Revista QT

Edição 2 – Novembro/Dezembro - 2010

Agora com 62 Páginas!

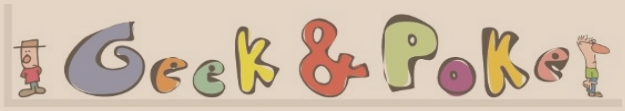
Opera House- Oslo

Aplicações Híbridas Instalação do QT para Mac Tutorial QML

Distribuindo apliações em Qt para Linux

revistaqt.com

Índice



Caríssimos leitores,

Há uns três meses, quando resolvi criar uma revista sobre o Qt eu já sabia que seria não seria nada fácil.
Pra começar esta seria a minha primeira experiência como "editor" e pra piorar eu estava sozinho nessa empreitada.

Com o tempo que sobrava de um emprego em tempo integral e uma família numerosa, eu ia escrevendo e como também era a minha primeira experiência como diagramador, ia apanhando do BrOffice Draw (programa que uso para editar a revista). Do meu entusiasmo com o Qt vinha a estímulo de que eu precisava para continuar.

Com apenas 24 páginas, a primeira edição da Revista Qt foi publicada no dia 8 de setembro deste ano. Eram poucos e um tanto inseguros, mas eram os meus primeiros passos no sentido da minha ideia.

Dois meses e muito trabalho depois, aqui estou eu de novo, desta vez escrevendo o editorial da segunda edição da Revista Qt, que passa a ter editorias definidas, uma diagramação ligeiramente melhorada - continua sendo feita por mim :) - e com o mesmo objetivo: compartilhar conhecimento sobre Qt.

Seguindo orientações do meu amigo Pierre Freire, criei editorias, de acordo com a natureza de cada artigo a ser publicado.

Estas serão as primeiras editorias da Revista Qt:

Iniciar

Esta editoria será dedicada a artigos para iniciantes em Qt. Se já existisse na primeira edição, os artigos: "Apresentando o Qt", "Instalação do Qt SDK" e "Alô Qt Creator" estariam nesta editoria, por requererem um conhecimento muito básico do Qt.

Laboratório

Artigos com um nível mais alto de complexidade ou que exijam maiores conhecimentos dos leitores serão publicados nesta editoria. O artigo "Qt + PHP – parte 1" da primeira edição é um exemplo de artigo desta editoria.

Versão Brasileira

É fato: existe pouca documentação sobre Qt em português disponível. No entanto, existe um bom material disponível em inglês. Nesta editoria serão publicadas traduções da documentação do Qt.

Caixa de Entrada

Nesta editoria serão publicados os e-mails dos leitores, enviados para revistaqt@gmail.com, com as respectivas respostas dadas pelo editor.

Notícias

Sendo uma publicação bimestral, se fosse publicar tudo o que acontece precisaríamos de uma centena de páginas para notícias. Então, apenas notícias relevantes para estudantes ou profissionais de Qt serão consideradas para esta editoria

Geek & Poke

As charges de Oliver Widder ganham uma seção específica na revista.

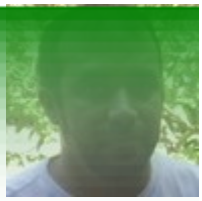
Além destas, existem planos para outras editorias, como uma que apresente *cases* de aplicação do Qt.

Encerrando este tutorial, agradeço pelas críticas e mensagens de apoio que tenho recebido desde o lançamento da primeira edição da revista.

Um grande abraço.

André Luiz de Oliveira Vasconcelos
editor

Qt + PHP – parte 2



Por: André Vasconcelos
alovasconcelos@gmail.com

Continuação do tutorial sobre desenvolvimento de aplicações híbridas Desktop + PHP

Na primeira parte deste tutorial, vimos um pequeno exemplo de aplicação híbrida Desktop + PHP. A partir de agora passamos a ver um exemplo mais complexo, utilizando o Zend Framework no servidor. Como os tutoriais publicados nesta seção da revista – Laboratório – serão voltados aos programadores mais experientes não vamos nos deter em aspectos básicos de programação Qt ou PHP.

Para esta segunda parte, precisaremos do Zend Framework, que pode ser obtido no endereço:

http://framework.zend.com/download/latest

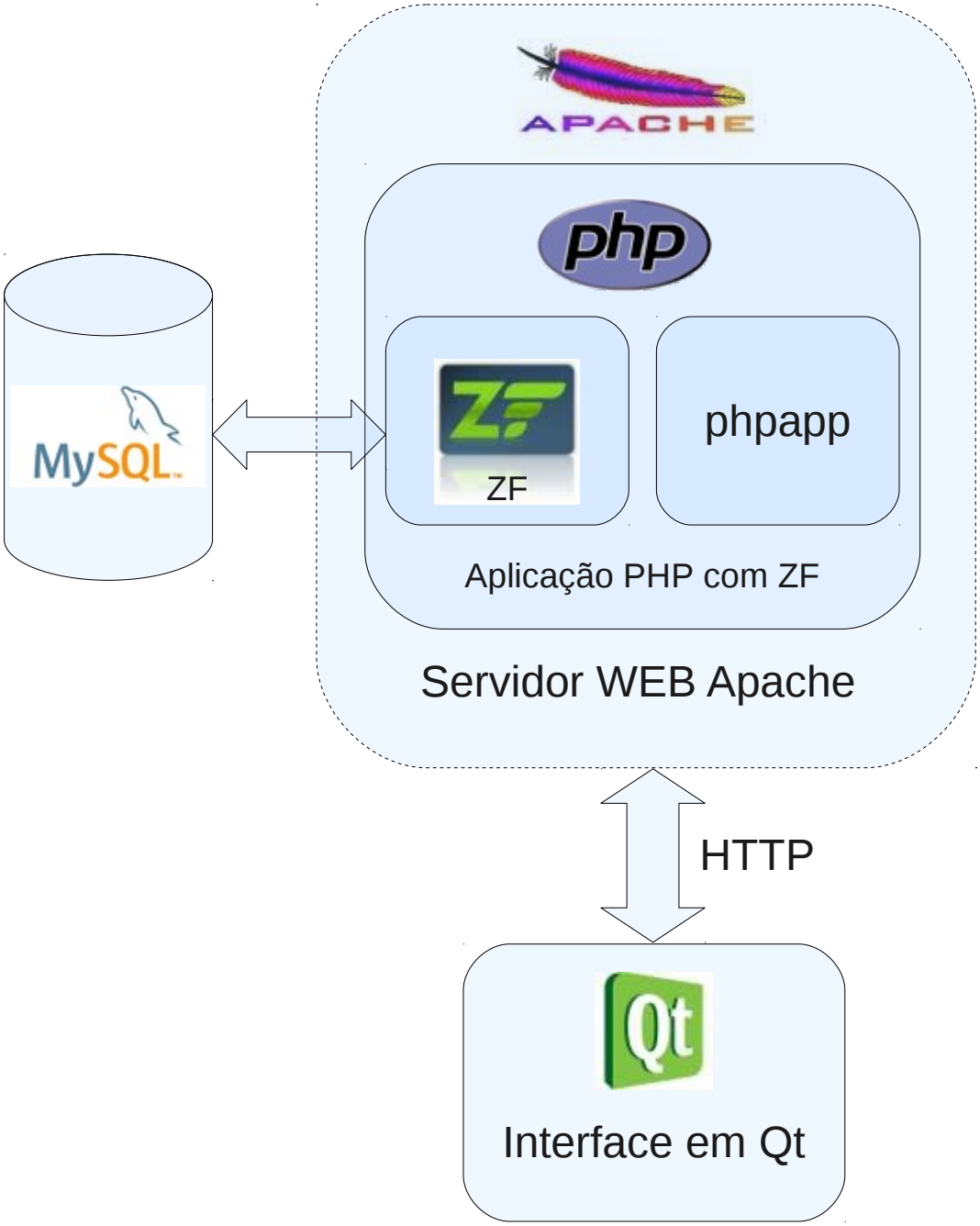
O motivo de adotar o Zend Framework neste projeto é apenas um: simplificar o desenvolvimento do lado servidor da nossa aplicação. Isto porque o ZF (vamos chamá-lo assim daqui pra frente) possui uma série de classes prontas para usar . Como usaremos apenas alguns componentes básicos do ZF, podemos utilizar a versão *minimal*.
Além do ZF, vamos precisar do MySQL instalado na máquina que vamos usar como servidor. O MySQL pode ser obtido no endereço:

http://dev.mysql.com/downloads/

Obviamente, como a parte servidora de nossa aplicação será desenvolvida em PHP, precisamos também do Apache e do PHP 5 instalados no servidor.

