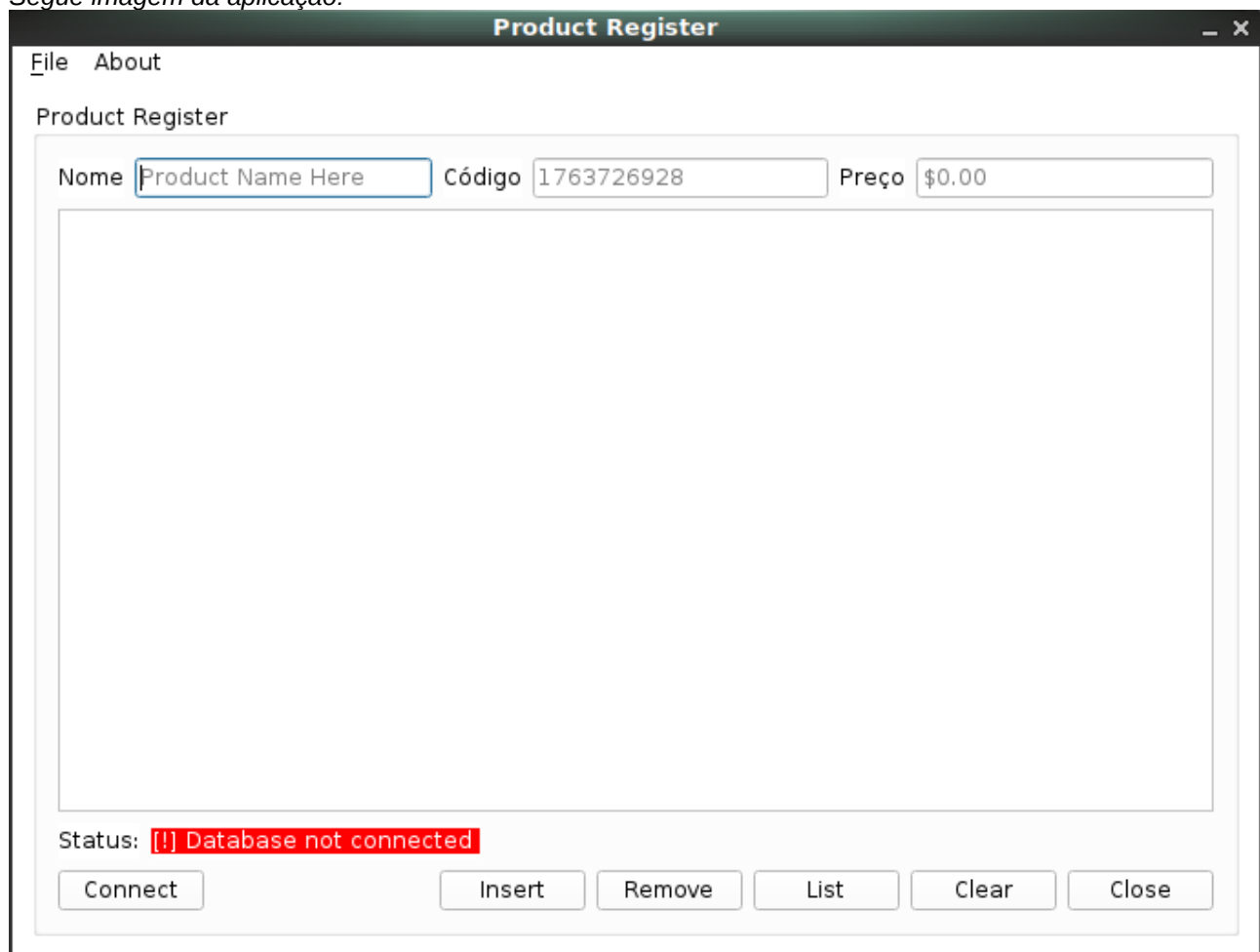


Iniciando uma aplicação em C++ no Qt Creator utilizando Banco de Dados Relacional PostgreSQL

Esta aplicação gerencia uma empresa que visa catalogar todos os seus produtos, armazenando-os de forma organizada num Banco de Dados Relacional (PostgreSQL). A aplicação armazena os dados do produto: Nome do Produto, Código do Produto e Preço do Produto. Outras funcionalidades também foram implementadas ao sistema, tais como a listagem dos produtos, a remoção de algum determinado produto, a inserção de um novo produto, conexão ao Banco de Dados Relacional (PostgreSQL), e também requisitos não funcionais ao sistema, como a limpeza dos campos preenchidos na aplicação.

Segue imagem da aplicação:

