalhar com o GCM.

Nesta classe existe um método denominado checkPlayServices() como intuito de verificar se o dispositivo possui os requisitos necessários para o GCM. Na Figura 28, é apresentado o método checkPlayServices().

```

public boolean checkPlayServices() {

    int resultCode = GooglePlayServicesUtil.isGooglePlayServicesAvailable(context);

    if (resultCode != ConnectionResult.SUCCESS) {

        if (GooglePlayServicesUtil.isUserRecoverableError(resultCode)) {

            GooglePlayServicesUtil.getErrorDialog(resultCode, new MainActivity(),
                PLAY_SERVICES_RESOLUTION_REQUEST).show();

        } else {

            Log.i(TAG, "Este dispositivo não suporta o GCM.");

        }

        return false;

    }

    return true;

}

```

Figura 28 – Método checkPlayServices(). **Fonte:**Elaborado pelos autores.

Logo após é chamado o método getRegistrationId(), que é incumbido de retornar o *Registration ID*. Na Figura 29 é possível ver o método getRegistrationId(). No entanto, se o registro retornado for nulo, isso significa que o aparelho não está cadastrado nos servidores da Google, então é executado o método registerInBackground() que fará esse cadastro, como demonstra a Figura 30.

```

private String getRegistrationId(Context context) {

    final SharedPreferences prefs = getGcmPreferences(context);

    String registrationId = prefs.getString(PROPERTY_REG_ID, "");
    if (registrationId.isEmpty()) {

        Log.i(TAG, "Falha ao registrar.");
        return "";

    }

    // Check if app was updated; if so, it must clear the registration ID
    // since the existing regID is not guaranteed to work with the new
    // app version.
    int registeredVersion = prefs.getInt(PROPERTY_APP_VERSION, Integer.MIN_VALUE);
    int currentVersion = getAppVersion(context);
    if (registeredVersion != currentVersion) {

        Log.i(TAG, "Versão alterada do aplicativo.");
        return "";

    }

    return registrationId;

}

```

Figura 29 – Método getRegistrationId(). **Fonte:**Elaborado pelos autores.

```

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();
                // If there is an error, don't just keep trying to register.
                // Require the user to click a button again, or perform
                // exponential back-off.
            }
            return msg;
        }

        @Override
        protected void onPostExecute(String msg) {
            Toast.makeText(new MainActivity(), msg, Toast.LENGTH_SHORT).show();
        }
    }.execute(null, null, null);
}

```

Figura 30 – Método registerInBackground(). **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.

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.

1.4.3.1 Montagem do Ambiente de Desenvolvimento

No que diz respeito à construção 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 HTTP¹ 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. 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 a IDE² 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 preferido pelo desenvolvedor.

Para armazenar os dados gerados e/ou recebidos, foi necessário fazer a instalação 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 HTTP - Hypertext Transfer Protocol

² IDE - Integrated Development Environment

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 visual de apoio para o PostgreSQL. Para criar o banco era necessário já 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 31.

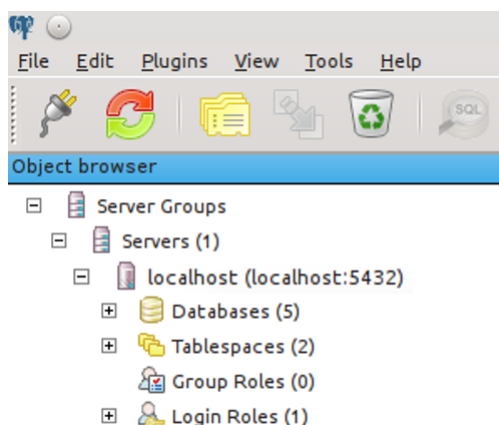


Figura 31 – Servidor de banco de dados local no PgAdmin. **Fonte:**Elaborado pelos autores.

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 32.

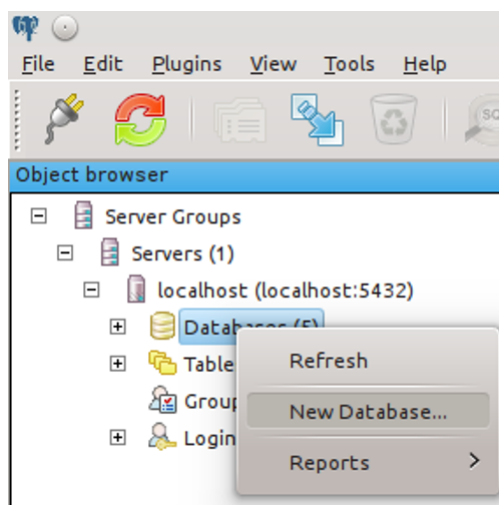


Figura 32 – Opção *New Database...* **Fonte:**Elaborado pelos autores.

Em seguida foi necessário preencher o dados da janela apresentada, como está apresentado na Figura 33.

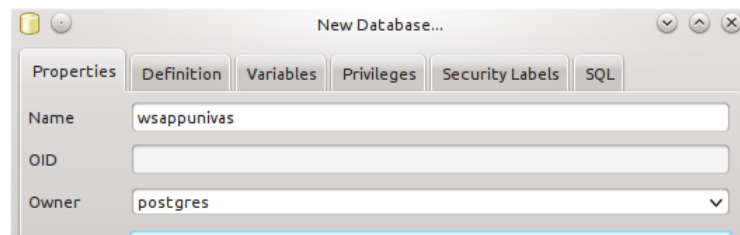


Figura 33 – Tela *New Database...* **Fonte:**Elaborado pelos autores.

Como pode ser visto foram preenchidos os campos **Name** e **Owner**. O campo **Name** se refere ao nome do banco de dados que foi definido com *wsappunivas*, e o campo **Owner**, o responsável pelo banco de dados, que para este caso foi usuário padrão do SGDB, que é o *postgres*. Além destas configurações mais nenhuma opção foi necessária. O banco de dados foi criado, porém sua estrutura não está 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 conforme Apêndice I. Antes de começar o desenvolvimento foi necessário ainda criar uma pasta a qual foi a responsável por conter todos os arquivos *.jar* das bibliotecas que foram usadas para o desenvolvimento do *web service*. Para criar uma nova pasta dentro do projeto foi necessário clicar com o botão direito do mouse sobre o projeto e navegar sobre a opção **New -> Folder**, como pode ser visto na Figura 34.

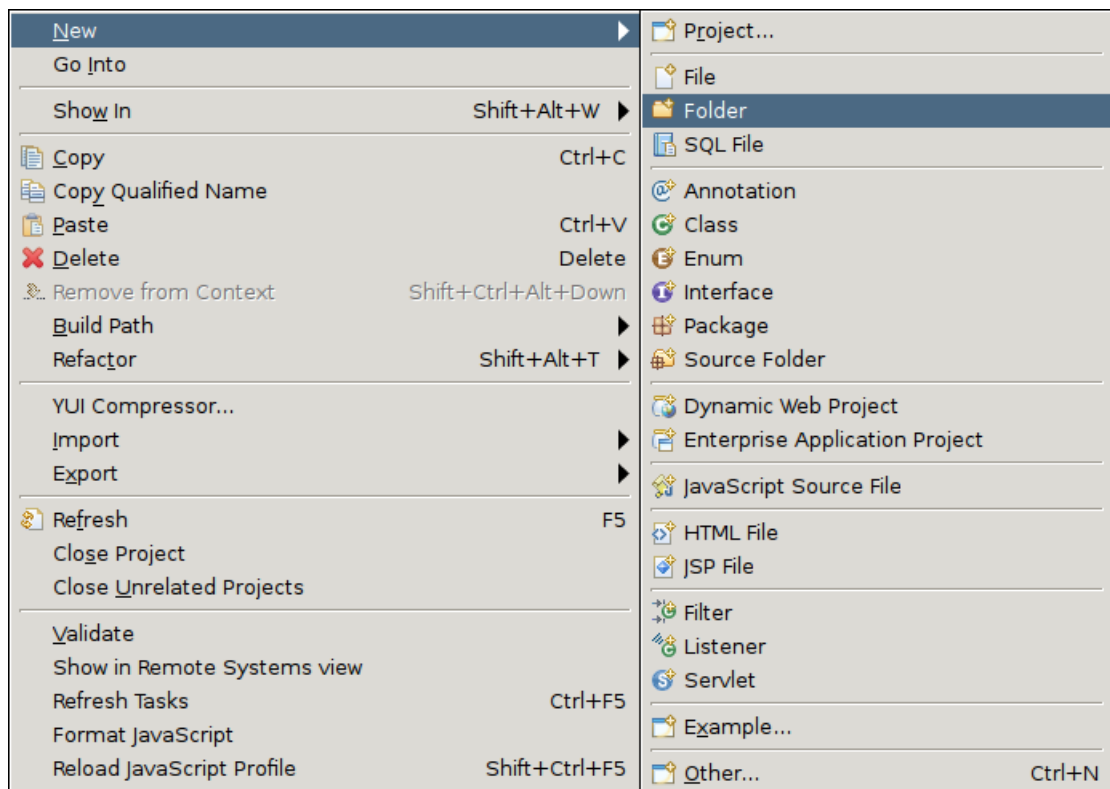


Figura 34 – Tela para criação de um novo projeto no Eclipse. **Fonte:**Elaborado pelos autores.

