

Engenharia de Software: Calculadora em 3 camadas - V1

Breno Keller

Universidade Federal de Ouro Preto

kellerbrenons@gmail.com

Introdução

- Trabalho prático para exemplificar o desenvolvimento de uma aplicação Cliente/Servidor utilizando APIs do QT.
- Objetivo: Evoluir a arquitetura do projeto.
- Código fonte disponível em:
<https://bitbucket.org/KellerBreno/calculadora>

Missão da Aplicação

Evitar que usuários não habilitados tenham acesso ao sistema. E permitir a realização de operações matemáticas simples por múltiplos usuários de forma simultânea em diferentes computadores.

Arquitetura

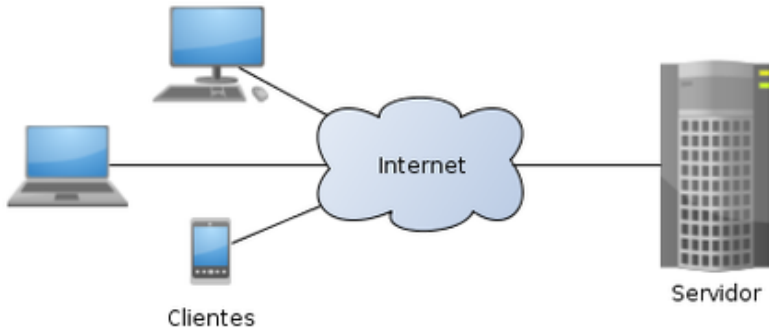


Figura: Arquitetura da Aplicação

Componente: Client

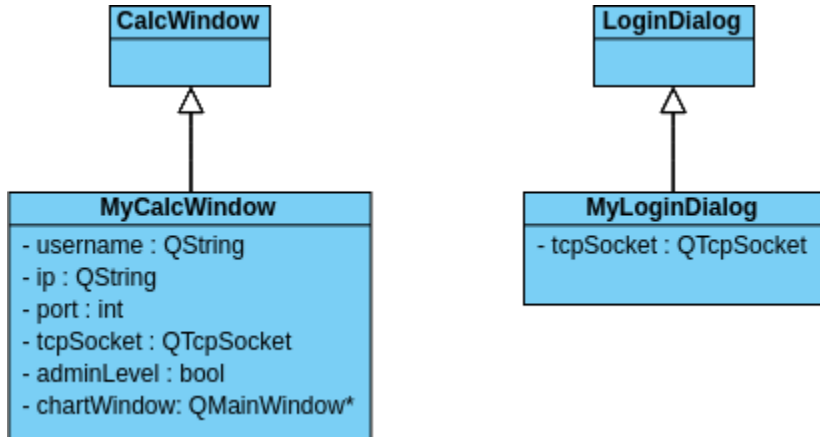


Figura: Diagrama de Classes: Client

Componente: Server

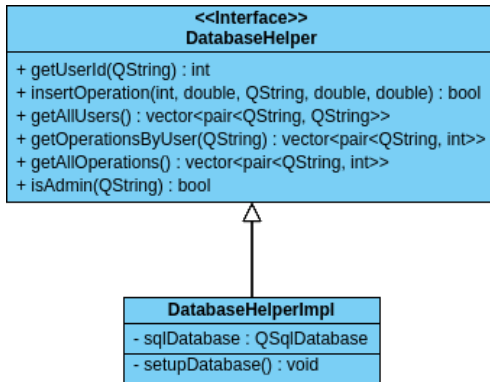


Figura: Diagrama de Classes: DatabaseHelper

Componente: Server

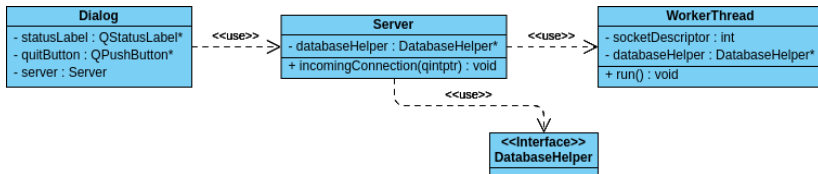


Figura: Diagrama de Classes: Server

Diagrama de Sequência

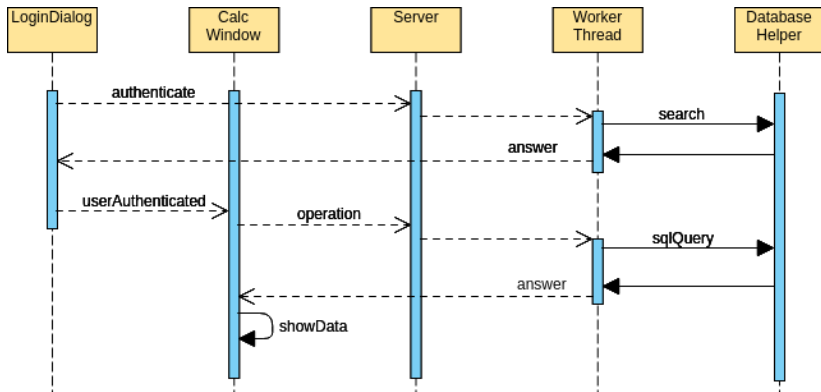


Figura: Diagrama de Sequência

Comunicação

Para realizar a comunicação entres os componentes foi criado um protocolo de mensagens baseado em JSON.

- 4 tipos de solicitação
 - Login;
 - Calcular;
 - Listar operações de um usuários;
 - Listar operações de todos usuários;
- 4 tipos de respostas
 - Login válido e se é administrador;
 - Resultado;
 - Quantidade de operações realizadas;
 - Erro.

Comunicação - Usuário Comum

```
// Client
{
    "operationType": 1,
    "username": "breno",
    "password": "keller"
}

// Server
{
    "valid" : true,
    "adminLevel" : false
}
```

Comunicação - Usuário Administrador

```
// Client
{
    "operationType": 1,
    "username": "admin",
    "password": "admin"