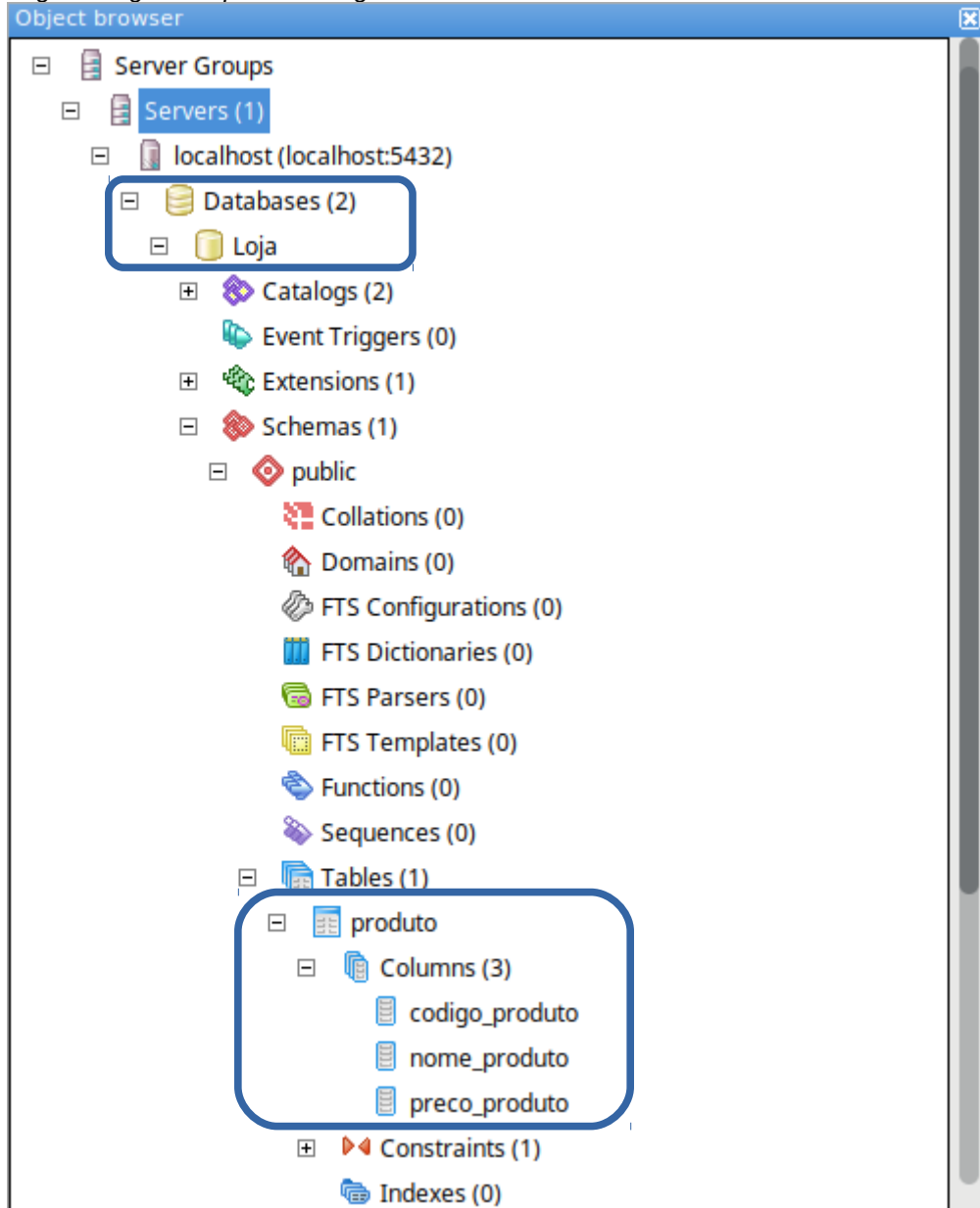


* Neste tutorial você irá aprender a como criar uma aplicação semelhante a essa, garantindo todas as integridades do sistema. O foco é aprender sobre as restrições da aplicação quando em conjunto ao Banco de Dados Relacional, tais como mensagens retornadas pelo sistema quando há algum erro de integridade ou sintaxe no programa.

Tutorial do Projeto

1º Passo: É necessário criar um banco de dados. Recomenda-se utilizar uma interface de gerenciamento, como o PgAdmin III, ou o próprio terminal.

Segue imagem do painel do PgAdmin III:

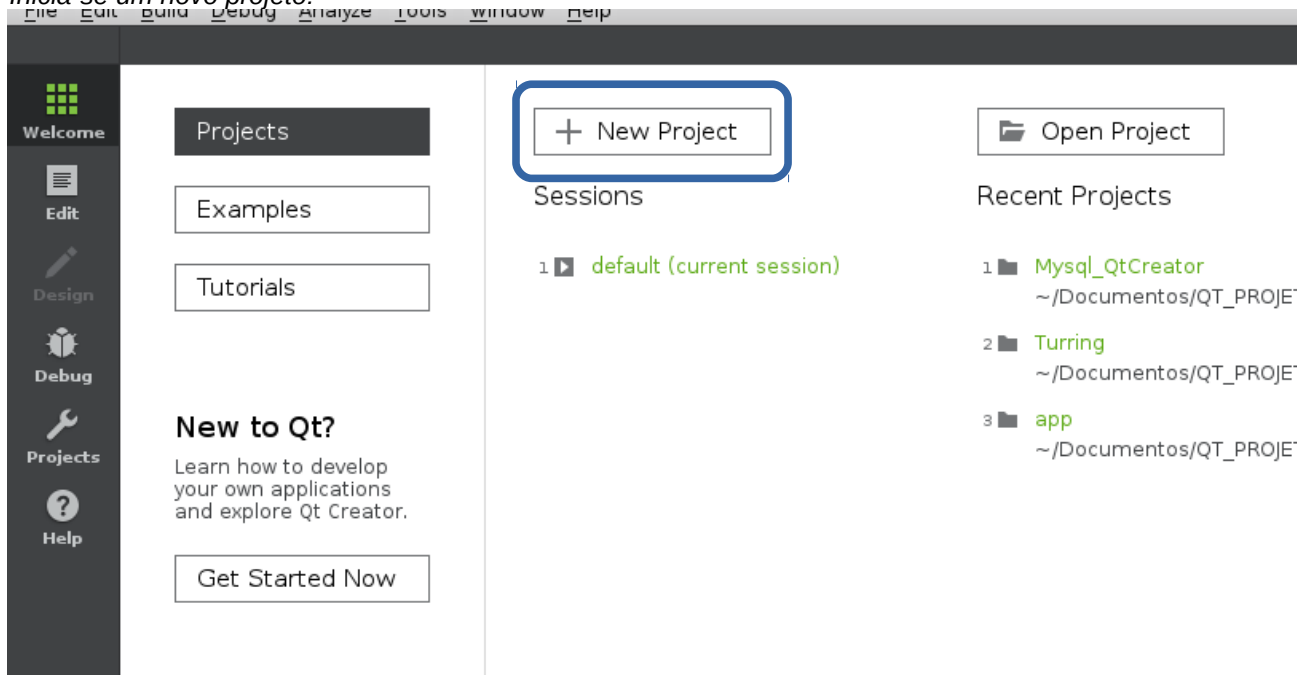


* Crie seu banco de dados de acordo com BD o demonstrado na figura acima:

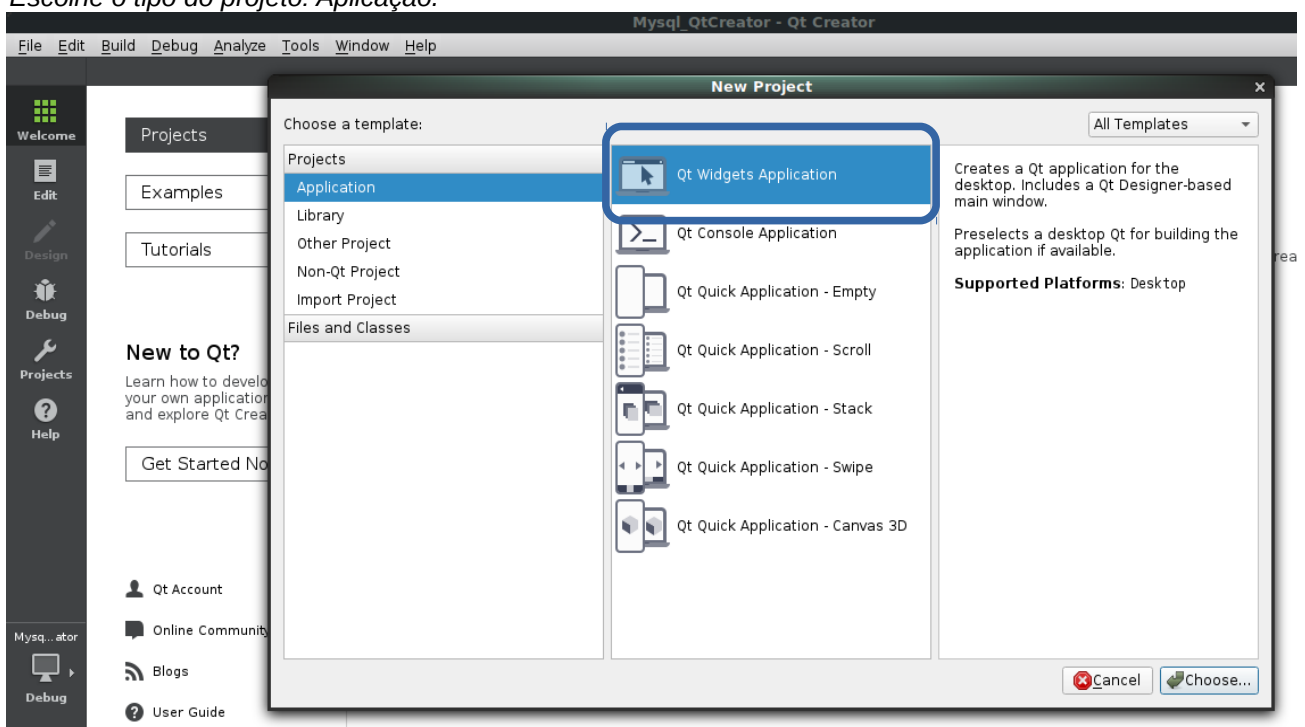
1. CREATE DATABASE Loja;
2. USE DATABASE Loja;
3. CREATE TABLE produto
(
 codigo_produto INT(11) PRIMARY KEY,
 nome_produto VARCHAR(255) NOT NULL,
 preco_produto FLOAT NOT NULL
);