A minha plataforma de desenvolvimento é a seguinte:

- Ubuntu 10.10
- Apache2
- PHP5
- MySQL 5.1.49-1
- Zend Framework 1.10.8-minimal



A aplicação desenvolvida nesta parte do tutorial carrega dados a partir de um servidor e mostra em um *Grid*. Os dados ficam em um banco MySQL e são lidos por uma aplicação em PHP disponível no servidor.

A interface da nossa aplicação – feita em Qt – faz uma requisição à aplicação em PHP no servidor, e apresenta o resultado. Teremos um botão para fazer nova requisição ao servidor e atualizar as informações no *Grid*.

Do lado servidor temos um programa em PHP com ZF que recebe o nome de uma classe e o nome de um método, executa o método e retorna o resultado como um XML.

Como diria Jack, o Estripador: “vamos por partes”. Comecemos pela criação do banco de dados da aplicação:

```
CREATE DATABASE `teste` DEFAULT CHARACTER SET latin1 COLLATE latin1_swedish_ci;
```

Com o banco de dados criado vamos à criação da tabela **estado**:

```
CREATE TABLE `estado` (
  `id_estado` int(11) NOT NULL AUTO_INCREMENT,
  `sigla` varchar(2) NOT NULL,
  `nome` varchar(40) NOT NULL,
  PRIMARY KEY (`id_estado`),
  UNIQUE KEY `estado_sigla` (`sigla`),
  UNIQUE KEY `estado_nome` (`nome`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=27 ;
```

Os comandos a seguir populam a tabela com a lista de estados e o Distrito Federal:

```
insert into estado (id_estado,nome,sigla) values (1, 'Amazonas', 'AM');
insert into estado (id_estado,nome,sigla) values (2, 'Acre', 'AC');
insert into estado (id_estado,nome,sigla) values (3, 'Rondonia', 'RO');
insert into estado (id_estado,nome,sigla) values (4, 'Roraima','RR');
insert into estado (id_estado,nome,sigla) values (5, 'Pará', 'PA');
insert into estado (id_estado,nome,sigla) values (6, 'Maranhão', 'MA');
insert into estado (id_estado,nome,sigla) values (7, 'Piauí', 'PI');
insert into estado (id_estado,nome,sigla) values (8, 'Ceará', 'CE');
insert into estado (id_estado,nome,sigla) values (9, 'Rio Grande do Norte', 'RN');
insert into estado (id_estado,nome,sigla) values (10, 'Paraíba', 'PB');
insert into estado (id_estado,nome,sigla) values (11, 'Sergipe', 'SE');
insert into estado (id_estado,nome,sigla) values (12, 'Alagoas','AL');
insert into estado (id_estado,nome,sigla) values (13, 'Bahia', 'BA');
insert into estado (id_estado,nome,sigla) values (14, 'Tocantins', 'TO');
insert into estado (id_estado,nome,sigla) values (15, 'Goiás', 'GO');
insert into estado (id_estado,nome,sigla) values (16, 'Distrito Federal', 'DF');
insert into estado (id_estado,nome,sigla) values (17, 'Espírito Santo', 'ES');
insert into estado (id_estado,nome,sigla) values (18, 'Minas Gerais', 'MG');
insert into estado (id_estado,nome,sigla) values (19, 'Rio de Janeiro', 'RJ');
insert into estado (id_estado,nome,sigla) values (20, 'São Paulo', 'SP');
insert into estado (id_estado,nome,sigla) values (21, 'Paraná', 'PR');
insert into estado (id_estado,nome,sigla) values (22, 'Santa Catarina', 'SC');
insert into estado (id_estado,nome,sigla) values (23, 'Rio Grande do Sul', 'RS');
insert into estado (id_estado,nome,sigla) values (24, 'Mato Grosso do Sul', 'MS');
insert into estado (id_estado,nome,sigla) values (25, 'Mato Grosso', 'MT');
insert into estado (id_estado,nome,sigla) values (26, 'Amapá', 'AP');
```

Agora que temos o banco de dados vamos à criação do lado servidor da nossa aplicação em PHP. O **documentRoot** do Apache em minha máquina aponta para o diretório /home/vasconcelos/Projetos/www. Se você não faz ideia do que seja **documentRoot**, recomendo que procure na Internet por um tutorial de instalação e configuração do Apache.

Estando no diretório correspondente ao **documentRoot** do Apache, crie um diretório chamado phpapp:

```
$ mkdir phpapp
```

A seguir vamos criar dentro do diretório de nossa aplicação um diretório chamado Classes, onde ficarão os arquivos com os códigos das classes da mesma:

```
$ cd phpapp

$ mkdir Classes
```

Como esta aplicação fará uso do ZF, vamos criar em seu diretório um link simbólico para o diretório contendo as bibliotecas do cara (o ZF).

```
$ ln -s ../ZendFramework-1.10.8-minimal/library/Zend Zend
```

No meu caso, o diretório contendo o ZF é **ZendFramework-1.10.8-minimal** e fica no mesmo nível do diretório da aplicação. Substitua a referência na criação do link de acordo com o nome do diretório e a localização do ZF no seu servidor.

Agora que temos o diretório de nossa aplicação e um link para o diretório com as bilbiotecas do ZF criados, vamos escrever o código PHP para ela, começando pelo arquivo index.php, que deverá ser criado no diretório da aplicação – **phpapp**.

```php
<?php
// index.php

// Include para o Servidor Rest do Zend Framework
require_once('Zend/Rest/Server.php');

// Nome da classe que está sendo requisitada
$className = $_GET['class'];

// Include para definição da classe requisitada
require_once("Classes/{$className}.php");

// Instancia servidor Rest
$server = new Zend_Rest_Server();

// Seta o nome da classe
$server->setClass($className);

// Processa a requisição
$server->handle();
```

No quadro acima temos todo o código do arquivo index.php. Como mencionei no início desta parte do tutorial, a adoção do ZF simplificou muito o trabalho. A classe Zend_Rest_Server faz toda a "mágica" acontecer. O Rest Server recebe uma requisição para execução de um método em uma determinada classe e retorna o resultado.

Nosso próximo passo será criar uma classe para conexão ao banco de dados. Antes de continuarmos, quero fazer algumas observações. A finalidade deste exemplo é demonstrar a abordagem de aplicações híbridas (Qt + PHP), não servir de base para uma aplicação "real". Aqui não estou preocupado com o tratamento de erros, por exemplo. Observe que no **index.php** não está sendo tratada a situação de inexistência do arquivo correspondente à classe que esteja sendo requisitada ao servidor.

A classe de conexão que vamos criar a seguir também não possuirá tratamento de erros e servirá apenas para executar *queries*.

Crie um arquivo chamado Conexao.php no diretório Classes da aplicação.

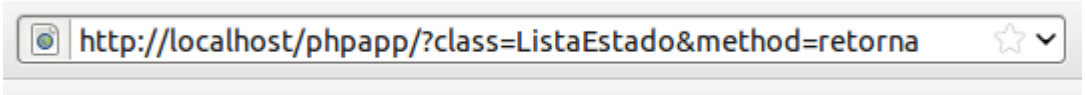
```php
<?php
// Classes/Conexao.php

require_once('Zend/Db.php');

// Definição da classe de conexão com o banco de dados
class Conexao {
    private $db;               // Recurso de conexão com o banco de dados
    static private $instancia; // Instância estática de Conexao

    // Método público para conectar ao banco de dados
    public function conecta()
    {
      // Conecta ao banco de dados
        $db = Zend_Db::factory('Pdo_Mysql', array(
            'host'     => 'localhost',  // servidor
            'username' => 'root',         // usuário
            'password' => 'margrande',    // senha
            'dbname'   => 'teste',        // banco de dados
            'charset'  => 'utf8'          // codificação
        ));
        $this->db = $db;
    }

    // Método estático para retornar ou instanciar uma nova conexão
    static public function getConexao(){
        if (!isset(self::$instancia)) {
            $c = __CLASS__;
            self::$instancia = new $c;
        }
        return self::$instancia;
    }

    // Método para execução de Queries SQL
    public function executaQuery($sql)
    {
         if(!is_resource($this->db)){
              $this->conecta();
         }
         return $this->db->fetchAll($sql);
    }
}
```

Esta classe de conexão usa o *Design Pattern conhecido* como *Singleton*. Mais informações sobre o assunto podem ser encontradas em **http://pt.wikipedia.org/wiki/Singleton**.