}

// Server
{
    "valid" : true,
    "adminLevel" : true
}
```

Comunicação - Usuário Não Cadastrado

```
// Client
{
    "operationType": 1,
    "username": "joão",
    "password": "maria"
}

// Server
{
    "valid" : false
}
```

Comunicação - Realizar Cálculo

```
// Client
{
    "operationType": 2,
    "username": "breno",
    "v1": 1,
    "v2": 2,
    "opCode": 4
}
```

```
// Server
{
    "answerType": 1,
    "result": 0.5
}
```

Comunicação - Listar Operações do Usuário

```
// Client
{
    "operationType": 3,
    "username": "breno"
}

// Server
{
    "answerType": 2,
    "Adição": 5,
    "Subtração": 5,
    "Divisão": 5,
    "Multiplicação": 5
}
```

Comunicação - Listar Operações de Todos os Usuários

```
// Client
{
    "operationType": 4,
    "username": "admin"
}
```

```
// Server
{
    "answerType": 3,
    "Adição": 15,
    "Subtração": 15,
    "Divisão": 15,
    "Multiplicação": 15
}
```

Comunicação - Operação Inválida

```
// Client
{
    "operationType": 4,
    "username": "breno"
}
```

```
// Server
{
    "answerType": 4
}
```


Lógica de Negócio em Client/Server

Onde inserir a lógica de negócio?

Três alternativas:

- Lógica no Server;
- Lógica no Client;
- Lógica dividida entre Client/Server.

Lógica no Server

Vantagens:

- Segurança;
- Controle de acesso a recursos;
- Equipamento mais "potente".

Desvantagens:

- Client é somente uma "interface";
- Server não escala bem;
- Gargalo.

Lógica no Client

Vantagens:

- Reduz a demanda no server.

Desvantagens:

- Atualização é obrigatória em todos os clientes;
- Segurança;
- Equipamento menos "potente".

Lógica dividida entre Client/Server

Vantagens:

- Utiliza o melhor os recursos de cada equipamento.

Desvantagens:

- Complexidade;
- Custo.