Em seguida na janela apresentada, foi definido o nome da nova pasta como `libs`. A tela e o nome preenchido podem ser vistos na Figura 35.

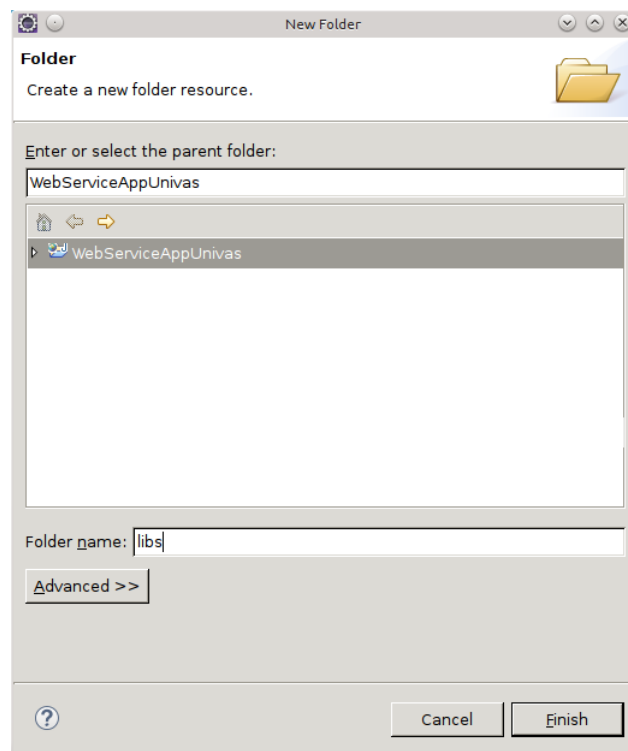


Figura 35 – Tela para criação de um novo projeto no Eclipse. **Fonte:**Elaborado pelos autores.

Em seguida foram copiados todos os arquivos .jar da biblioteca Hibernate que eram necessários ao projeto, para dentro desta pasta e também o jar do *driver* JDBC do PostgreSQL, que seria responsável por fazer a comunicação entre o banco de dados e a aplicação. Além disso era necessário mais uma configuração para que as bibliotecas pudessem ser reconhecidas como parte do projeto. Era necessário fazer a inclusão destas bibliotecas para dentro do *build path* do projeto. Para isso foi necessário selecionar todos os arquivos .jar que estavam dentro da pasta *libs*, clicar com o botão direito do mouse sobre eles e escolher a opção **Build Path -> Add to Build Path**. A pasta *libs* que foi criada, os arquivos .jar das bibliotecas usadas e a configuração do *build path* do projeto, podem ser visto na Figura 36.

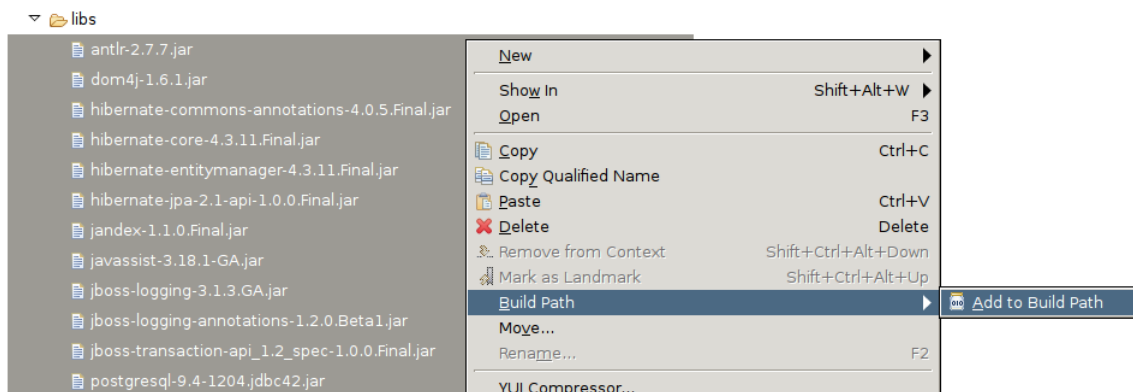


Figura 36 – Tela para criação de um novo projeto no Eclipse. **Fonte:**Elaborado pelos autores.

Com o projeto devidamente configurado, começou-se de fato a desenvolver a camada de persistência da aplicação. 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 37.

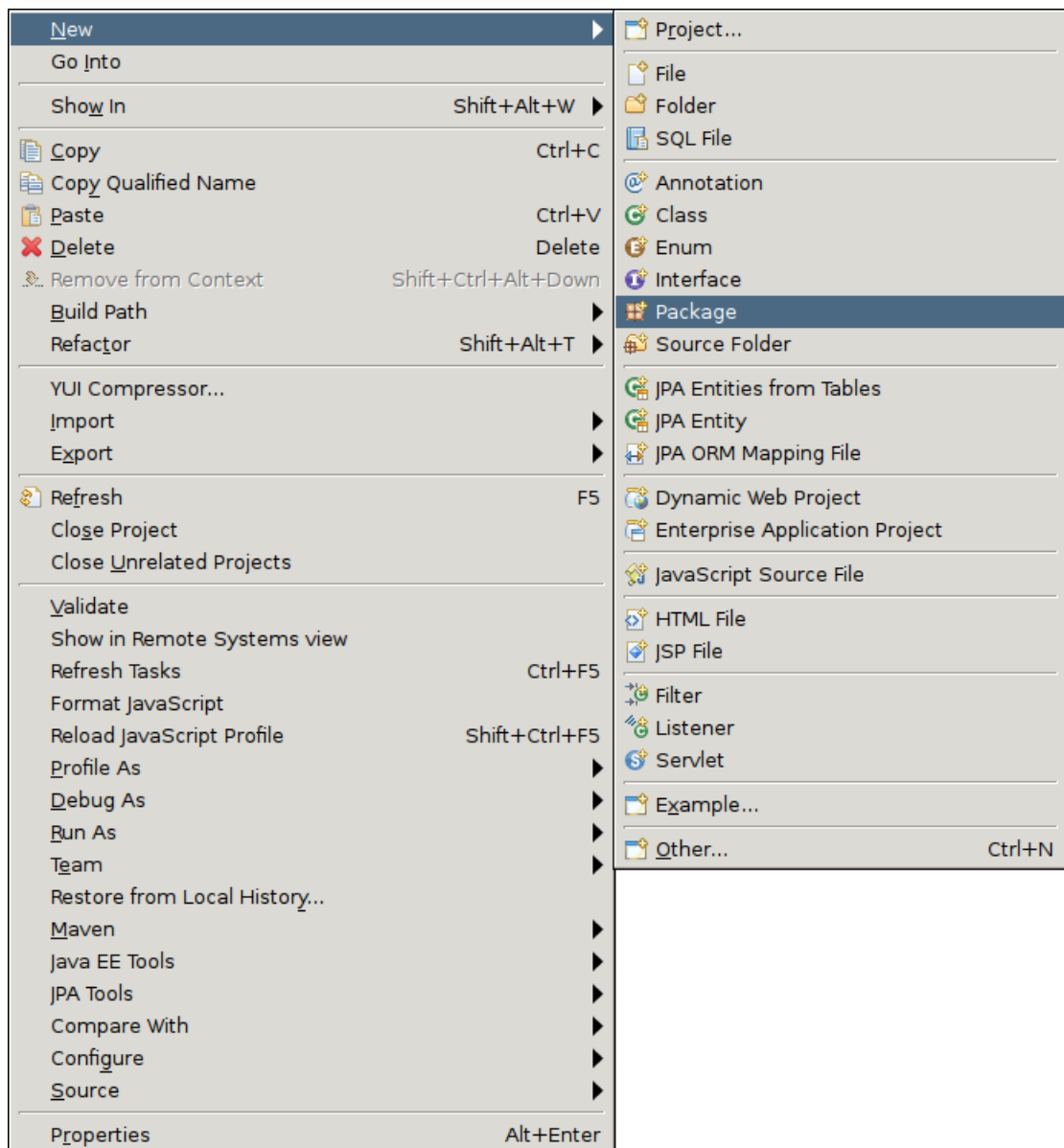


Figura 37 – Tela para criação de um novo projeto no Eclipse. **Fonte:**Elaborado pelos autores.