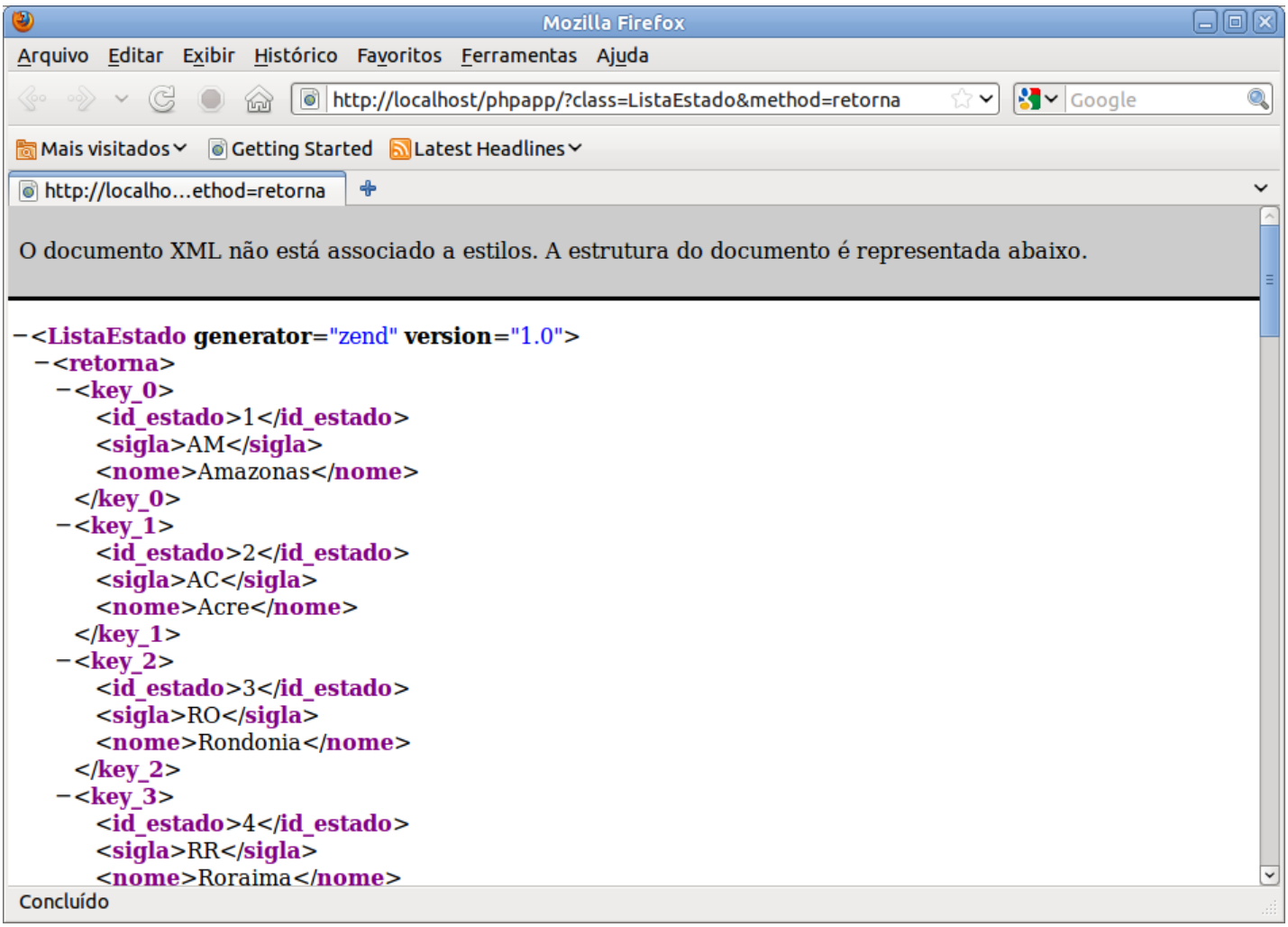
Para concluir a parte servidora desta aplicação falta apenas criar a classe que será requisitada pelo programa em Qt.

```
<?php

// Classes/ListaEstado.php

class ListaEstado {
    public function retorna()
    {
        include_once1('Classes/Conexao.php');
        $db = Conexao::getConexao();
        return $db->executaQuery("SELECT
                                    id_estado,
                                    sigla,
                                    nome
                                  FROM
                                    estado");
    }
}
```

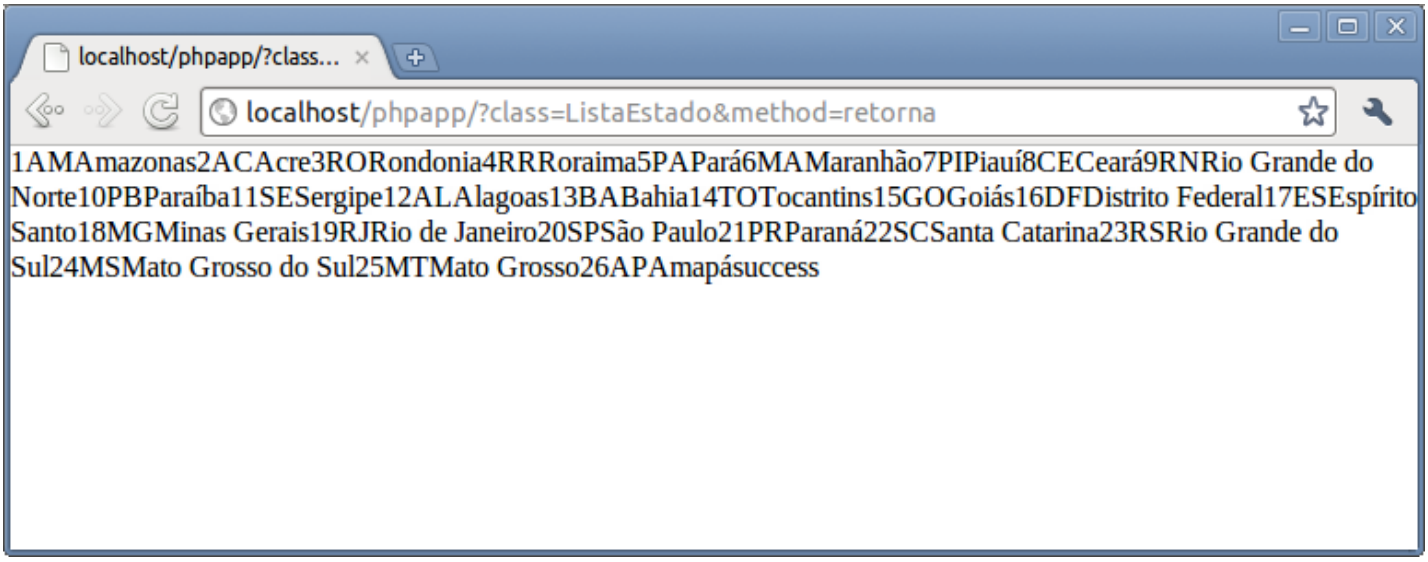
É isso. Para testar acesse a aplicação pelo *browser*, passando na url os argumentos **class** e **method**, como mostra a figura abaixo:



O resultado apresentado dependerá do browser que você estiver utilizando. O Firefox apresenta o XML retornado pela aplicação como mostra a próxima figura.



O Chrome suprime a exibição das *tags* XML, exibindo apenas os valores retornados. Se você selecionar a opção exibir código fonte da página (ou *View page source*), verá o XML completo.



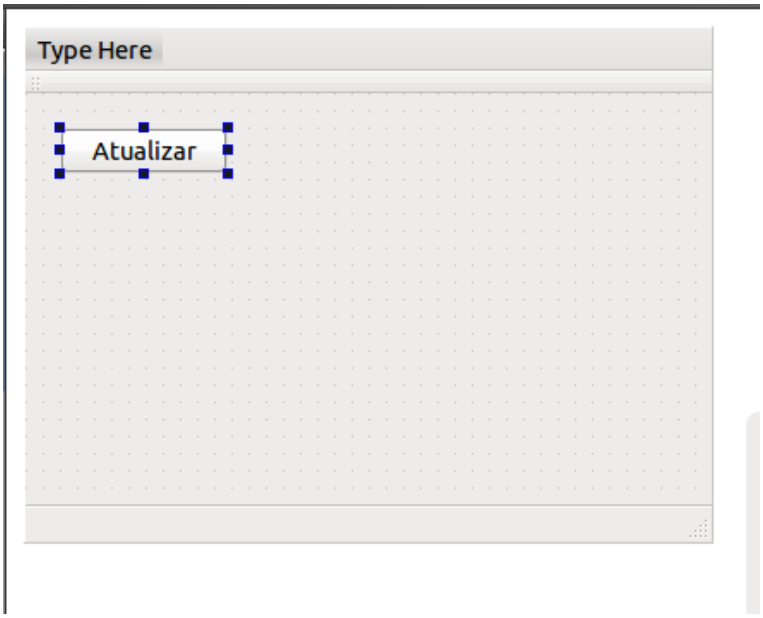
Agora que o servidor está pronto, vamos criar a parte cliente da nossa aplicação em Qt.