2º Passo: O próximo passo é criar um novo projeto de aplicação no Qt Creator.

Inicia-se um novo projeto:



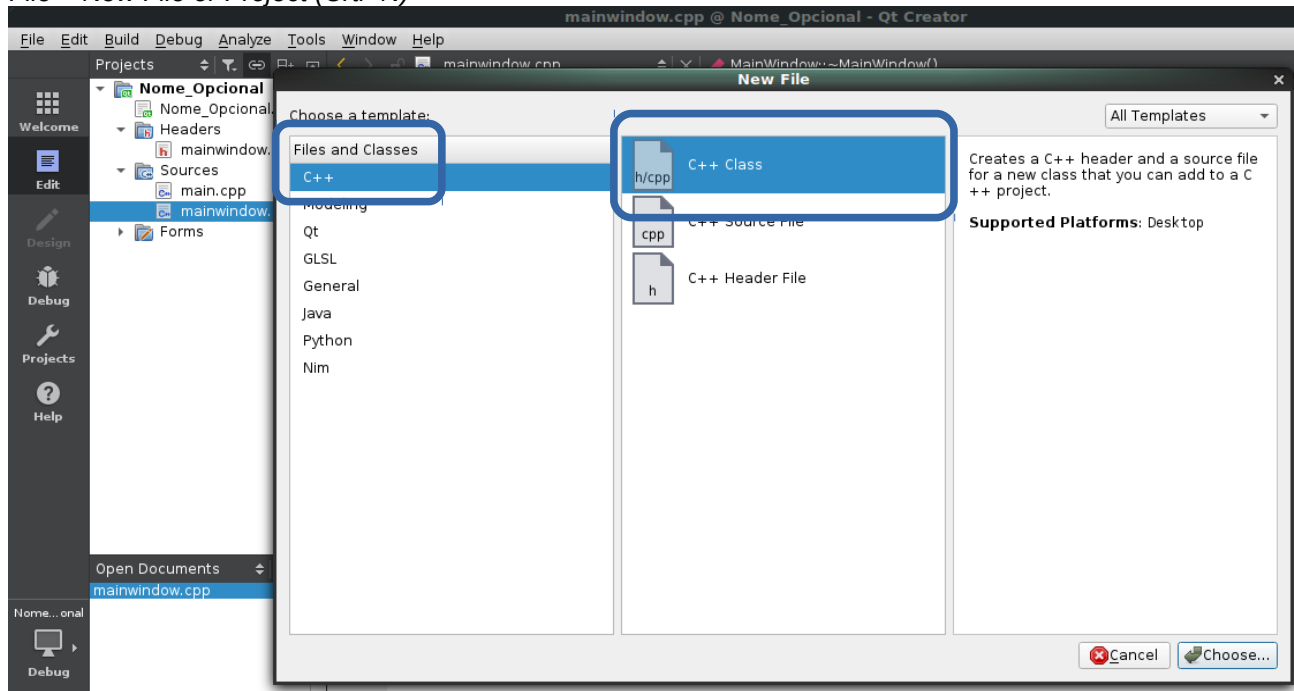
Escolhe o tipo do projeto: Aplicação.



OBS.: Após escolher essa opção, basta escolher o nome do projeto e prosseguir até finalizar a criação da nova aplicação.

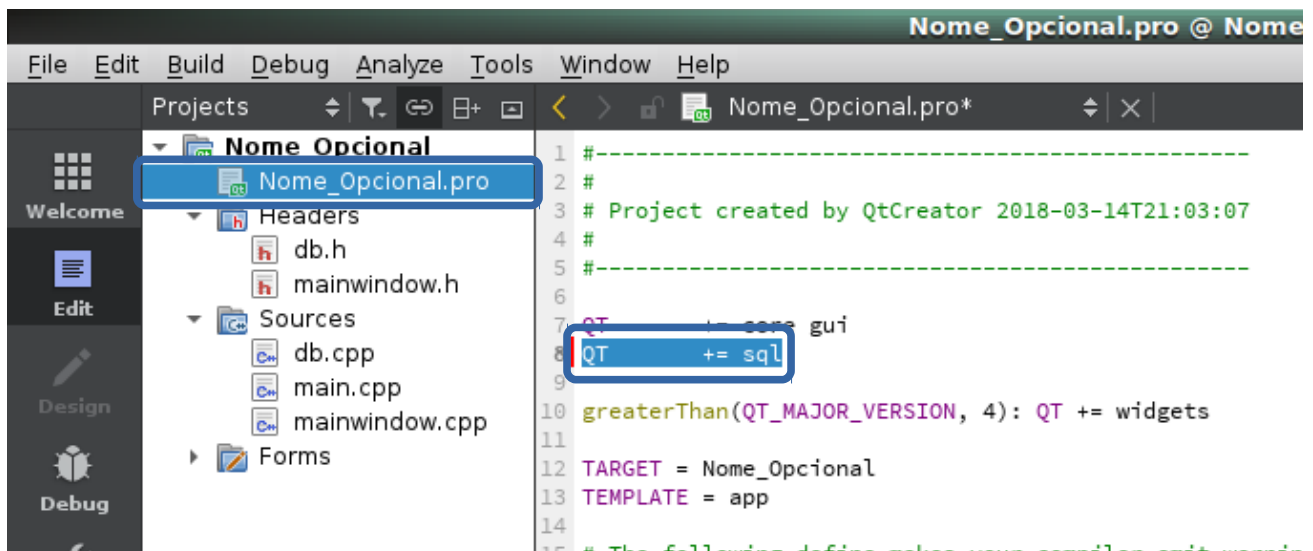
3º Passo: Criação da classe responsável pela conexão da sua aplicação ao banco de dados.