Em seguida foi apresentada a janela New Java Package, para a criação de um novo pacote mostrada na Figura 38. 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.

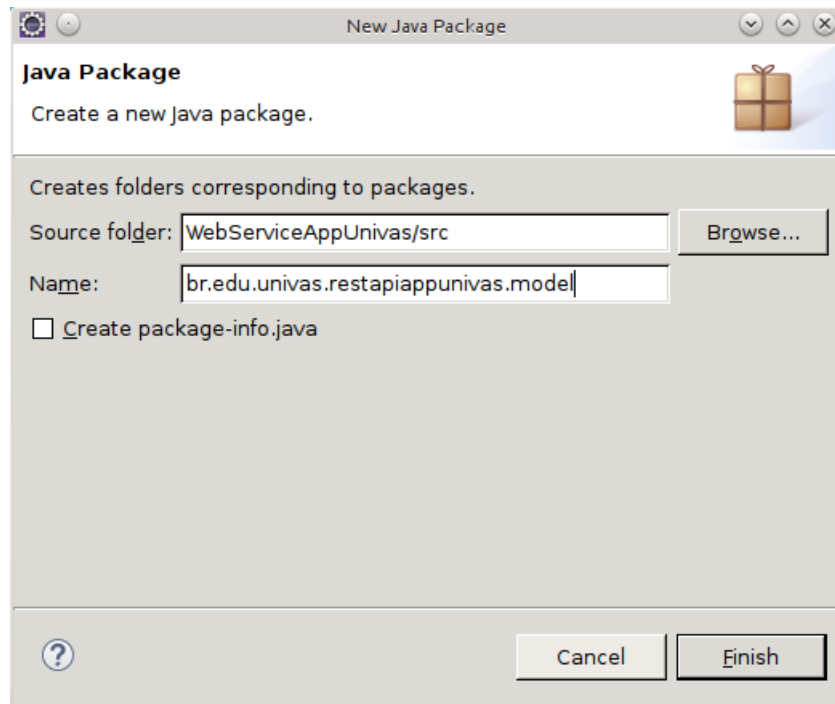


Figura 38 – Tela para criação de um novo projeto no Eclipse. **Fonte:**Elaborado pelos autores.

Com este pacote criado, já era possível criar as classes do ORM. Foi criada primeiramente a classe `Student.java`. Para a criação desta classe foi necessário clicar com o botão direito do *mouse* sobre o projeto e navegar até a opção **New -> Class** como pode ser visto na Figura 39.

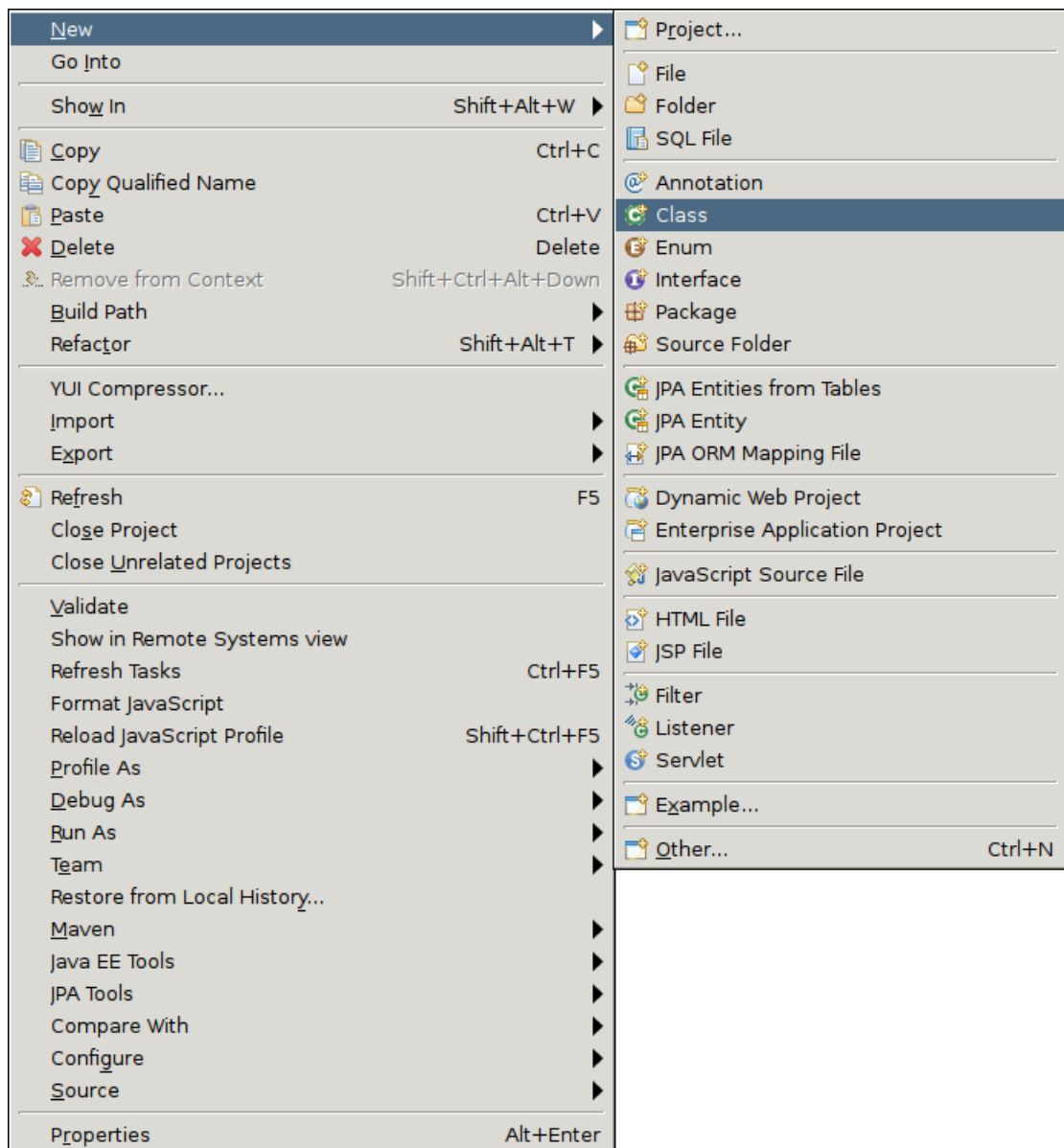


Figura 39 – Opção para Criação de nova classe java no Eclipse. **Fonte:**Elaborado pelos autores.

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 e ainda pode ser definido o pacote.