Como já foi esclarecido tanto no Editorial, como no início desta parte do tutorial, a editoria Laboratório (da qual o presente artigo faz parte) é dedicada àqueles com mais experiência em Qt. Serão abordados aqui, tópicos um pouco mais avançados e que vão portanto requerer do leitor o conhecimento básico de Qt.

Usando o Qt Creator, crie um projeto chamado QtPHP, tendo sua classe principal chamada QtPHP.

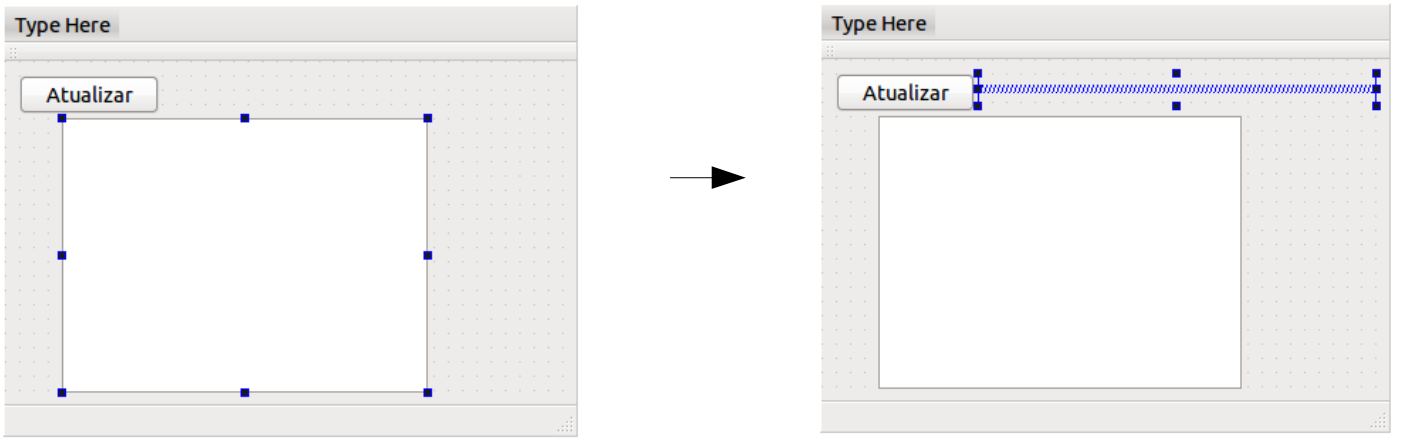
Quem tiver dúvidas sobre como criar uma aplicação em Qt, pode consultar o artigo **Alô, Qt Creator** publicado na primeira edição da **Revista Qt**.

Com um projeto com o nome de QtPHP criado, vamos à crição da interface (bem simples) da aplicação. Arraste para a janela da aplicação um componente **QpushButton.** Troque sua propriedade **objectName** para **btnAtualizar** e sua propriedade **text** para **Atualizar**.

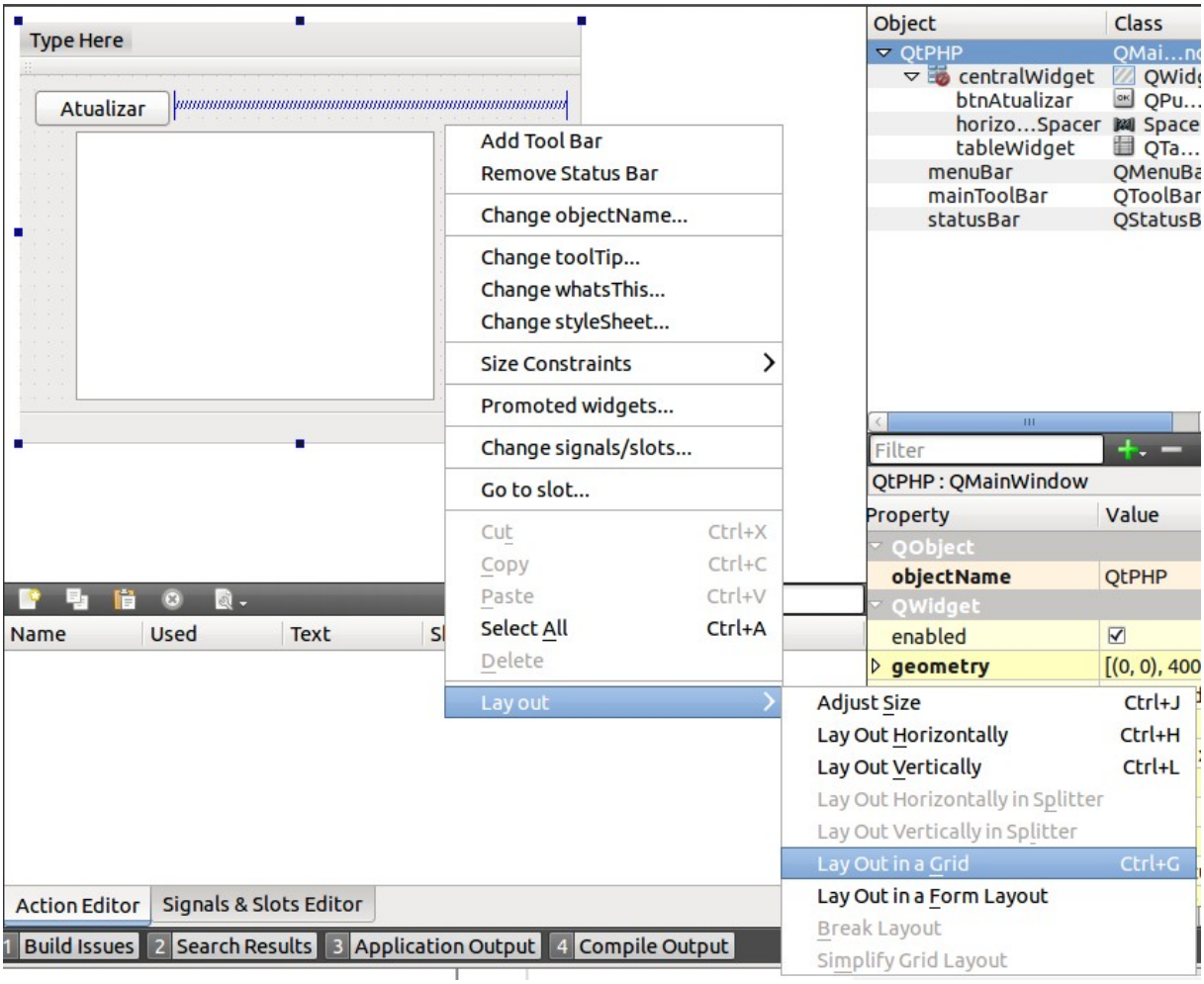


Arraste um componente **QtableWidget** para a janela da aplicação.

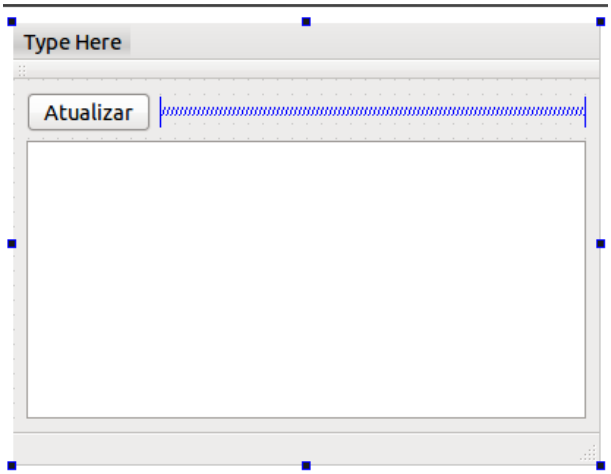
Coloque um componente **Spacer** entre o botão Atualizar e a borda da janela, como mostrado abaixo.