File > New File or Project (Ctrl+N)



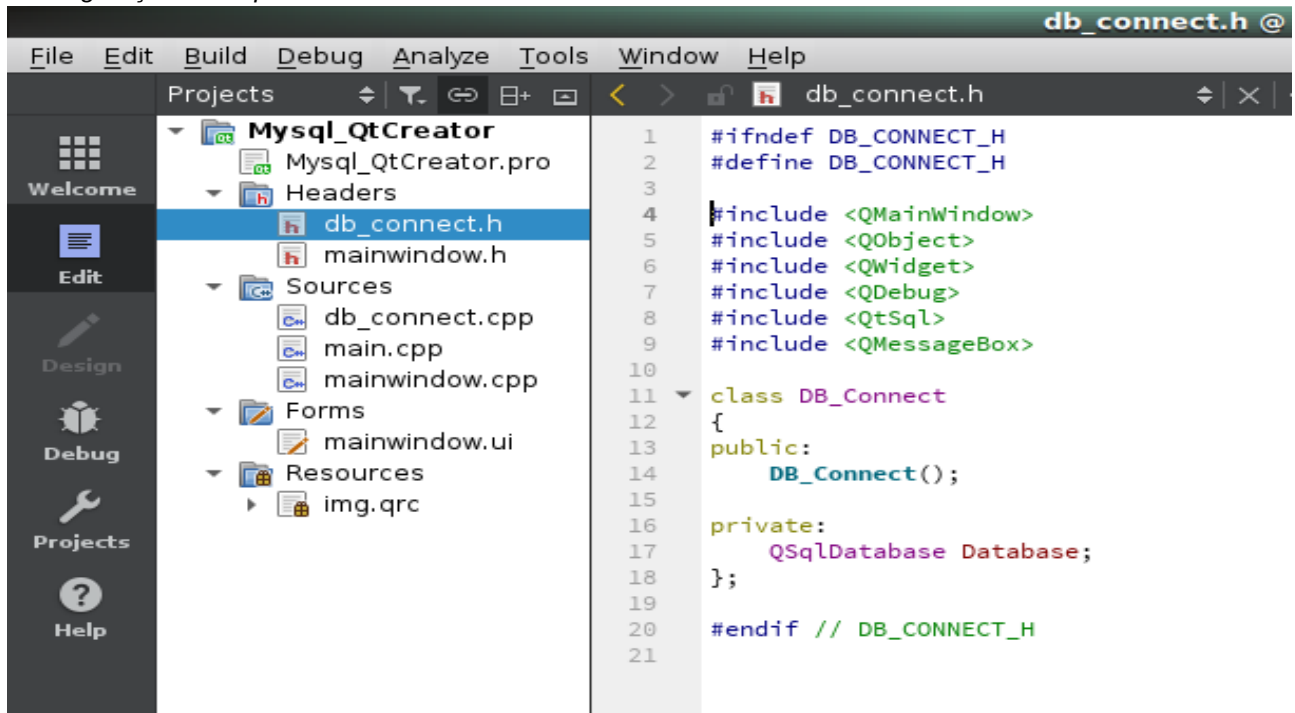
OBS.: Então é só prosseguir, escolher o nome da classe e finalizar a criação da classe.

* Após criar a classe de conexão, é necessário adicionar as dependências do SQL ao projeto.



* Após criada a classe e adicionada as dependências do SQL ao projeto, basta começar a implementação nos arquivos `.h` e `.cpp`.

* Configuração do arquivo .h



The screenshot shows the Qt Creator IDE with the 'db_connect.h' file open. The left sidebar displays the project structure for 'Mysql_QtCreator', including 'Headers' (db_connect.h, mainwindow.h) and 'Sources' (db_connect.cpp, main.cpp, mainwindow.cpp). The main editor area shows the content of 'db_connect.h'.

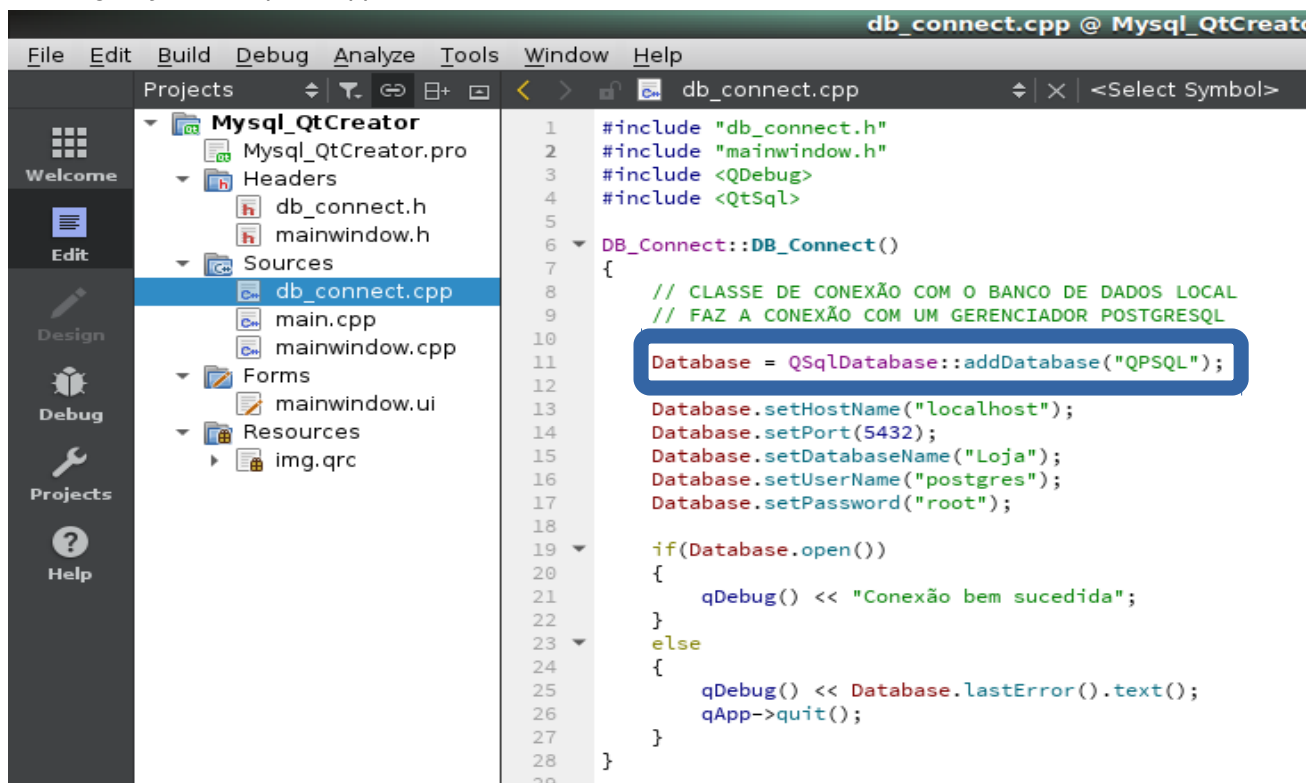
```
1  #ifndef DB_CONNECT_H
2  #define DB_CONNECT_H
3
4  #include <QMainWindow>
5  #include <QObject>
6  #include <QWidget>
7  #include <QDebug>
8  #include <QtSql>
9  #include <QMessageBox>
10
11 class DB_Connect
12 {
13 public:
14     DB_Connect();
15
16 private:
17     QSqlDatabase Database;
18 };
19
20 #endif // DB_CONNECT_H
21
```

» Inclua as bibliotecas `<Qdebug>` (Para retornar logs de atividades do sistema), `<QtSql>` (Para utilizar todas as funcionalidades do BD no Qt, como linhas de comando do PostgreSQL), `<QMessageBox>` (Para exibir janelas de alerta após realizar alguma ação).