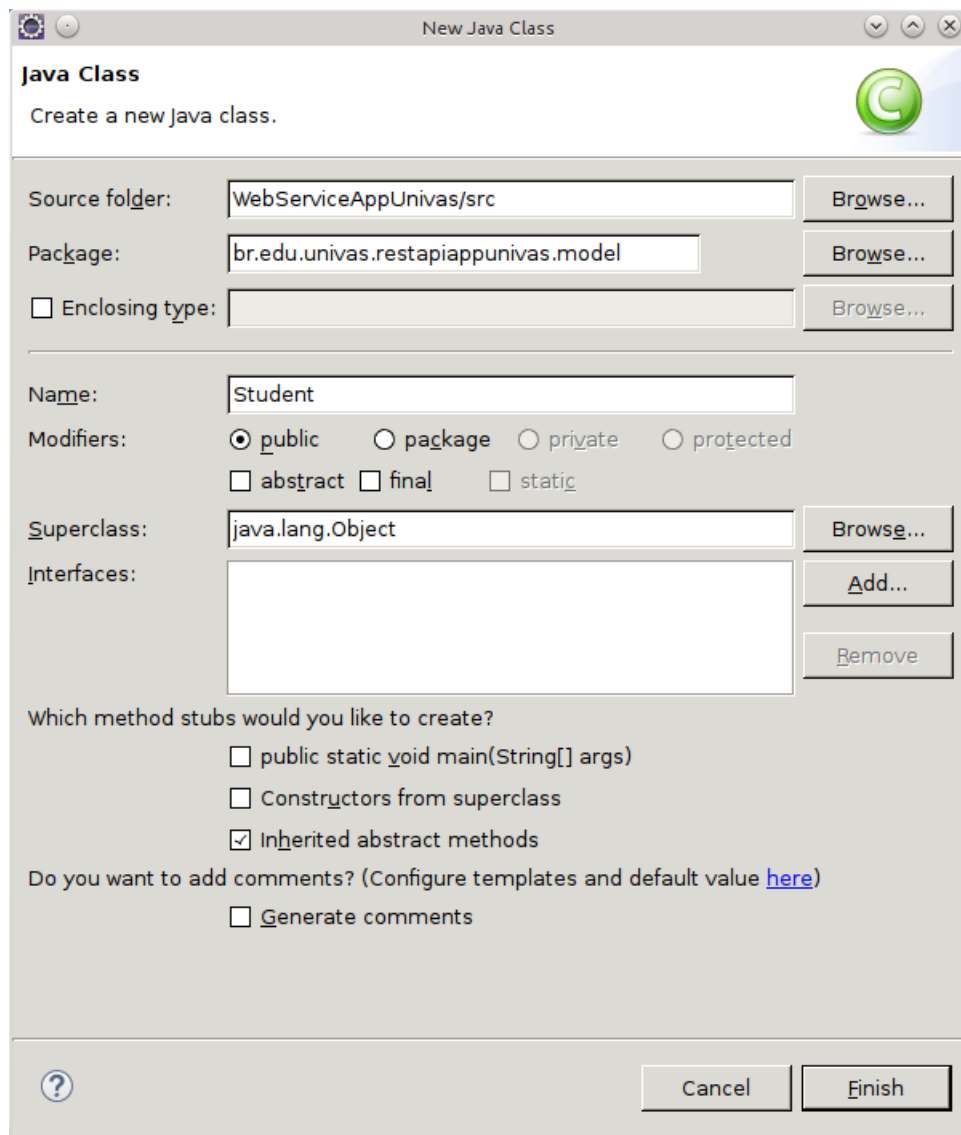


Figura 40 – Tela para Criação de nova classe java no Eclipse. **Fonte:**Elaborado pelos autores.

Esta classe foi definida para representar as informações referente aos alunos. O código fonte desta classe pode ser visto na Figura 41.

```

1  package br.edu.univas.restapiappunivas.model;
2  /**
3   *imports omitidos
4   */
5
6  @Entity
7  @Table(name = "student")
8  public class Student {
9
10     @Id
11     @SequenceGenerator(name = "id_student", sequenceName = "
12         seq_id_student",
13         allocationSize = 1)
14     @GeneratedValue(generator = "id_student", strategy =
15         GenerationType.IDENTITY)
16     @Column(name = "id_student", nullable = false)
17     private Long idStudent;
18
19     @Column(name = "id_external", nullable = false)
20     private Long idDatabaseExternal;
21
22     @Column(length = 100, nullable = false)
23     private String name;
24
25     @Column(length = 100, nullable = false)
26     private String email;
27
28     @OneToMany(mappedBy="student", fetch = FetchType.EAGER)
29     private List<Event> events;
30
31     @OneToOne(optional = false, fetch = FetchType.LAZY)
32     @JoinColumn(name = "id_user")
33     private User user;
34
35     /**
36     * Omitidos todos Getters e Setters
37     */
38
39     @Override
40     public int hashCode() {
41         /**
42         * Omitido
43         */
44     }
45
46     @Override
47     public boolean equals(Object obj) {
48         /**
49         * Omitido
50         */
51     }
52 }

```

Figura 41 – Classe Student.java. **Fonte:**Elaborado pelos autores.

É válido lembrar que 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 uma 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;
- `@GeneratedValue`: esta anotação indica qual será a estratégia usada para incrementar a chave primária da tabela.
- `@SequenceGenerator`: esta anotação define o mecanismo com que a chave primária será incrementada.
- `@Column`: define algumas propriedades do campo da tabela do banco de dados, o qual o atributo que ele anota representa. Estas configurações podem são:
 - `name`: mapeia 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. Anota 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 persistência de dados e é simplesmente um POJO ou seja, um objeto simples que contém somente atributos privados e os métodos *getters* e *setters* que servem apenas para encapsular estes atributos, e não possui nenhuma lógica de negócios. Além desta classe, foram criadas outras com os mesmos propósitos, que podem ser vista no diagrama da Figura 42.



Figura 42 – Modelo físico do banco de dados. **Fonte:**Elaborado pelos autores.

Estas classes tinham a mesma finalidade da anterior, porém com pequenas diferenças no que diz respeito aos atributos, metodos e anotações. Estas classes representam, de maneira individual, as tabelas no banco de dados. O modelo físico do banco de dados pode ser visto na Figura 43.

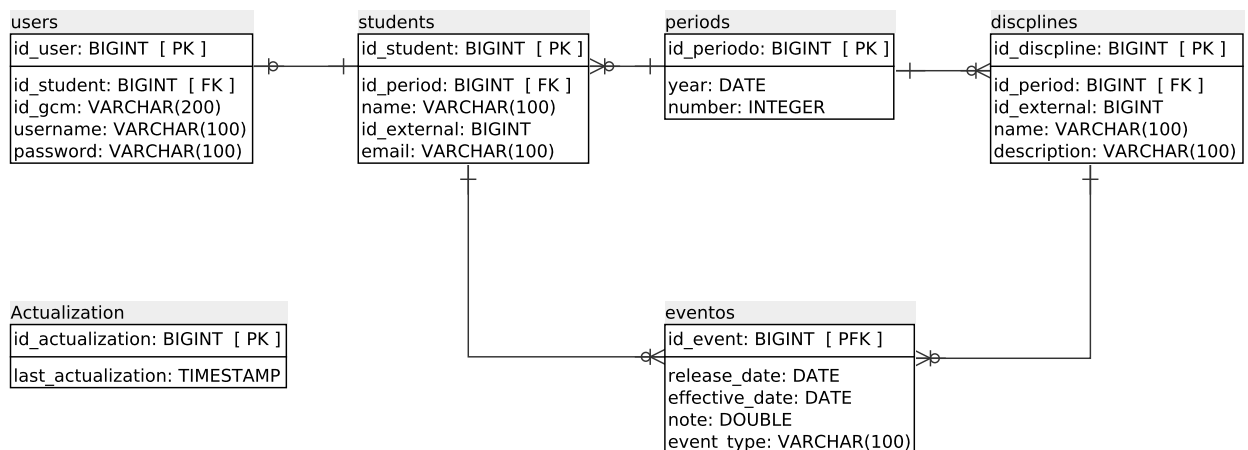


Figura 43 – Modelo físico do banco de dados. **Fonte:**Elaborado pelos autores.

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 fácil 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 44.