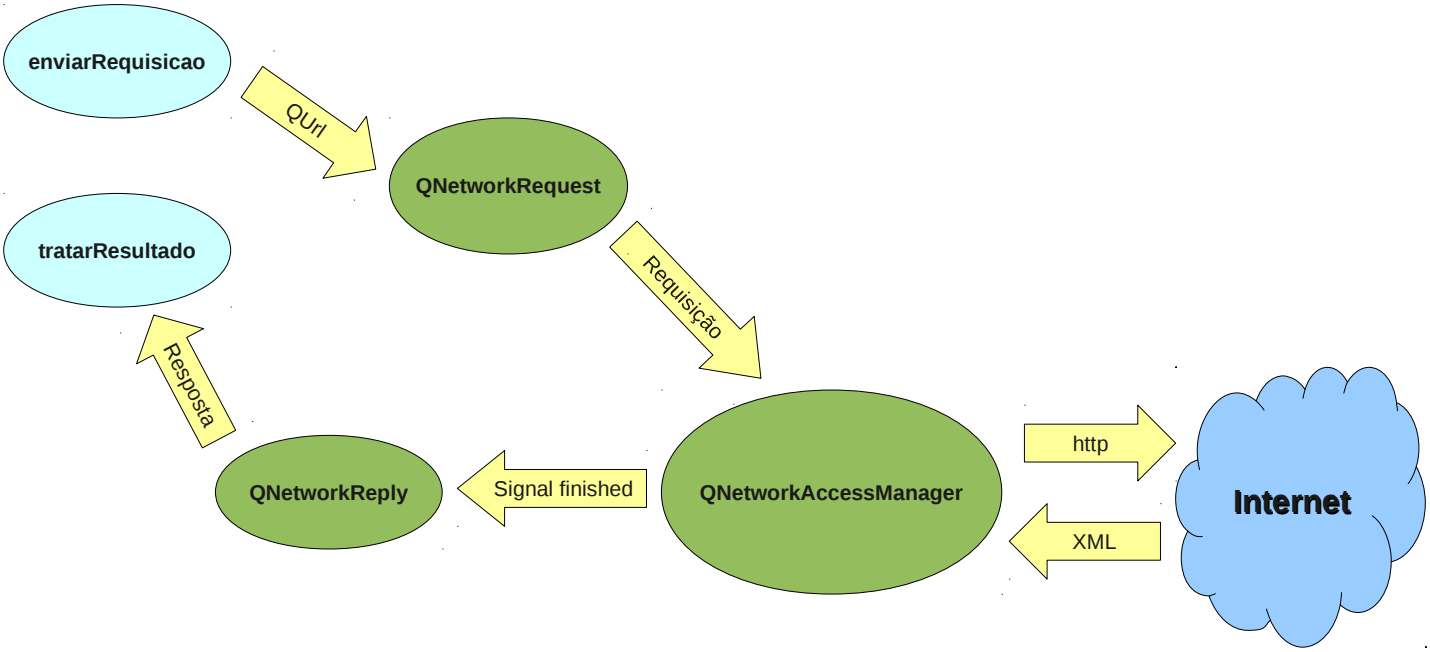
Usando o menu de contexto da janela de nossa aplicação (botão direito do mouse), selecione a opção **Layout → Layout in a Grid**.



A janela da aplicação terá a seguinte aparência, nesse ponto:



Com a interface da nossa aplicação “desenhada”, podemos passar ao código-fonte, mas antes vamos a uma breve descrição de seu funcionamento.



A base do nosso programa são os *slots* **enviarRequisicao** e **tratarResultado**. O primeiro, utiliza um objeto do tipo QNetworkRequest para enviar ao servidor uma requisição http. No caso desta aplicação, a requisição será:

```
http://localhost/phpapp/?class=ListaEstado&method=retorna
```

O *slot* **enviarRequisicao** monta uma URL que é submetida através de um objeto do tipo **QNetworkRequest** pelo método **get** de um objeto **QNetworkAccessManager**. O *signal* finished do objeto **QNetworkAccessManager** será conectado ao *slot* **tratarResultado** que receberá um objeto do tipo **QNetworkReply** com o resultado da requisição feita ao servidor.

O *slot* **tratarResultado** processará o XML recebido como resposta do servidor, preenchendo o grid **QTableWidget** que colocamos na interface do programa.

Nosso programa utilizará os módulos **QtNetwork** e **QtXml**, portanto, edite o arquivo **qtPHP.pro** e altere a linha:

```
QT       += core gui
```

para

```
QT       += core gui network xml
```

O código da função **main**, no arquivo **main.cpp** não apresenta qualquer alteração em relação ao criado pelo Qt Creator, como vemos a seguir:

```
#include <QtGui/QApplication>
#include "qtphp.h"

int main(int argc, char *argv[])
{
      QApplication a(argc, argv);
      QtPHP w;
      w.show();
      return a.exec();
}
```

Agora edite o arquivo **qtphp.h** para que seu conteúdo seja igual ao mostrado na listagem abaixo:

```
#ifndef QTPHP_H
#define QTPHP_H

#include <QMainWindow>
#include <QNetworkAccessManager>
#include <QNetworkRequest>
#include <QNetworkReply>
#include <QMessageBox>
#include <QdomDocument>
#include <QTimer>

namespace Ui {
      class QtPHP;
}

class QtPHP : public QMainWindow
{
      Q_OBJECT

public:
      explicit QtPHP(QWidget *parent = 0);
      ~QtPHP();

private slots:
      void enviarRequisicao();
      void tratarResultado(QNetworkReply * resposta);

private:
      Ui::QtPHP *ui;
      QNetworkAccessManager * requisicaoRede;
};

#endif // QTPHP_H
```

As linhas em destaque (cor vermelha) no código acima referem-se àquelas que devem ser incluídas em relação ao arquivo originalmente criado pelo Qt Creator.

Temos a inclusão das definições das classes que usaremos no programa, dos protótipos dos *slots* **enviarRequisicao** e **tratarResultado** e do atributo **requisicaoRede**.

Bom, agora vamos à parte mais importante deste programa. A implementação, propriamente dita, da classe QtPHP, no arquivo **qtphp.cpp**.

Como o código é um mais extenso (pouco mais de cem linhas), vamos ver cada parte dele separadamente.

Começando pelos includes feitos para o *header* da classe e para o *header* da classe correspondente à interface gráfica.

```
#include "qtphp.h"
#include "ui_qtphp.h"
```

A próxima parte do código da nossa classe é a declaração do construtor.

```
QtPHP::QtPHP(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::QtPHP)
{
    ui->setupUi(this);

    requisicaoRede = new QNetworkAccessManager(this);

    connect(ui->btnAtualizar,
            SIGNAL(clicked()),
            this,
            SLOT(enviarRequisicao()));

    connect(requisicaoRede,
            SIGNAL(finished(QNetworkReply*)),
            this,
            SLOT(tratarResultado(QNetworkReply*)));

    // Configura o QTableWidget
    ui->tableWidget->setColumnCount(3);
    ui->tableWidget->verticalHeader()->setVisible(false);

    // Monta o cabeçalho da tabela
    QStringList cabecalho;
    cabecalho << "Id" << "Sigla" << "Nome";
    ui->tableWidget->setHorizontalHeaderLabels(cabecalho);

    // Chama o SLOT enviarRequisicao
    QTimer::singleShot(0, this, SLOT(enviarRequisicao()));

}
```

As linhas em destaque (cor vermelha) foram incluídas no método construtor originalmente criado pelo Qt Creator.

Primeiro temos a instanciação de um objeto do **QNetworkAccessManager** que é o responsável neste caso pela comunicação entre nosso programa em Qt e a aplicação Web. Este objeto é um atributo da classe QtPHP, de modo que esteja acessível em todos os métodos da mesma. Se não fosse um atributo da classe, precisaríamos passar por referência a todos os métodos que precisam utilizá-lo.

Como temos um botão, chamado **btnAtualizar** que deverá executar uma requisição ao servidor, conectamos o *signal* **clicked** do mesmo ao *slot* **enviarRequisicao** da nossa classe.

Quando ocorrer uma requisição, o objeto **QNetworkAccessManager** aguarda pela resposta do servidor e emite um *signal* **finished** quando isto ocorrer. Para que nosso programa "saiba" que a requisição foi atendida e reaja à resposta, conectamos o *signal* **finished** do objeto **QNetworkAccessManager** ao *slot* **tratarResultado**, passando ao mesmo um ponteiro para um objeto do tipo **QNetworkReply**.

A seguir temos a configuração do objeto **QTableWidget** que exibirá a lista de estados obtida do servidor. Estamos setando o número de colunas para três (3) e ocultando o *header* vertical, ou seja, a coluna mostrada à esquerda da tabela com a numeração das linhas.

Continuando, setamos o cabeçalho (*header)* do objeto **QTableWidget**, com "Id" para a primeira coluna, "Sigla" para a segunda coluna e "Nome" para a terceira.

Agora vem um macete publicado por Mark Summerfield em seu livro *Advanced Qt Programming* da *Prentice Hall*. Observe a utilização de uma chamada ao método estático **singleShot** da classe **QTimer** para chamar o método **enviarRequisicao**. A ideia de colocar a chamada ao método **enviarRequisicao** no método construtor era de que, ao executar o programa, a lista de estados seja automaticamente carregada e exibida, sem que o usuário tenha que clicar no botão **Atualizar**.