» Crie o construtor da classe `DB_Connect()`, que será responsável por carregar os dados de conexão com o banco de dados.

» Declaração da variável do tipo `QSqlDatabase` para conexão ao BD.

* Configuração do arquivo .cpp



The screenshot shows the Qt Creator IDE with the 'db_connect.cpp' file open. The left sidebar displays the project structure for 'Mysql_QtCreator', including 'Headers' (db_connect.h, mainwindow.h) and 'Sources' (db_connect.cpp, main.cpp, mainwindow.cpp). The main editor area shows the content of 'db_connect.cpp'.

```
1  #include "db_connect.h"
2  #include "mainwindow.h"
3  #include <QDebug>
4  #include <QtSql>
5
6  DB_Connect::DB_Connect()
7  {
8      // CLASSE DE CONEXÃO COM O BANCO DE DADOS LOCAL
9      // FAZ A CONEXÃO COM UM GERENCIADOR POSTGRESQL
10
11     Database = QSqlDatabase::addDatabase("QPSQL");
12
13     Database.setHostName("localhost");
14     Database.setPort(5432);
15     Database.setDatabaseName("Loja");
16     Database.setUserName("postgres");
17     Database.setPassword("root");
18
19     if(Database.open())
20     {
21         qDebug() << "Conexão bem sucedida";
22     }
23     else
24     {
25         qDebug() << Database.lastError().text();
26         QApplication->quit();
27     }
28 }
29
```

» A variável Database, do tipo *QsqlDatabase*, é setada para adicionar uma nova base de dados, para isso recebe o drive de conexão como parâmetro (Nesse caso o drive do PostgreSQL).

» Após isso, é setada as demais informações para conexão:

HostName: Endereço do BD (Neste caso o nosso BD está local)

Port: Porta de conexão ao BD local configurado

DatabaseName: Nome do BD criado

UserName: Login para entrar no gerenciador

Password: Senha para entrar no gerenciador (Se houver)

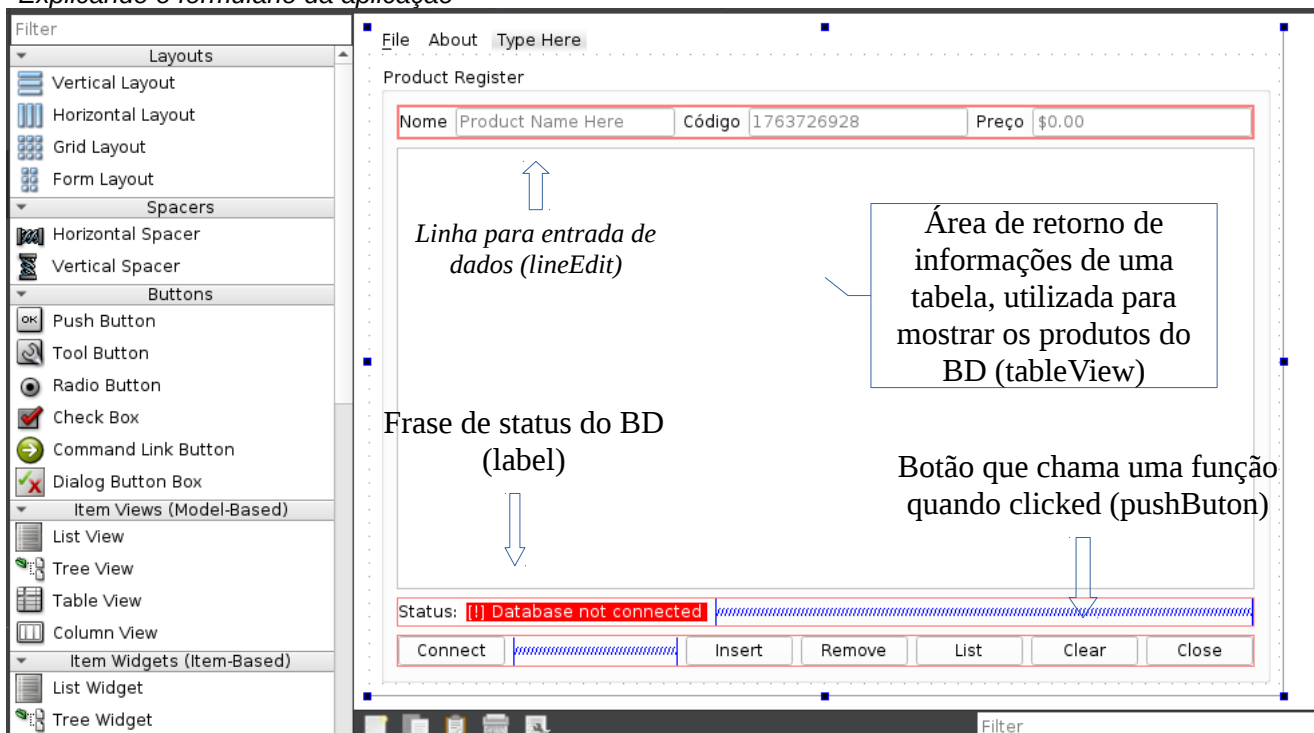
» Agora é só verificar se o BD foi aberto com a condição *Database.open()*

Caso seja, retorna um log de sistema confirmando sua abertura

Caso não seja, retorna um log detalhando o motivo do erro

4º Passo: Formatação e criação do formulário da aplicação.

Explicando o formulário da aplicação



* Cada objeto (pushButton, lineEdit, label, tableView, ...) pode ter seu texto e seu object name alterados. Para melhor identificação, os nomes foram alterados de acordo com a sua função.