APIs Utilizadas

Para a implementação do projeto foram utilizadas as seguintes APIs:

- QT SQL, API para banco de dados;
- QT Thread, API para multiprocessamento;
- QT Network, API para operações de rede;
- QT Widgets, API para interfaces gráficas;
- QT Charts, API para geração de gráficos.

QVariant

- Atua como uma união dos tipos de dados básicos do QT.
- Disponível no core do QT.
- Utilizado para manipulação de propriedade de QObject e trabalho com bancos de dados.

Utilizando Json

Classes Principais

- **QJsonObject:** Classe para manipular um objeto JSON.
- **QJsonDocument:** Classe para manipular JSON por meio de escrita e leitura.

QJsonObject

// Criar um JSON e inserir seus valores

```
QJsonObject jsonObject;  
jsonObject.insert("operationType", 2);  
jsonObject.insert("username", username);  
jsonObject.insert("v1", parcela1);  
jsonObject.insert("opCode", opCode);  
jsonObject.insert("v2", parcela2);
```

// Recuperar um valor

```
double resultado = jsonObject.value("result").toDouble();
```


QJsonDocument

```
// Criar um objeto JSON por meio de uma String
QJsonDocument jsonDocument =
    QJsonDocument::fromJson(message.toUtf8());
QJsonObject jsonObject = jsonDocument.object();

// Criar uma string equivalente ao JSON
QJsonDocument answerDocument(answerResult);
QString answerString(answerDocument.
    toJson(QJsonDocument::Compact));
```

QT SQL

API do QT para o uso de bancos de dados, dividido em 3 camadas:

- Camada do Driver;
- Camada da API SQL;
- Camada da interface de Usuário.

Para utilizar necessário adicionar: 'QT += sql' no .pro

Drivers

O QT utiliza drivers para traduzir as ações da API QT para banco.

Tabela: Driver QT SQL

Driver Name	Banco de Dados
QDB2	IBM DB2 (versão 7.1 e acima)
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Oracle Call Interface Driver
QODBC	Open Database Connectivity (ODBC)
	Microsoft SQL Server ou outros ODBC
QPSQL	PostgreSQL (versão 7.3 e acima)
QSQLITE2	SQLite versão 2
QSQLITE	SQLite versão 3
QTDS	Sybase Adaptive Server

QSQL

Classes Principais

- **QSqlDatabase:** Classe que gerencia a conexão com o banco de dados por meio dos drivers do QT.
- **QSqlQuery:** Fornece recursos para manipular instruções SQL, como SELECTs e INSERTs.

Utilizando QSqlDatabase

```
// Habilita o driver que será utilizado e  
// informa o endereço do banco de dados  
QSqlDatabase sqlDatabase;  
sqlDatabase = QSqlDatabase::addDatabase("QSQLITE");  
sqlDatabase.setDatabaseName("calc_example.sqlite");  
  
// Abre uma conexão com o banco, retorna falso caso falhe  
sqlDatabase.open();  
  
// Fecha uma conexão com o banco de dados  
sqlDatabase.close();
```

Utilizando QSqlQuery com Select

```
// Query SELECT usando concatenação
QSqlQuery select("SELECT id FROM users WHERE login = '" +
                username + "'", sqlDatabase);

// Move para o primeiro elemento válido da busca
while(select.next()){
    // Recebe o valor do id do usuário
    int userId = select.value(0).toInt();
}
```

Utilizando QSqlQuery com Insert

```
// Query INSERT utilizando placeholders
QSqlQuery insert;
insert.prepare("INSERT INTO operations (user_id, "
               "operando1, operador, operando2, "
               "resultado) VALUES (:user, :par1, :op, "
               ":par2, :resultado)");
insert.bindValue(":user", userId);
insert.bindValue(":par1", v1);
insert.bindValue(":op", operacao);
insert.bindValue(":par2", v2);
insert.bindValue(":resultado", resultado);
// Retorna se foi possível executar a query
bool b = insert.exec();
```

QT Thread

API do QT para o uso de threads, já utilizado pelos widgets para se ter processamento assíncrono. Disponível no core do QT.