Até aí tudo bem, mas porque não coloquei no final do método construtor simplesmente uma chamada ao método **enviarRequisicao**? De acordo com Mark, deve-se limitar ao construtor, chamadas a métodos que estejam relacionados à criação do objeto. A chamada ao método de carga da lista de estados pressupõe que a janela de nossa aplicação esteja "pronta". Chamadas diretas a métodos do objeto que está sendo construído são consideradas inseguras, porque não existe a garantia de que o objeto esteja pronto durante a execução do construtor.

Assim, Mark recomenda o uso de uma chamada ao método estático singleShot de QTimer com zero como argumento correspondente ao intervalo, como foi feito aqui:

```
    QTimer::singleShot(0, this, SLOT(enviarRequisicao()));}
```

Com esta instrução estamos executando em zero milissegundos (imediatamente, portanto), no objeto **this** o *slot* **enviarRequisicao**. Esta prática garante que a execução ocorrerá apenas quando o objeto correspondente a nossa aplicação esteja pronta.

Nenhuma alteração foi implementada no método destrutor da classe criado pelo Qt Creator, como visto a seguir:

```
/**
 * @brief Destrutor
 */
QtPHP::~QtPHP()
{
    delete ui;
}
```

O próximo método da nossa classe é o **enviarRequisicao**, que é um *slot* privado da classe **QtPHP**, para que possa ser conectado ao *signal* **clicked** do botão Atualizar.

```
void QtPHP::enviarRequisicao()
{
    ui->tableWidget->setRowCount(0);
    ui->tableWidget->clearContents();

    // Monta a URL para requisição dos dados ao servidor
    QString url = "http://localhost/phpapp/?class=ListaEstado&method=retorna";

    // Executa a requisição ao servidor
    requisicaoRede->get(QNetworkRequest(QUrl(url)));
}
```

As duas primeiras linhas do *slot* enviarRequisicao, servem para apagar o conteúdo do objeto tableWidget, que exibirá a lista de estados carregada do servidor. Desta forma, quando o usuário clicar no botão Atualizar, a nova lista carregada substituirá a anterior. A próxima instrução no método é a montagem da URL que será submetida ao servidor. Aqui foi utilizado um objeto do tipo QString para armazenar o endereço.

A última instrução do método usa o método get do atributo requisicaoRede que é um QNetworkAccessManager. O método get de QNetworkAccessManager recebe um objeto QNetworkRequest, o qual como o próprio nome indica, é uma requisição de rede. A requisição retornará ao QNetworkAccessManager um ponteiro para um objeto QNetworkReply, que receberá o resultado de tal requisição. Quando a resposta for recebida pelo objeto QNetworkAccessManager, este emitirá um *signal* finished. que foi conectado lá no construtor com o próximo e último método de nossa classe: o *slot* tratarResultado.

Como o código de tratarResultado é maior e é o responsável pela apresentação do resultado de nosso programa, vamos listar o seu código por blocos:

```
void QtPHP::tratarResultado(QNetworkReply * resposta)
{
    // Verifica se houve erro na resposta
    if(resposta->error() != QNetworkReply::NoError){
        QMessageBox::critical(this, "Erro",
                              "Não foi possível recuperar dados do servidor");
        return;
    }
```

O *slot* tratarResultado recebe como argumento um ponteiro para um objeto do tipo QNetworkReply passado para ele pelo objeto QNetworkAccessManager no momento em que o *signal* finished for emitido (lembre-se de que no construtor da classe QtPHP, conectamos o *signal* finished do objeto requisicaoRede que é um QNetworkAccessManager ao *slot* tratarResultado).

O primeiro passo em tratarResultado é verificar se ocorreu erro na requisição feita ao servidor. Para isso verificamos se o resultado da chamada ao método error do objeto resposta é diferente de QNetworkReply:NoError. Caso tenha ocorrido um erro, o programa emite uma mensagem informando ao usuário e retorna.

A resposta recebida do servidor será um XML com a seguinte estrutura:

```
<ListaEstado generator="zend" version="1.0">
      <retorna>
            <key_0>
                  <id_estado>1</id_estado>
                  <sigla>AM</sigla>
                  <nome>Amazonas</nome>
            </key_0>

            ...

            <key_25>
                  <id_estado>26</id_estado>
                  <sigla>AP</sigla>
                  <nome>Amapá</nome>
            </key_25>
            <status>success</status>
      </retorna>
</ListaEstado>
```

Na primeira linha temos a tag raiz do nosso XML, identificando nesse caso o nome da classe que foi executada para emissão da resposta – ListaEstado. Os atributos **generator** e **version** indicam respectivamente quem gerou o XML e qual a versão. A próxima tag do XML tem o nome do método executado – **retorna**. Em seguida, temos para cada um dos registros retornados a tag key_n, onde **n** representa o número sequencial do registro, começando por zero (0). Para cada registro, temos três tags indicando os nomes dos campos retornados, a saber:

<id_estado>

<sigla>

<nome>

O valor contido em nas tags acima é o conteúdo do registro propriamente dito.

```
// Cria objeto DOM para tratamento do XML de resposta e
// verifica se a resposta pode ser atribuída ao este objeto
QDomDocument doc;
if(!doc.setContent(resposta)){
    QMessageBox::critical(this,"Erro","Erro tratando resultado");
    return;
}
```

No próximo passo, caso não tenha ocorrido um erro no retorno da requisição do servidor, temos a criação de um objeto do tipo QDomDocument que será utilizado para tratamento do XML retornado. Na atribuição do conteúdo ao objeto QDomDocument verificamos se ocorreu erro. Se a resposta do servidor não puder ser atribuída ao objeto QDomDocument, o programa emite uma mensagem ao usuário e retorna.

Com o XML da resposta carregado no objeto QDomDocument podemos usar suas facilidades para navegar pelas tags do documento.

```
// Verifica se o retorno foi gerado pela classe ListaEstado
QDomElement classe = doc.documentElement();
if(classe.tagName() != "ListaEstado"){
    QmessageBox::critical(this,"Erro",
                          "O XML recebido não é da classe ListaEstado");
    return;
}
```

Neste trecho do código, utilizamos um objeto QDomElement para armazenar o primeiro elemento do documento DOM. Se o nome da tag do primeiro elemento do XML recebido do servidor não for **ListaEstado**, o programa avisa ao usuário e retorna.

```
// Nó correspondente ao nome do método - retorna
QDomNode metodo = classe.firstChild();
```

Em seguida o primeiro filho do elemento ListaEstado é atribuido a um objeto QDomNode. A tag filha de ListaEstado em nosso XML é **retorna**, que corresponde ao nome do método executado pelo servidor.

```
    // Nó correspondente ao registro
    QDomNode registro = metodo.firstChild();
```

O primeiro filho da tag **retorna**, lida no passo anterior, corresponde ao primeiro registro retornado pela consulta.

```
    // Nó correspondente ao status - o último
    QDomNode status = metodo.lastChild();
    if(status.toElement().text() != "success"){
        QMessageBox::critical(this,"Erro","O servidor retornou erro");
        return;
    }
```

Antes de começar a navegar pelos registros, temos a leitura do último filho da tag **retorna**, que corresponde ao *status* da execução do comando. Caso o texto deste último elemento seja diferente de “success”, significa que ocorreu um erro. Neste caso, o programa avisa ao usuário e retorna.

```
    // Percorre os registros
    int linha   = 0;
    while(!registro.isNull() && registro != status){
        ui->tableWidget->insertRow(linha);
        QDomNode campo = registro.firstChild();
        int coluna  = 0;
        // Percorre os campos
        while(!campo.isNull()){
            QTableWidgetItem * item = new QTableWidgetItem(campo.toElement().text());
            ui->tableWidget->setItem(linha, coluna, item);
            coluna++;
            campo = campo.nextSibling();
        }
        linha++;
        registro = registro.nextSibling();
    }
}
```

Se a execução do programa chegou a este ponto, só falta percorrer a lista de registros e popular o QTableWidget da nossa aplicação com os dados. Neste trecho do código, o primeiro *loop* **while** percorre registro, até que seja o final desde que não seja o registro de status.

Como cada campo de um registro é um filho seu, temos um segundo *loop* para percorrer os filhos de cada registro e setar a linha/coluna correspondente no QTableWidget com o conteúdo do campo.