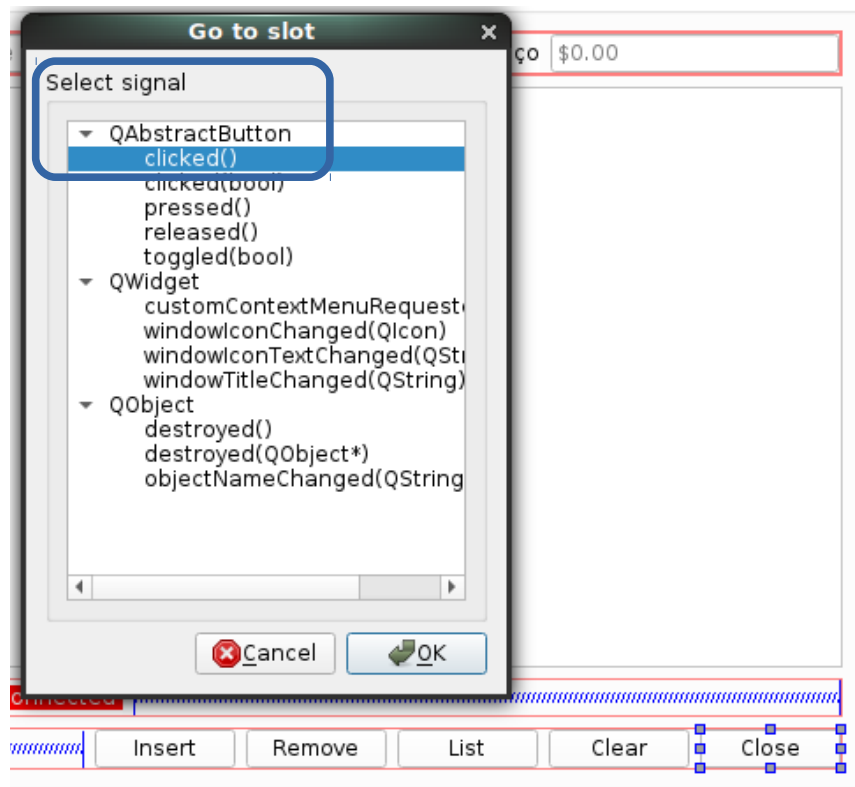
Ex.: Objeto pushButton "Close":

Text: Close; // Nome exibido para o usuário final;

Object Name: pushButton_Close;

* Para criar as funções a partir de um objeto basta clicar sobre ele com o botão direito do mouse e selecionar a opção: GO TO SLOT.

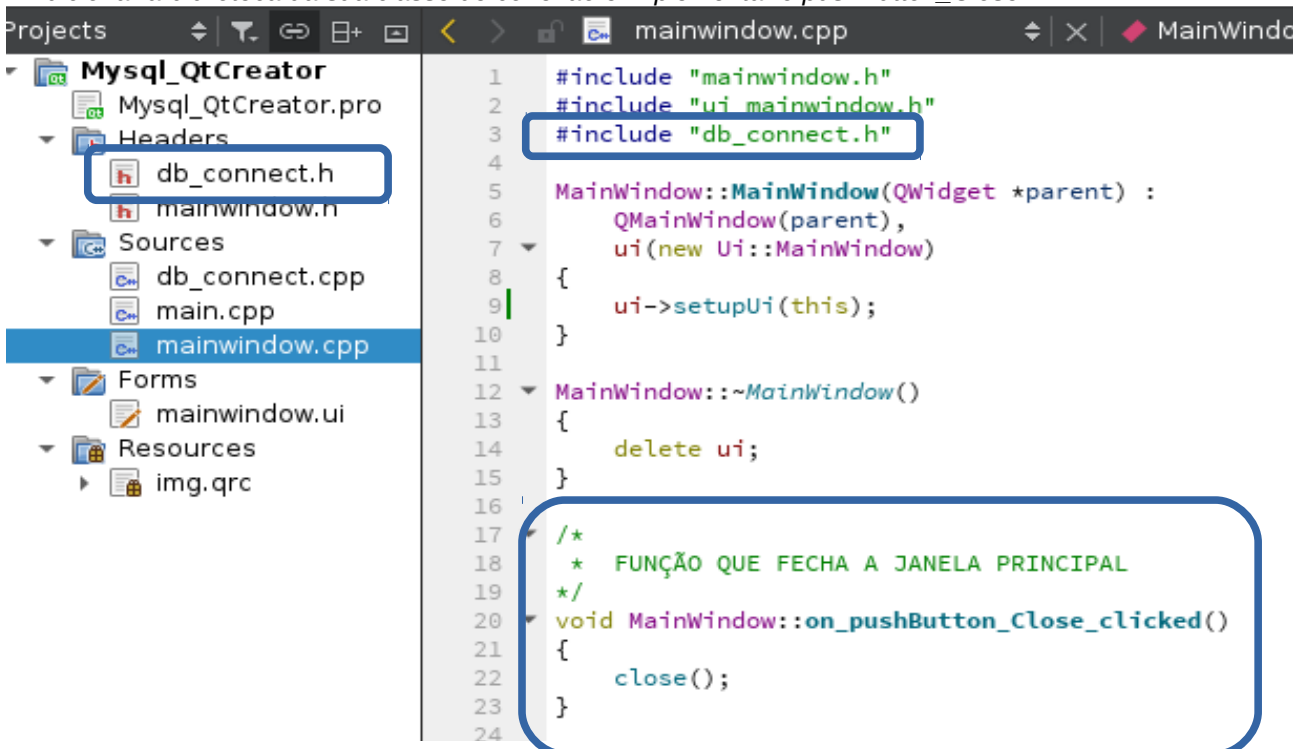
Seleciona como a função será acionada, neste caso, ao clicar no objeto.



- » OBS.: Todas as funções dos objetos pushButtons nesta aplicação são acionadas através do `Clicked()`;
- » As lineEdits e a tableView não geram funções, pois os lineEdits são métodos de entrada de dados e a tableView é um método para exibição do resultado da busca no BD.

5º Passo: Implementação dos métodos dos objetos.

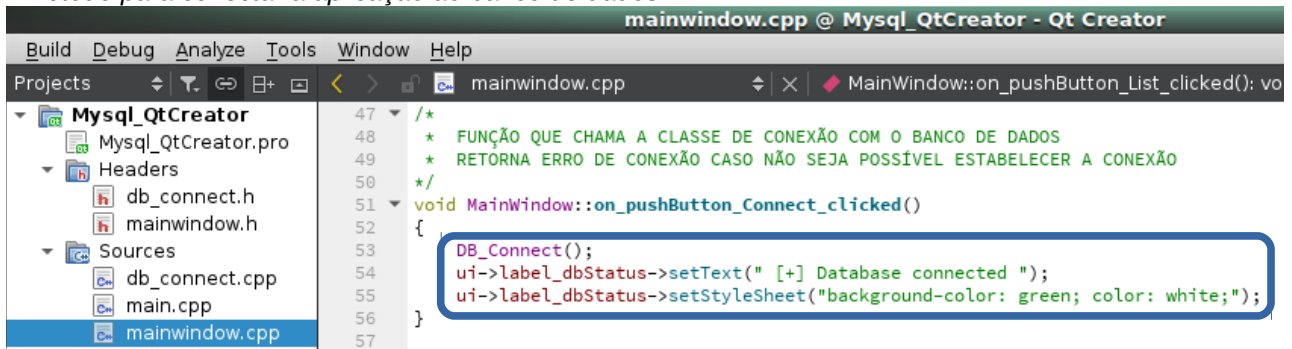
» Adicionar a biblioteca da sua classe de conexão e implementar o `pushButton_Close`



» Nesse método é implementado a função de fechar a aplicação com o comando *Close()*. Esse método é chamado quando há uma ação de clique sobre o objeto no formulário.