- **QThread:** Classe que permite a criação e manipulação de threads de forma independente ao sistema operacional utilizado.

Herdando de QThread

```
// Criação de thread customizada onde o processamento é  
// delegado a função run  
class WorkerThread : public QThread {  
    Q_OBJECT  
  
public:  
    WorkerThread(QObject *parent);  
    void run() override;  
};
```

Instanciando uma QThread customizada

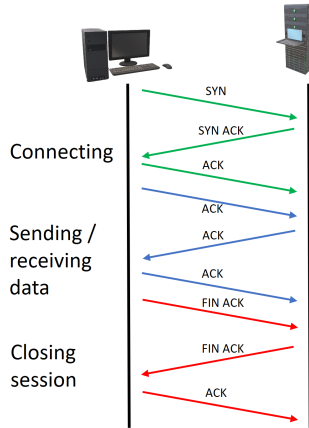
```
// Criar uma thread para realizar um trabalho  
WorkerThread *thread = new WorkerThread(this);  
  
// Marca seu signal de finalização para apagar os dados  
connect(thread, SIGNAL(finished()),  
         thread, SLOT(deleteLater()));  
  
// Executa a thread  
thread->start();
```

QT Network

API do QT para o uso de comunicação TCP/IP por meio da rede.
Essa API trabalha com requisições, cookies e operações HTTP.

Para utilizar necessário adicionar: 'QT += network' no .pro

TCP/IP



QT Network

Classes principais **QTcpServer**: Classe que cria um servidor baseado em TCP. **QTcpSocket**: Classe que cria um socket baseado em TCP.

Criando um QTcpServer customizado

```
// Exemplo de Herança de QTcpSocket  
class Server : public QTcpServer{  
    Q_OBJECT  
  
public:  
    Server(QObject *parent = 0);  
  
protected:  
    // Sobrecarregado para criar uma thread  
    // que lida com a conexão  
    void incomingConnection(qintptr socketDescriptor)  
        override;  
};
```

Conectando e enviando com QTcpSocket

// Utilização no client

```
QTcpSocket tcpSocket;  
tcpSocket.connectToHost(ip, port);
```

// Envio de dados

```
QByteArray jsonData = jsonString.toUtf8();  
tcpSocket.write(jsonData);
```

Recebendo uma conexão com QTcpSocket

```
// Utilização no server
```

```
QTcpSocket tcpSocket;
```

```
// Configurar o socket
```

```
tcpSocket.setSocketDescriptor(socketDescriptor);
```

```
// Recebe um array de bytes da rede
```

```
QByteArray data = tcpSocket.readLine();
```


QT Widgets

API para criação de elementos de interface gráfica para desktops.

Para utilizar necessário adicionar: 'QT += widgets' no .pro

QT Widgets

Classes Principais

- **QWidget:** Classe base para componentes de interface de usuário.
- **QMainWindow:** Classe para criar uma janela principal de aplicação.

QWidget

```
class MyLoginDialog : public QWidget,  
                      private Ui::LoginDialog {  
    Q_OBJECT  
  
public:  
    MyLoginDialog(QWidget *parent = nullptr);  
};
```

QMainWindow

```
class MyCalcWindow : public QMainWindow,  
                      private Ui::CalcWindow{  
    Q_OBJECT  
  
public:  
    MyCalcWindow(QWidget *parent = nullptr);  
};
```

QT Charts

API do QT para criação de gráficos simples. Os gráficos podem ser utilizados como QWidgets, portanto integrados a interfaces.

Para utilizar necessário adicionar: 'QT += charts' no .pro

QChartView

// Conjunto de dados para o gráfico

```
QPieSeries *series = new QPieSeries();  
series->append("Teste1", 15);  
series->append("Teste2", 25);
```

// Grafico

```
QChart *chart = new QChart();  
chart->addSeries(series);  
chart->setTitle("Teste Chart");
```

// Widget para exibição de gráficos

```
QChartView *chartView = new QChartView(chart);  
chartView->setRenderHint(QPainter::Antialiasing);
```

Fim?