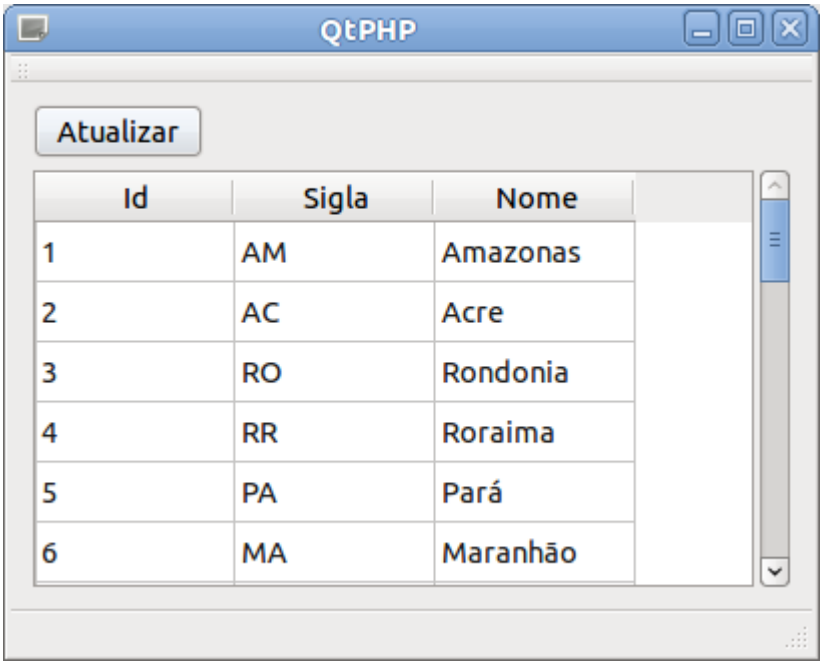
Para navegar entre os itens de um elemento do DOM, usamos o método **nextSibling**.

A seguir, a listagem completa do *slot* tratarResultado.

```
void QtPHP::tratarResultado(QNetworkReply * resposta)
{
    // Verifica se houve erro na resposta
    if(resposta->error() != QNetworkReply::NoError){
        QMessageBox::critical(this, "Erro",
                              "Não foi possível recuperar dados do servidor");
        return;
    }

    // Cria objeto DOM para tratamento do XML de resposta e
    // verifica se a resposta pode ser atribuída ao este objeto
    QDomDocument doc;
    if(!doc.setContent(resposta)){
        QMessageBox::critical(this,"Erro","Erro tratando resultado");
        return;
    }

    // Verifica se o retorno foi gerado pela classe ListaEstado
    QDomElement classe = doc.documentElement();
    if(classe.tagName() != "ListaEstado"){
        QMessageBox::critical(this,"Erro",
                              "O XML recebido não é da classe ListaEstado");
        return;
    }

    // Nó correspondente ao nome do método - retorna
    QDomNode metodo = classe.firstChild();

    // Nó correspondente ao registro
    QDomNode registro = metodo.firstChild();

    // Nó correspondente ao status - o último
    QDomNode status = metodo.lastChild();
    if(status.toElement().text() != "success"){
        QMessageBox::critical(this,"Erro","O servidor retornou erro");
        return;
    }

    // Percorre os registros
    int linha   = 0;
    while(!registro.isNull() && registro != status){
        ui->tableWidget->insertRow(linha);
        QDomNode campo = registro.firstChild();
        int coluna  = 0;
        // Percorre os campos
        while(!campo.isNull()){
            QTableWidgetItem * item = new QTableWidgetItem(campo.toElement().text());
            ui->tableWidget->setItem(linha, coluna, item);
            coluna++;
            campo = campo.nextSibling();
        }
        linha++;
        registro = registro.nextSibling();
    }
}
```

A explicação sobre o funcionamento deste programa não foi feita linha a linha, mas para aqueles com alguma experiência em Qt, os comentários colocados no código-fonte já devem ajudar bastante. A mesma observação vale para a parte PHP desta aplicação.

A figura a seguir mostra o resultado da execução do nosso programa:



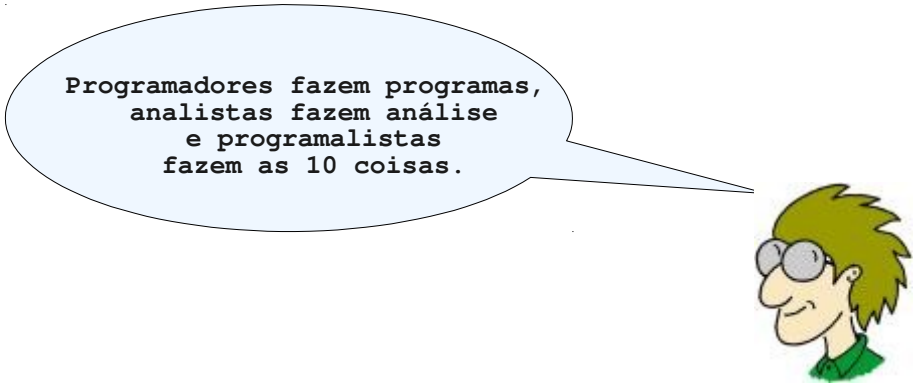
Comentando um dia desses sobre aplicações híbridas Qt + PHP com um amigo, ele questionou: não poderíamos fazer a parte servidora também em Qt, usando CGI? Verdade – poderíamos. Mas fazer a parte servidora usando a dupla PHP / Zend Framework facilita muito o trabalho. Não precisamos nos preocupar com detalhes da conexão com o banco de dados ou com a formatação do resultado em XML.

É comum encontrar programadores especialistas em uma ferramenta, querendo utilizá-la para tudo aquilo que pretendam fazer. Sempre uso o seguinte exemplo para ilustrar esta situação:

Imagine um trabalhador que possui um excelente conjunto de chaves de fenda. Vários tamanhos, torques, modelos, etc. - uma maravilha... Mas se tentar colocar um prego, um martelo seria mais útil do que todo o seu conjunto de chaves. A escolha da ferramenta deve ser feita de acordo com o trabalho a ser feito. Ok, eu admito - tá parecendo coisa de livro de “auto-ajuda”, mas se tem uma coisa que os meus vinte e poucos anos de programação me ensinaram é que não existe ferramenta definitiva.

Na próxima edição continuamos com mais exemplos de aplicações híbridas, utilizando Qt e PHP.

Um grande abraço.



Geek & Poke

Por: Oliver Widder
oliver.widder@gmail.com

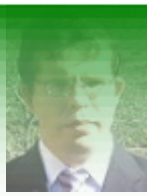
Graças ao seu Apple agora você se sente seguro

Outra tentativa fracassada de viver completamente nas nuvens.

http://geekandpoke.typepad.com

O novo paradigma - parte 1

Criando aplicativos com Qt 4.7 no Mac OS X



Por: Pierre Freire
blog@pierrefreire.com.br

Neste tutorial iremos aprender a como instalar um ambiente de desenvolvimento com QT no Mac OS X.

Com o crescimento da Apple no mercado, tem se tornado muito comum o uso de notebooks e desktops da empresa de Cupertino por usuários e empresas, com isto a demanda por novos aplicativos tem crescido bastante e este novo filão para os programadores esta apenas começando.
O objetivo deste tutorial é abordar a instalação do SDK (Kit de Desenvolvimento da QT para Mac OS), outras informações os links abaixo poderão ajudar bastante.