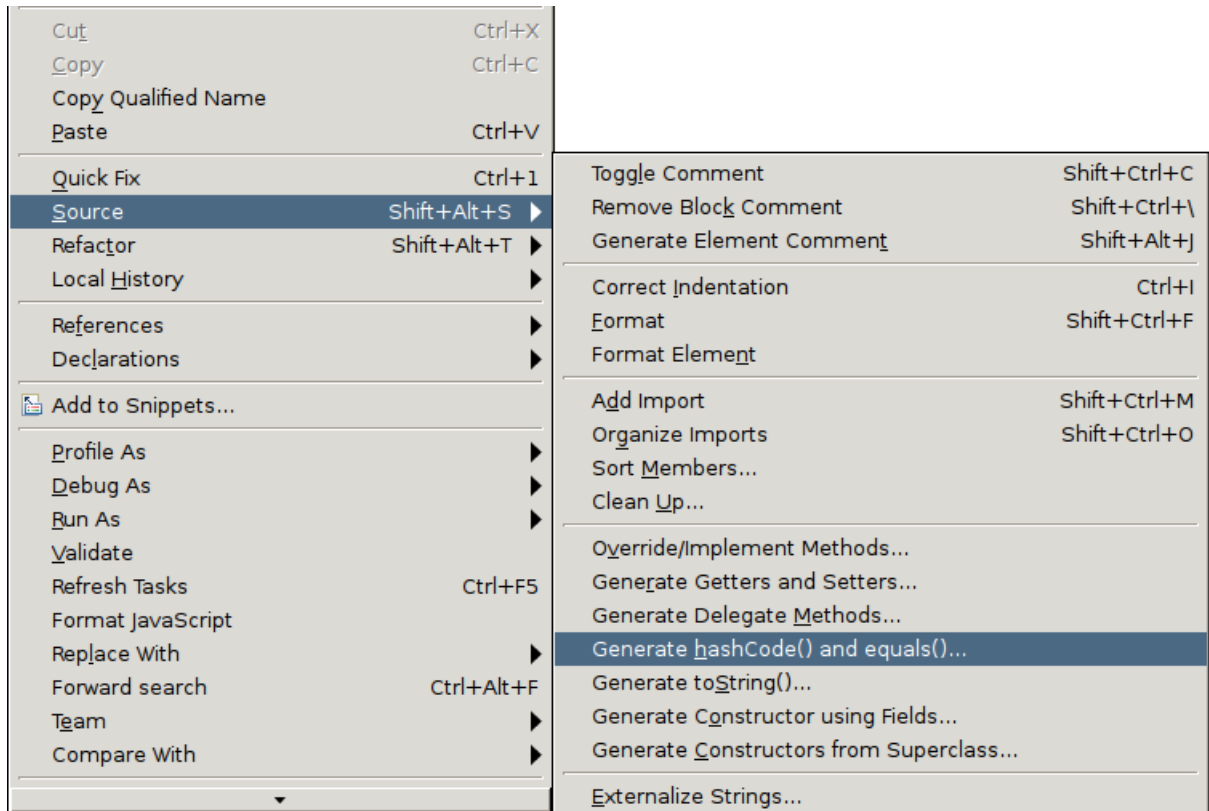


Figura 44 – Opção Generate hashCode() and equals()... **Fonte:**Elaborado pelos autores.

Em seguida na janela que foi apresentada foi necessário escolher qual atributo seria usado para a comparação dentro dos métodos, como esta apresentado na Figura 45 .

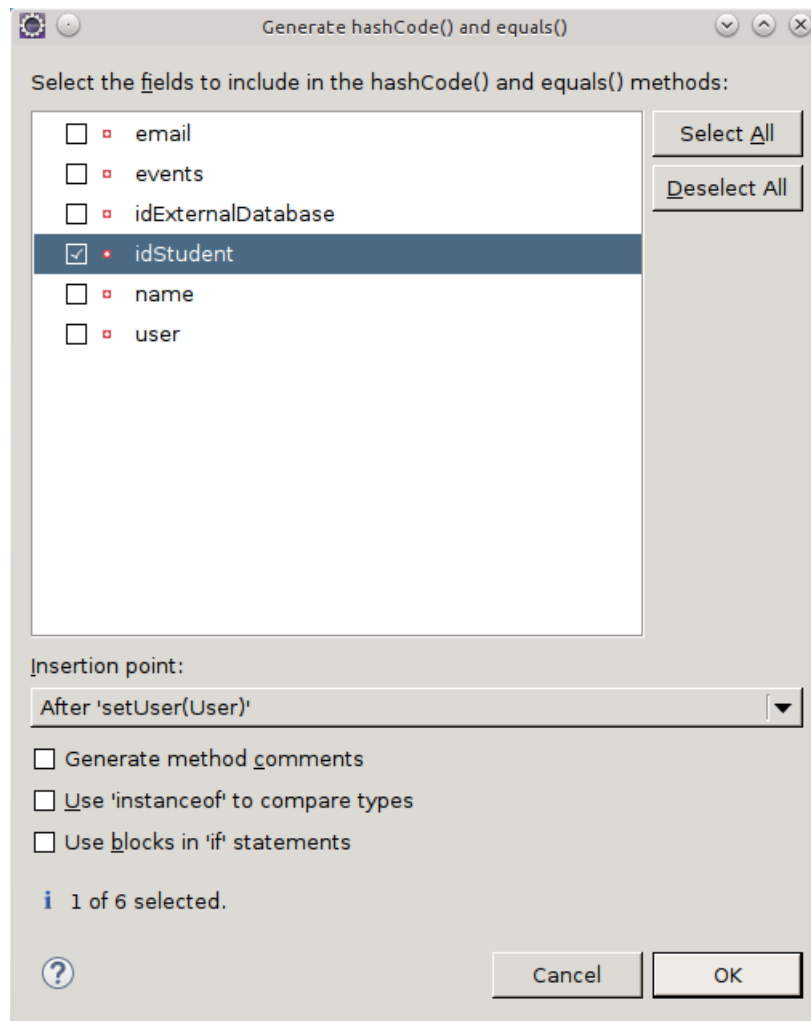


Figura 45 – Opção Generate hashCode() and equals(). . . **Fonte:**Elaborado pelos autores.

Os métodos hashCode e equals da classe Student.java foram implementados usando o atributo idStudent e podem ser vistos na Figura 48.

```

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 46 – Implementação os métodos hashCode() e equals(). **Fonte:**Elaborado pelos autores.

Além destas classes, foi necessário criar um tipo enumerado (ou enum), para definir quais seriam os tipos dos eventos, haja vista que estes teriam um numero limitado de possibilidades. Para a criação desta, foi necesário clicar com o botão direito do *mouse* sobre o projeto e navegar até a opção **New -> Enum** como pode ser visto na Figura 49.

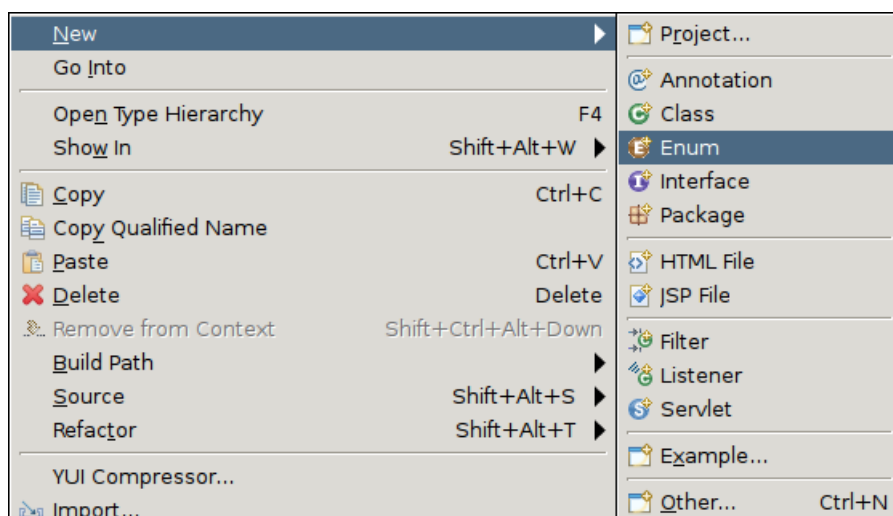


Figura 47 – Opção Generate hashCode() and equals(). ... **Fonte:**Elaborado pelos autores.

Para esta enumeração foi definido o nome `EventType`. Os tipos de eventos definidos foram três:

- `PROVA_AGENDADA`: que define um evento como agendamento de uma atividade avaliativa;
- `PROVA_APLICADA`: que define um evento como, a efetiva realização de uma atividade avaliativa;
- `FALTAS`: representa o lançamento de uma falta;

A implementação da enumeração pode ser vista na Figura 48.

```
1
2 package br.edu.univas.restapiappunivas.model;
3
4 public enum EventType {
5     PROVA_AGENDADA, PROVA_APLICADA, FALTAS
6 }
```

Figura 48 – Implementação os métodos `hashCode()` e `equals()`. **Fonte:**Elaborado pelos autores.

Após a criação das entidades, foi necessário configurar o arquivo `persistence.xml`. Foi necessário criar a pasta META-INF dentro da pasta de de códigos fontes do projeto que também é conhecida como *classpath*, com a finalidade de abrigar este arquivo. Em seguida foi criado o arquivo dentro da pasta criada. 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. Este por sua vez recebeu o nome de `WsAppUnivas`. Uma destas configurações merece uma atenção especial trata-se da configuração `<property name="hibernate.hbm2ddl.auto" value="create"/>` que é a responsável por definir que o próprio Hibernate irá criar a estrutura do banco de dados (tabelas, sequences, etc.) através do mapeamento objeto relacional das classes. O arquivo `persistence.xml` está exposto na Figura 49.

```

<?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 49 – Arquivo persistence.xml. **Fonte:**Elaborado pelos autores.

Em seguida à confecção do persistence.xml foi criada a classe JpaUtil.java que está representada na Figura 50. Para isso foi necessário criar um pacote que seria responsável por armazená-la para que a organização do projeto pudesse ser mantida. Tal pacote recebeu o nome de de "br.edu.univas.restapiappunivas.util" e poderia se necessário abrigar outras classes de utilidades do projeto assim como a classe JpaUtil.java.