* Agora vamos implementar os outros métodos dos objetos.

» *Método para conectar a aplicação ao banco de dados*



```
mainwindow.cpp @ Mysql_QtCreator - Qt Creator
Build Debug Analyze Tools Window Help
Projects
Mysql_QtCreator
  Mysql_QtCreator.pro
  Headers
    db_connect.h
    mainwindow.h
  Sources
    db_connect.cpp
    main.cpp
    mainwindow.cpp
mainwindow.cpp
47  /*
48  * FUNÇÃO QUE CHAMA A CLASSE DE CONEXÃO COM O BANCO DE DADOS
49  * RETORNA ERRO DE CONEXÃO CASO NÃO SEJA POSSÍVEL ESTABELECEER A CONEXÃO
50  */
51  void MainWindow::on_pushButton_Connect_clicked()
52  {
53      DB_Connect();
54      ui->label_dbStatus->setText(" [+] Database connected ");
55      ui->label_dbStatus->setStyleSheet("background-color: green; color: white;");
56  }
57
```

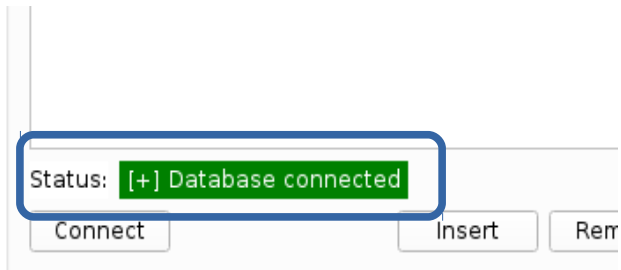
» O método chama o construtor da classe de conexão, criando assim uma nova Database à aplicação.

» *ui* é o objeto da classe *MainWindow* que aponta para os objetos do formulário, possibilitando setá-los, como nesse caso, quando a label de status do BD será setada quando houver êxito na criação da conexão com o banco de dados.

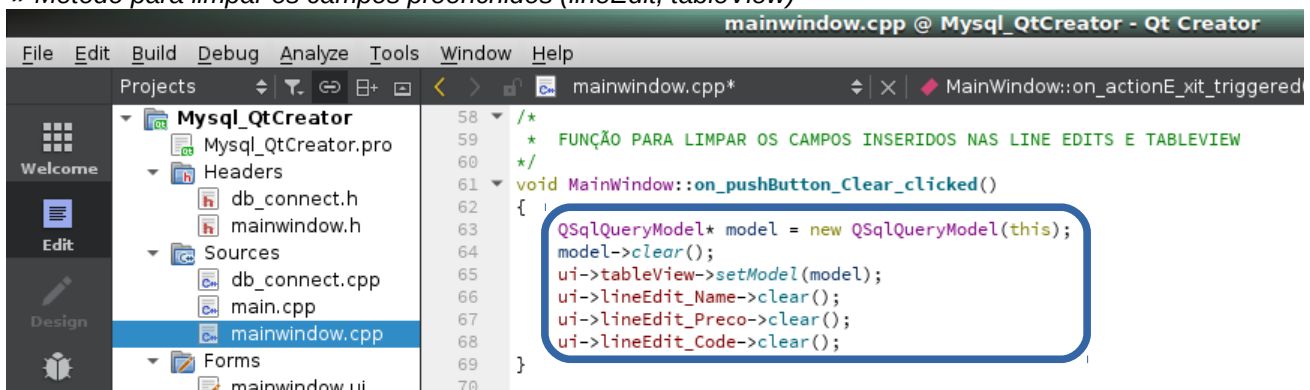
Label quando a aplicação é iniciada



Label após ser setada pelo objeto da classe



» *Método para limpar os campos preenchidos (lineEdit, tableView)*



```
mainwindow.cpp @ Mysql_QtCreator - Qt Creator
File Edit Build Debug Analyze Tools Window Help
Projects
Mysql_QtCreator
  Mysql_QtCreator.pro
  Headers
    db_connect.h
    mainwindow.h
  Sources
    db_connect.cpp
    main.cpp
    mainwindow.cpp
Forms
  mainwindow.ui
mainwindow.cpp
58  /*
59  * FUNÇÃO PARA LIMPAR OS CAMPOS INSERIDOS NAS LINE EDITS E TABLEVIEW
60  */
61  void MainWindow::on_pushButton_Clear_clicked()
62  {
63      QSqlQueryModel* model = new QSqlQueryModel(this);
64      model->clear();
65      ui->tableView->setModel(model);
66      ui->lineEdit_Name->clear();
67      ui->lineEdit_Precos->clear();
68      ui->lineEdit_Code->clear();
69  }
70
```

» *QsqlQueryModel* é uma biblioteca para criação de um modelo para realizar leitura de dados, nesse caso, uma tabela.

» O objeto da classe, *ui*, seta os valores ou funções para os objetos do formulário. Nesse método é setado a função *clear()*, para limpar todos os campos selecionados.

» Método para inserção de novos dados.



```
71  /*
72  * VALIDA PRIMEIRAMENTE SE O BANCO DE DADOS FOI ABERTO
73  * A FUNÇÃO RECEBE OS DADOS DAS LINE EDITS E OS ARMAZENA EM VARIÁVEIS
74  * PARA QUE POSSA SER ENVIADA COMO PARÂMETROS NO BANCO DE DADOS.
75  * FEITO ISSO O COMANDO É EXECUTADO E OS DADOS SÃO INSERIDOS NO DATABASE
76  */
77  void MainWindow::on_pushButton_Insert_clicked()
78  {
79      QString Name;
80      QString Preco;
81      QString Code;
82
83      Name = ui->lineEdit_Name->text();
84      Preco = ui->lineEdit_Preco->text();
85      Code = ui->lineEdit_Code->text();
86
87      QSqlQuery query;
88
89      if(query.exec("INSERT INTO produto VALUES ('+Code+', '"+Name+"', '"+Preco+'")"))
90      {
91          QMessageBox::warning(this, "Insert", "Inserção realizada com sucesso!", QMessageBox::Ok);
92          qDebug() << "Inserção realizada com sucesso!";
93      }
94      else
95      {
96          QMessageBox::critical(this, "Error", query.lastError().text(), QMessageBox::Ok);
97          qDebug() << query.lastError().text();
98      }
99  }
100
```

» Inicialmente é necessário criar as variáveis que receberão os valores contidos nas lineEdits. Feito isso, as variáveis receberão os valores dos objetos do formulário convertidos em texto, conforme a figura.


» A biblioteca *QsqlQuery* permite o programador executar comandos SQL na aplicação (Insert, Remove, Update, Drop, ...). Sendo assim, é criada uma variável desse tipo (query) para inserção dos valores no nosso BD.

» Feito isso basta verificar se o comando passado por parâmetro na nossa *query.exec* foi executado.