A classe JpaUtil.java é responsável por criar uma EntityManagerFactory. Este por sua vez é uma fábrica de instâncias de EntityManager que é um *persistence unit* ou unidade de persistência. O *persistence unit* foi configurado através do arquivo persistence.xml mostrado anteriormente. Este por sua vez tem a responsabilidade de prover um modo de comunicação entre a aplicação e o banco de dados. No entanto a classe JpaUtil.java cria uma única instância de EntityManagerFactory, que é responsável por disponibilizar e gerenciar as instâncias de EntityManager de acordo com a necessidade da aplicação.

```

1 package br.edu.univas.restapiappunivas.util;
2
3 /**
4  *Imports Omitidos
5  */
6
7 public class JpaUtil {
8     private static EntityManagerFactory factory;
9
10    static {
11        factory = Persistence.createEntityManagerFactory("WsAppUnivas");
12    }
13
14    public static EntityManager getEntityManager() {
15        return factory.createEntityManager();
16    }
17
18    public static void close() {
19        factory.close();
20    }
21
22 }

```

Figura 50 – Classe JpaUtil.java. **Fonte:**Elaborado pelos autores.

Depois de finalizada a criação da camada de persistência do projeto foi necessário uma configuração adicional. Percebeu-se que além das bibliotecas já usadas no projeto, seriam necessárias mais algumas bibliotecas para que se pudesse chegar ao resultado final esperado. Por esse motivo tornava-se inviável ficar controlando as bibliotecas de maneira manual no projeto. Foi necessário então, fazer a conversão do projeto para um projeto que fosse controlado pelo Maven. Para tanto foi necessário clicar com o botão direito do mouse sobre o projeto e navegar até a opção **Configure -> Convert to Maven Project**, como pode ser visto na Figura 53

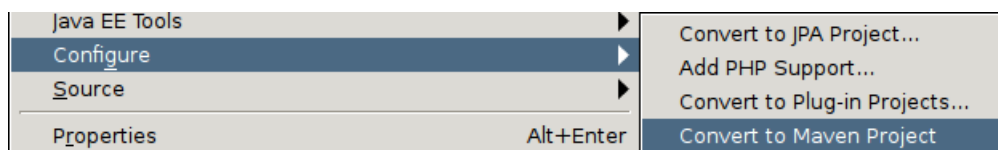


Figura 51 – Opção Generate hashCode() and equals(). . . **Fonte:**Elaborado pelos autores.

Na janela que foi apresentada na sequência, foi necessário preencher os campos **Group id**, **Artifact id**, **Version** e **Packaging**, da seguinte forma:

- **Group id** representa id do grupo a qual pertence o projeto, que recebeu o nome de "br.edu.univas".
- **Artifact id:** foi preenchido como o nome do próprio projeto já criado anteriormente pois, este seria o nome do artefato final do gerado pelo projeto.
- **Version** versão a qual está o projeto. Neste caso manteve-se o que já veio por padrão.

- **Packaging** a forma como o projeto seria empacotado após a compilação do mesmo. Foi escolhido ao empacotamento do tipo `war` por se tratar de um projeto Java para *web*.

A janela apresentada com as informações preenchidas podem ser vistas na Figura 54.

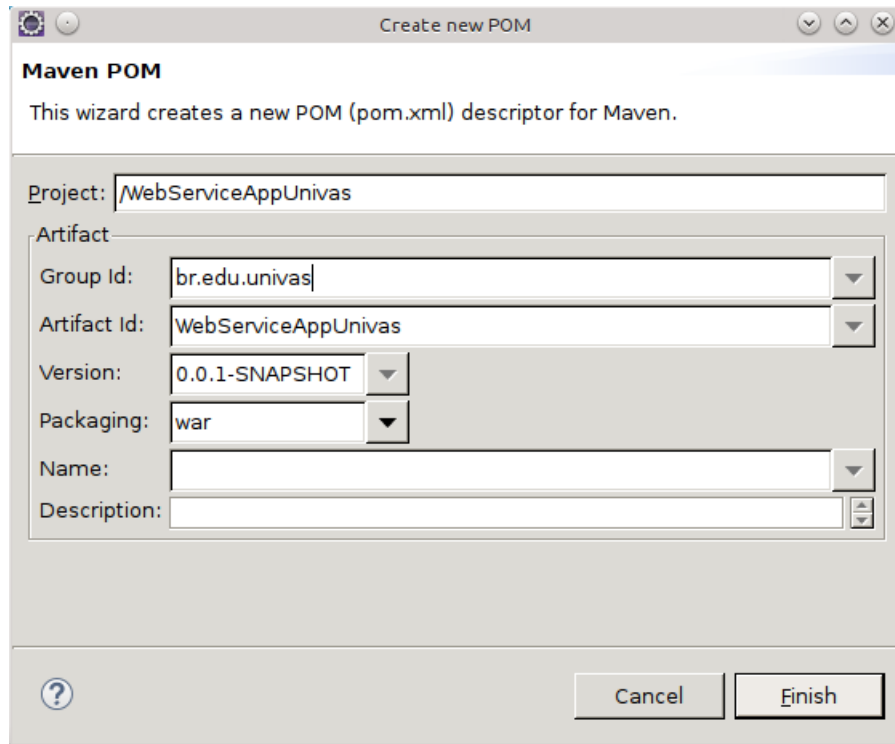


Figura 52 – Opção Generate hashCode() and equals(). . . **Fonte:**Elaborado pelos autores.

Com a conclusão da conversão do projeto, foi gerado o arquivo `pom.xml`. Este arquivo provê as informações necessárias para que o Maven faça a gerência do projeto. Este arquivo possui, além das informações apresentadas anteriormente (**Group id**, **Artifact id**, **Version** e **Packaging**), algumas informações a respeito da compilação do projeto e o principal que é a gerência das dependências do projeto. Inicialmente foi incluída somente as dependências referentes ao *driver* JDBC do PostgreSQL e a biblioteca Hibernate como pode ser visto na Figura 53.

```

<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.edu.univas</groupId>
  <artifactId>WebServiceAppUnivas</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <build>
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.4</version>
        <configuration>
          <warSourceDirectory>WebContent</warSourceDirectory>
          <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <!--driver JDBC postgresql -->
    <dependency>
      <groupId>org.postgresql</groupId>
      <artifactId>postgresql</artifactId>
      <version>9.2-1003-jdbc4</version>
    </dependency>
    <!--hibernate implementation -->
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-core</artifactId>
      <version>4.3.11.Final</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-entitymanager</artifactId>
      <version>4.3.11.Final</version>
    </dependency>
  </dependencies>
</project>

```

Figura 53 – Arquivo pom.xml. **Fonte:**Elaborado pelos autores.

Em seguida a essa configuração foi necessário apagar a pasta `libs` do projeto, pois esta já não era mais necessária, haja vista que as bibliotecas usadas no projeto já estavam sendo ge-

reenciadas pelo Maven. A partir daí foi necessário criar a parte de disponibilização dos serviços REST. Para isto foram criados três novos pacotes para abrigar as classes que formaram a parte de serviços do *web service*, são eles:

- `br.edu.univas.restapiappunivas.controllers`: pacote responsável por agrupar as classes responsáveis por realizar as consultas na camada de persistência do projeto e prover o resultados para publicação nos serviços.
- `br.edu.univas.restapiappunivas.entities`: pacote responsável por agrupar as classes que serviriam para abrigar os dados retonados nas consultas dos *controllers* e servir como retorno aos métodos das classes de serviços.
- `br.edu.univas.restapiappunivas.resources`: pacote responsável por agrupar as classes responsáveis por prover os serviços REST.

Foi necessário criar algumas classes que eram responsáveis por fazer as consultas no banco de dados usando o Hibernate. Estas classes tinham alguns métodos que tinham a responsabilidade de fazer a consulta e retornar os dados que seriam usados pelas classes que iam tornar disponíveis os serviços do *web service*. Uma destas classes é a `StudentEventsCtrl.java`, que pode ser vista na Figura 54.

```

1 package br.edu.univas.restapiappunivas.controller;
2 /**
3  * Imports omitidos
4  */
5 public class StudentEventCtrl {
6
7     public StudentEvent getEventsByRegistration(Long idStudent) {
8
9         EntityManager em = JpaUtil.getEntityManager();
10
11         String jpql = "select distinct e.idEvent, e.date, e.value, e.note,
12             ";
13         jpql += " e.description, e.eventType, d.idDiscipline, d.
14             idDatabaseExternal from Discipline d ";
15         jpql += " right outer join d.events e right outer join e.student s
16             where s.idDatabaseExternal = :id ";
17
18         try {
19             Query query = em.createQuery(jpql);
20             query.setParameter("id", idStudent);
21
22             List<Object[]> resultSet = query.getResultList();
23             List<StudentEvent> studentEvents = new ArrayList<StudentEvent>()
24                 ;
25
26             for (Object[] obj : resultSet) {
27                 StudentEvent se = new StudentEvent();
28                 se.setIdEvent((Long) obj[0]);
29                 se.setDate((Date) obj[1]);
30                 se.setValue((int) obj[2]);
31                 se.setNote((int) obj[3]);
32                 se.setDescription((String) obj[4]);
33                 se.seteventType((TipoEvento) obj[5]);
34                 se.setIdDiscipline((Long) obj[6]);
35                 se.setidDatabaseExternal((Long) obj[7]);
36
37                 studentEvents.add(se);
38             }
39
40             StudentEvents students = new StudentEvents();
41             students.setEvents(studentEvents);
42
43             return students;
44         } catch (Exception e) {
45
46             e.printStackTrace();
47
48             throw new WebApplicationException(Status.NOT_FOUND);
49         } finally {
50
51             em.close();
52         }
53     }
54 }

```

Figura 54 – Arquivo pom.xml. **Fonte:**Elaborado pelos autores.

Esta classe é composta unicamente pelo método `getEventsByRegistration()` o qual era responsável por fazer a busca de todos os eventos de um determinado aluno pela sua ma-

trícula. Ainda na Figura 54 é perceptível que dentro deste método são usadas instancias das classes StudentEvent e StudentEvents. A classe StudentEvent foi criada apenas para servir como um simples POJO que conteria os retorno da consulta JPQL³. O código desta classe pode ser visto na Figura 55.

```
1 package br.edu.univas.restapiappunivas.entities;
2
3 /**
4  * Imports Omitidos
5  */
6 public class StudentEvent {
7
8     @XmlElement(name = "id_event")
9     private Long idEvent;
10
11     @XmlElement(name = "date")
12     private Date effectiveDate;
13
14     @XmlElement(name = "value")
15     private int value;
16
17     @XmlElement(name = "note")
18     private int note;
19
20     @XmlElement(name = "description")
21     private String description;
22
23     @XmlElement(name = "event_type")
24     private TipoEvento eventType;
25
26     @XmlElement(name = "id_discipline")
27     private Long idDiscipline;
28
29     @XmlElement(name = "id_external_discipline")
30     private Long idExternalDiscipline;
31
32     @XmlElement(name = "id_student")
33     private Long idStudent;
34
35     /**
36     * Metodos getters e setter Omitidos
37     */
38
39 }
```

Figura 55 – Arquivo pom.xml. **Fonte:**Elaborado pelos autores.

³ JPQL - *Java Persistence Query Language*

Por outro lado a classe `StudentEvents` recebe uma coleção de `StudentEvent` para que a mesma pudesse ser convertida corretamente para JSON, nas classes que disponibilizam o serviço. O código fonte desta classe pode ser visto na Figura 56.

```
1 package br.edu.univas.restapiappunivas.entities;
2 /**
3  *Imports omitidos
4  */
5
6 @XmlRootElement
7 public class StudentEvents {
8
9     @XmlElement
10    private List<StudentEvent> events = new ArrayList<StudentEvent>();
11
12    public List<StudentEvent> getEvents() {
13        return events;
14    }
15
16    public void setEvents(List<StudentEvent> events) {
17        this.events = events;
18    }
19
20 }
```

Figura 56 – Arquivo pom.xml. **Fonte:**Elaborado pelos autores.

Nas duas classes mostradas anteriormente nas Figuras 55 e 56 são usadas algumas anotações.

Era necessário também que de tempos em tempos o próprio *web service* fizesse uma busca internamente, fazendo o levantamento de novos eventos lançados para os alunos para que os os mesmos pudessem ser enviados ao GCM para que este pudesse notificar os dispositivos cadastrados.

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 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. Para esta pesquisa o módulo foi simulado.

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.

Apêndices

CRIAÇÃO DE UM PROJETO DYNAMIC WEB PROJECT NO ECLIPSE LUNA

Para proceder com a criação de um projeto do tipo *Dynamic Web Project* no Eclipse, é necessário acessar na IDE, a opção **File -> New-> Dynamic Web Project** como pode ser visto na Figura 57.

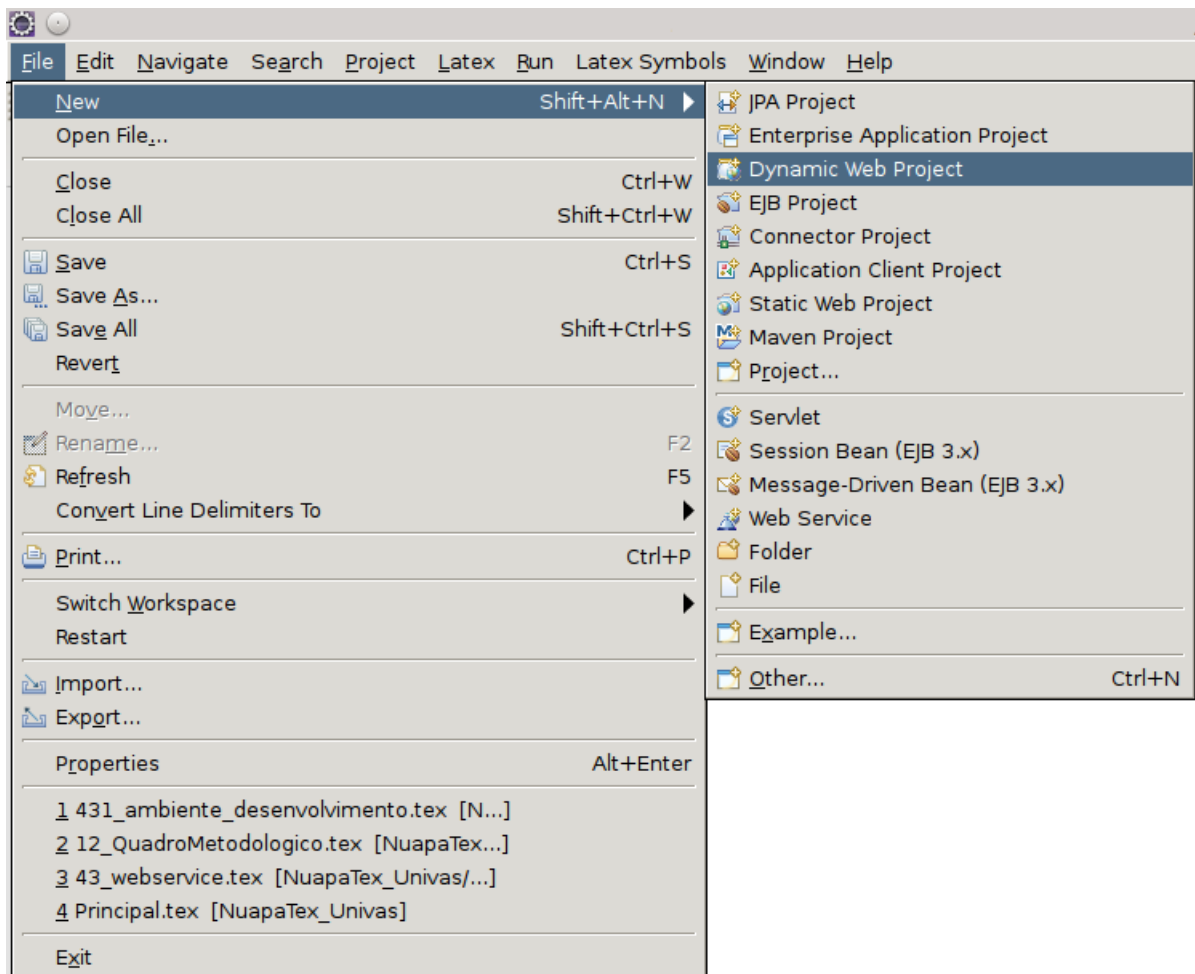


Figura 57 – Tela *New Database...* Fonte:Elaborado pelos autores.

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 58.

The screenshot shows the 'New Dynamic Web Project' dialog box in Eclipse. The title bar reads 'New Dynamic Web Project'. The main heading is 'Dynamic Web Project' with a subtitle 'Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.' The dialog is divided into several sections: 'Project name' with a text field containing 'WebServiceAppUnivas'; 'Project location' with a checked 'Use default location' checkbox and a text field showing the path '/home/diogenes/workspace1/WobServiceAppUnivas'; 'Target runtime' with a dropdown menu set to 'Apache Tomcat v7.0' and a 'New Runtime...' button; 'Dynamic web module version' with a dropdown menu set to '3.0'; 'Configuration' with a dropdown menu set to 'Default Configuration for Apache Tomcat v7.0' and a 'Modify...' button; 'EAR membership' with an unchecked 'Add project to an EAR' checkbox, an 'EAR project name' field set to 'EAR', and a 'New Project...' button; and 'Working sets' with an unchecked 'Add project to working sets' checkbox and a 'Select' button. At the bottom, there are four buttons: a help icon (?), '< Back', 'Next >', 'Cancel', and 'Finish'.