Caso a instrução seja executada, a biblioteca *QMessageBox* irá criar uma janela de interação com o usuário, informando que a inserção foi realizada com sucesso. Logo após é gerado um log no sistema dessa mensagem de êxito.

Caso a instrução não seja executada, a mesma biblioteca *QMessageBox* irá criar outra janela de interação com o usuário, retornando o último erro encontrado na tentativa de execução do comando SQL. Esse erro é o mesmo gerado no log do sistema pela função *qDebug()*.

» Método para listagem dos produtos



```
25  /*
26  * VALIDA PRIMEIRAMENTE SE O BANCO DE DADOS FOI ABERTO
27  * FEITO ISSO, A FUNÇÃO DE LISTAR IMPRIME EM TELA A TABELA
28  * DE TODOS OS PRODUTOS CADASTRADOS NO DATABASE.
29  */
30  void MainWindow::on_pushButton_List_clicked()
31  {
32      QSqlQueryModel* model = new QSqlQueryModel(this);
33      model->setQuery("SELECT * FROM produto");
34
35      ui->tableView->setModel(model);
36      ui->tableView->setColumnWidth(0, 207);
37      ui->tableView->setColumnWidth(1, 206);
38      ui->tableView->setColumnWidth(2, 206);
39
40      if(model->lastError().text() != " ")
41      {
42          QMessageBox::critical(this, "Error", model->lastError().text(), QMessageBox::Ok);
43          qDebug() << model->lastError().text();
44      }
45  }
46
```

» Cria uma variável do tipo `QSqlQueryModel` para receber a leitura de dados do banco. Essa variável recebe por parâmetro o comando SQL para selecionar todos os dados contidos nas colunas da tabela *produto*.

» Após isso, esses dados são setados na nossa *tableView* através do nosso objeto *ui* da classe *MainWindow*.

» Para fins de formatação da aplicação, o tamanho da colunas também foram setadas pelo objeto da classe.

» Então é feita a verificação se houve o retorno de algum erro na execução da instrução

Caso seja retornado algum erro, a biblioteca `QMessageBox` irá criar uma janela de interação com o usuário, retornando o último erro encontrado na tentativa de execução do comando SQL. Esse erro é o mesmo gerado no log do sistema pela função `QDebug()`.

» Método para remoção de dados



```
101  /*
102  * VALIDA PRIMEIRAMENTE SE O BANCO DE DADOS FOI ABERTO
103  * APÓS VALIDADO, O USUÁRIO PODE INFORMAR QUAL PRODUTO DESEJA REMOVER
104  * PASSANDO POR REFERÊNCIA O CÓDIGO DO PRODUTO
105  */
106  void MainWindow::on_pushButton_Remove_clicked()
107  {
108      QString Code;
109
110      Code = ui->lineEdit_Code->text();
111
112      QSqlQuery query;
113
114      if( query.exec("DELETE FROM produto WHERE codigo_produto = "+Code+"") )
115      {
116          QMessageBox::warning(this, "Remove", "Remoção realizada com sucesso!", QMessageBox::Ok);
117          qDebug() << "Remoção realizada com sucesso!";
118          qDebug() << query.lastError().text();
119      }
120      else
121      {
122          QMessageBox::critical(this, "Error", query.lastError().text(), QMessageBox::Ok);
123          qDebug() << query.lastError().text();
124      }
125  }
126  }
```

» A variável `Code` receberá o valor da `lineEdit_Code`. A mesma será passada como referência na linha de instrução para realizar a remoção no BD.

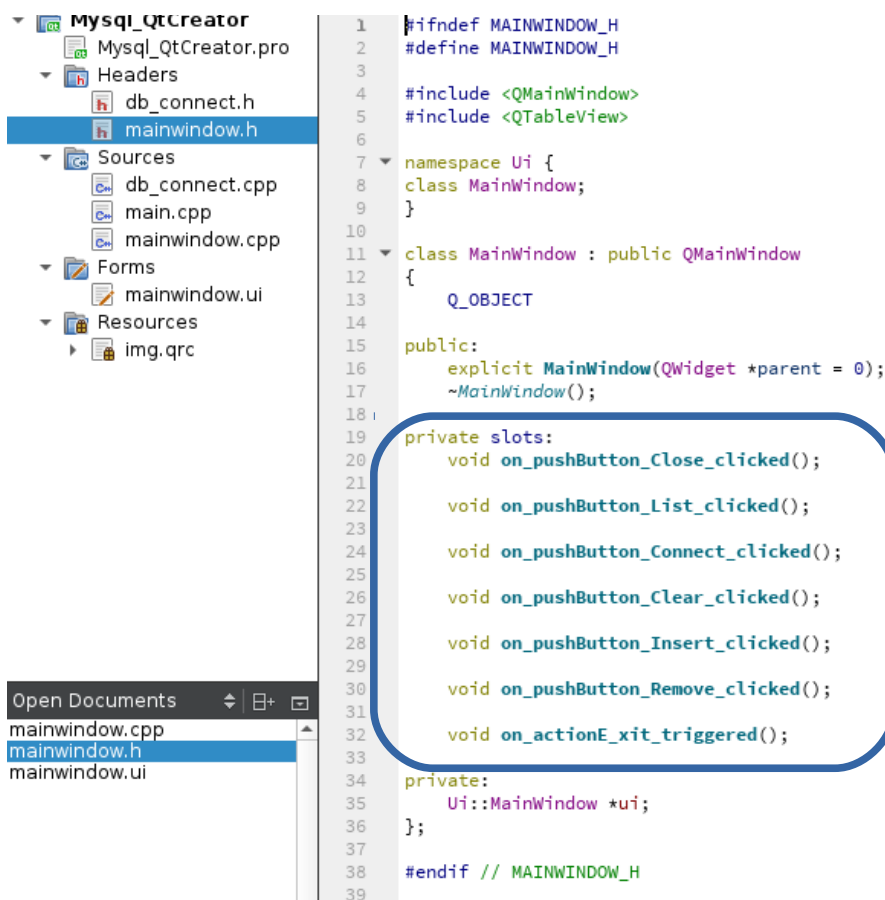
» Também é criada uma variável do tipo `QSqlQuery` (`query`) para que seja passadas as instruções de remoção de dados no BD.

» Feito isso basta verificar se o comando passado por parâmetro na nossa `query.exec` foi executado.

Caso a instrução seja executada, a biblioteca `QMessageBox` irá criar uma janela de interação com o usuário, informando que a remoção foi realizada com sucesso. Logo após é gerado um log no sistema dessa mensagem de êxito.

Caso a instrução não seja executada, a mesma biblioteca `QMessageBox` irá criar outra janela de interação com o usuário, retornando o último erro encontrado na tentativa de execução do comando SQL. Esse erro é o mesmo gerado no log do sistema pela função `QDebug()`.

* OBS.: Quando se cria o método do objeto a partir do formulário, as funções são declaradas no escopo da classe *MainWindow*.



Fim Do Tutorial

*Leonardo Emilly Dias Santos Bidó
Pontifícia Universidade Católica
Ciência da Computação
Goiânia – 2018/1*