Figura 58 – 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 59.

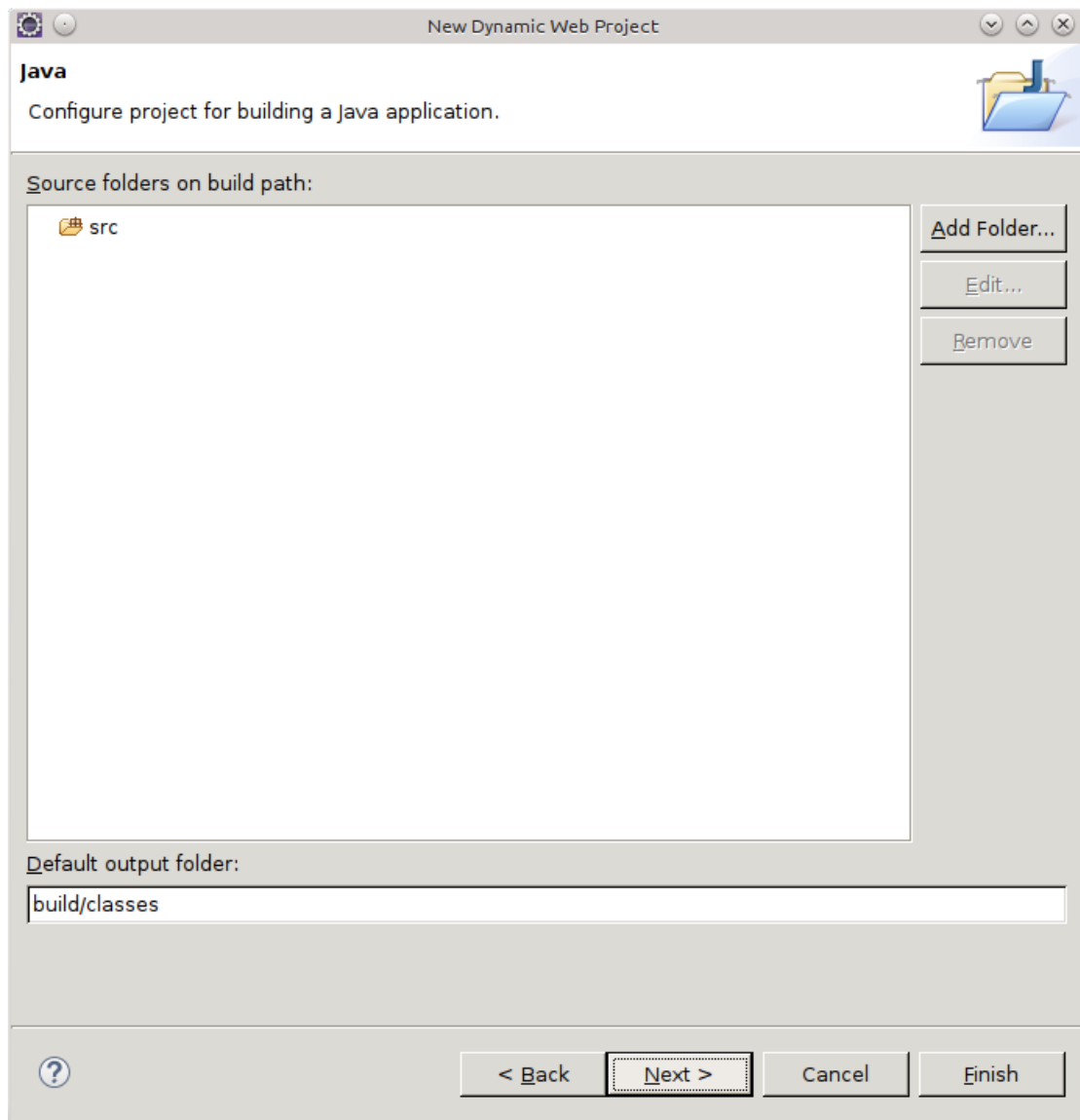


Figura 59 – Tela para criação de um novo projeto no Eclipse. **Fonte:**Elaborado pelos autores.

Na sequência, 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 marcada 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 está apresentada na Figura 60.

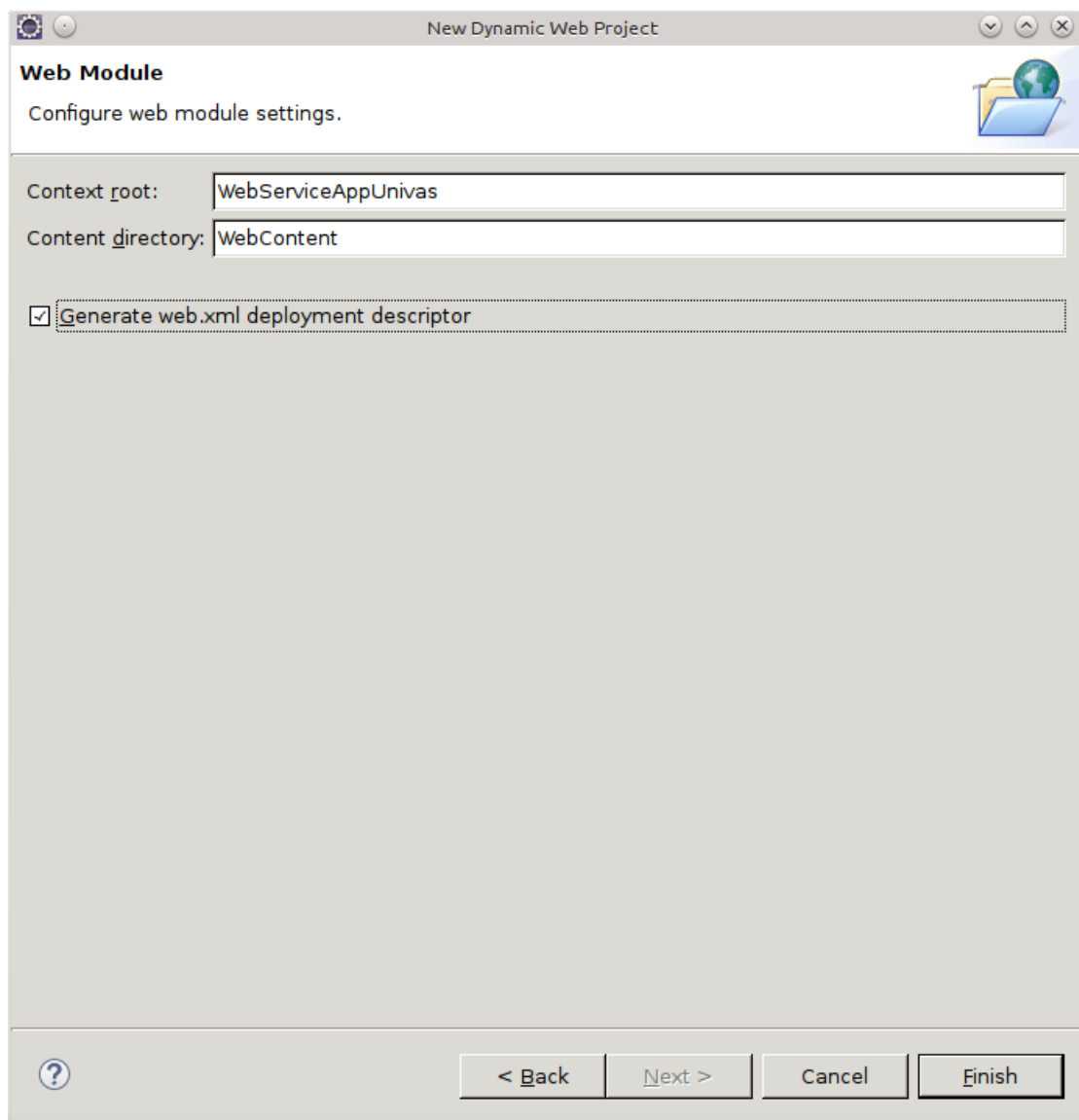


Figura 60 – Tela para criação de um novo projeto no Eclipse. **Fonte:**Elaborado pelos autores.

Após este passo foi concluído a criação do projeto.