http://www.revistaqt.com
http://qt.nokia.com

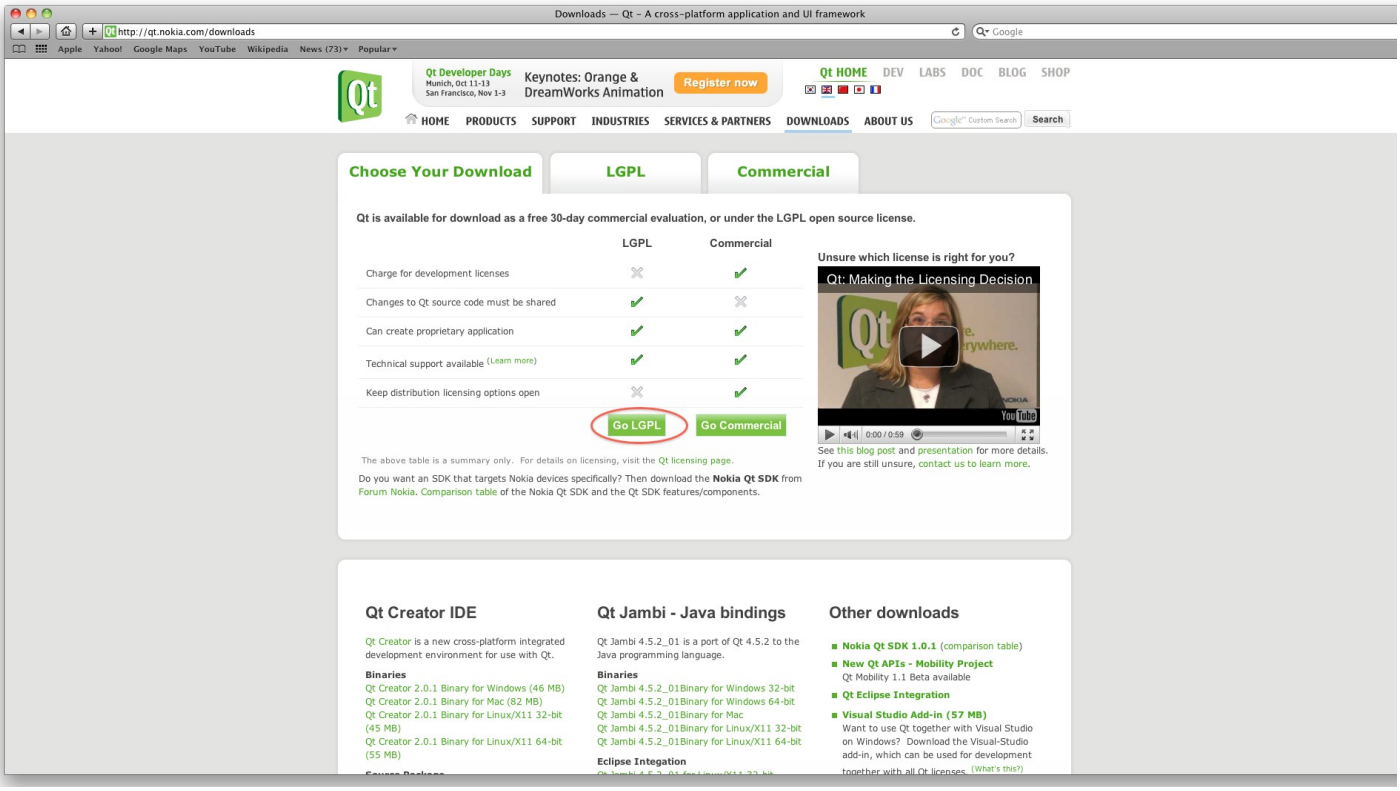
Ambiente testado:

MacBook White
Sistema operacional Snow Leopard 10.6.4

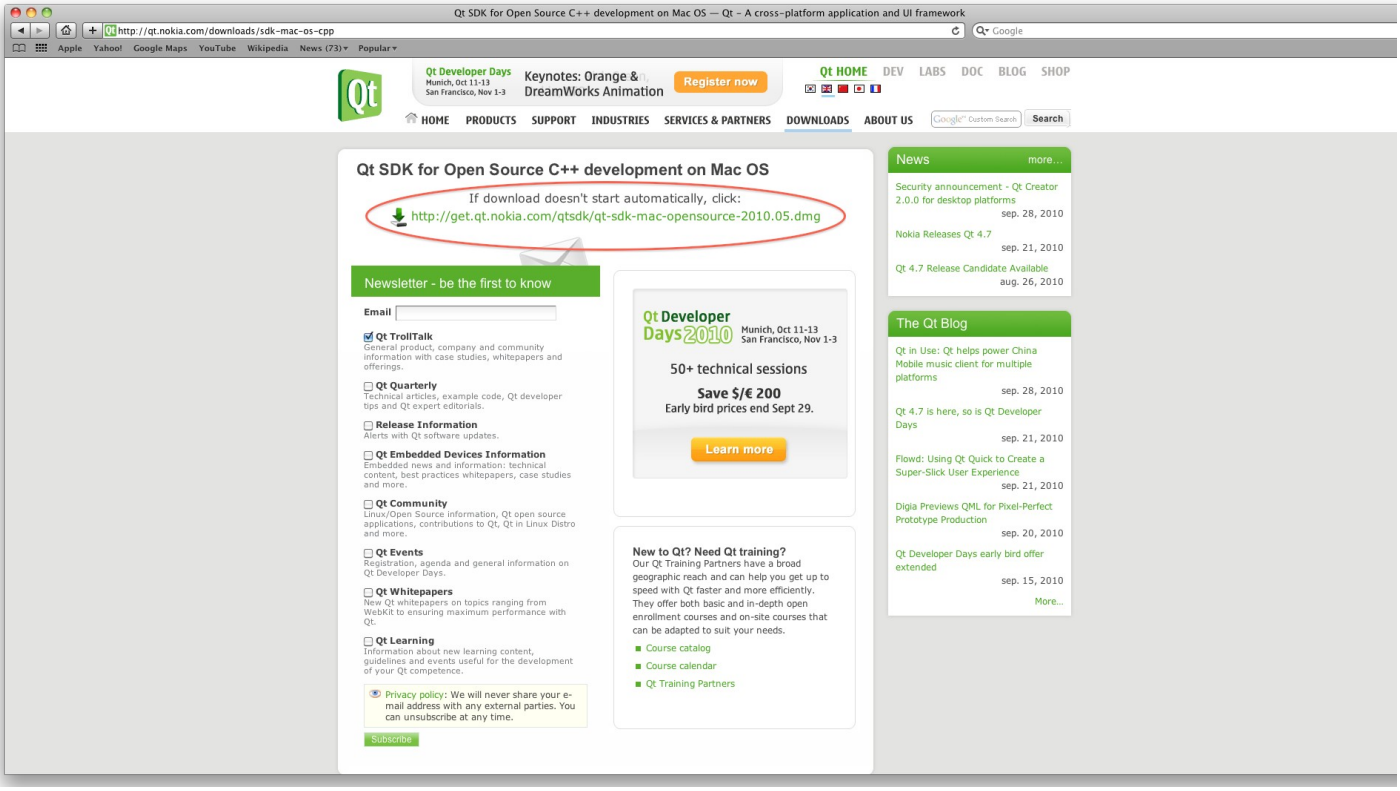
1 – Acessando o site da Nokia

1.1 - O Kit de desenvolvimento da QT pode ser obtido gratuitamente no site http://qt.nokia.com/ acessando o link de DOWNLOADS:

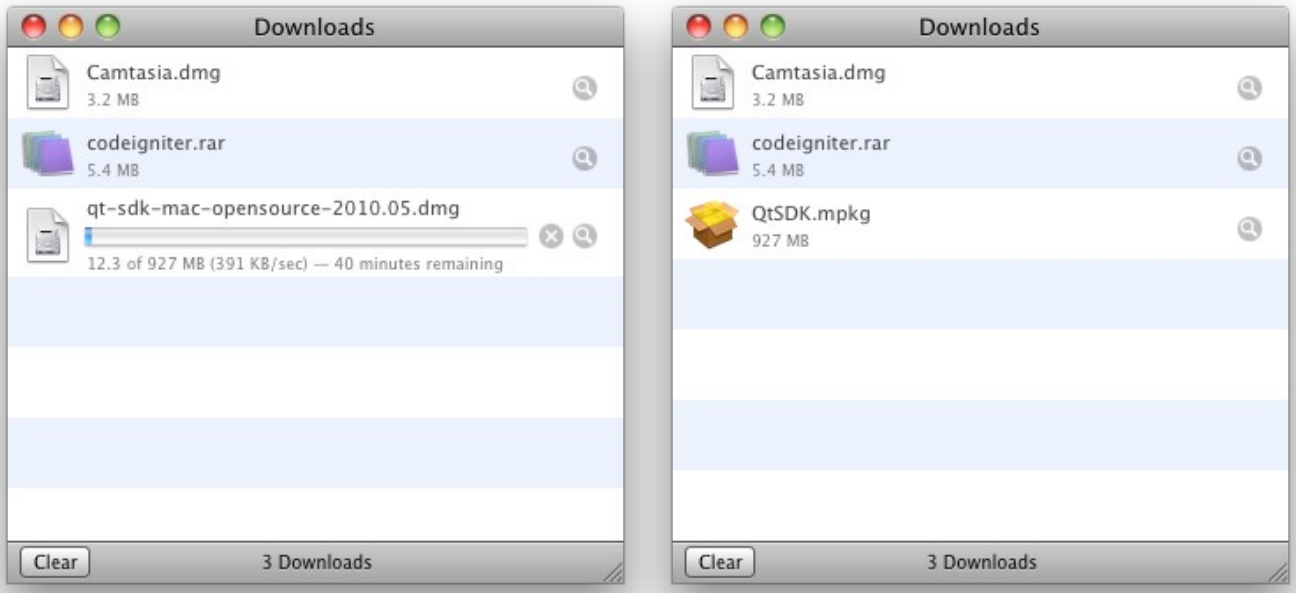
1.2 – Você tem agora a opção de usar a versão comercial ou lgpl (livre), no nosso tutorial estamos baixando a versão lgpl, conforme a imagem abaixo:



1.3 – Caso o download não comece, clique no link, conforme a figura:

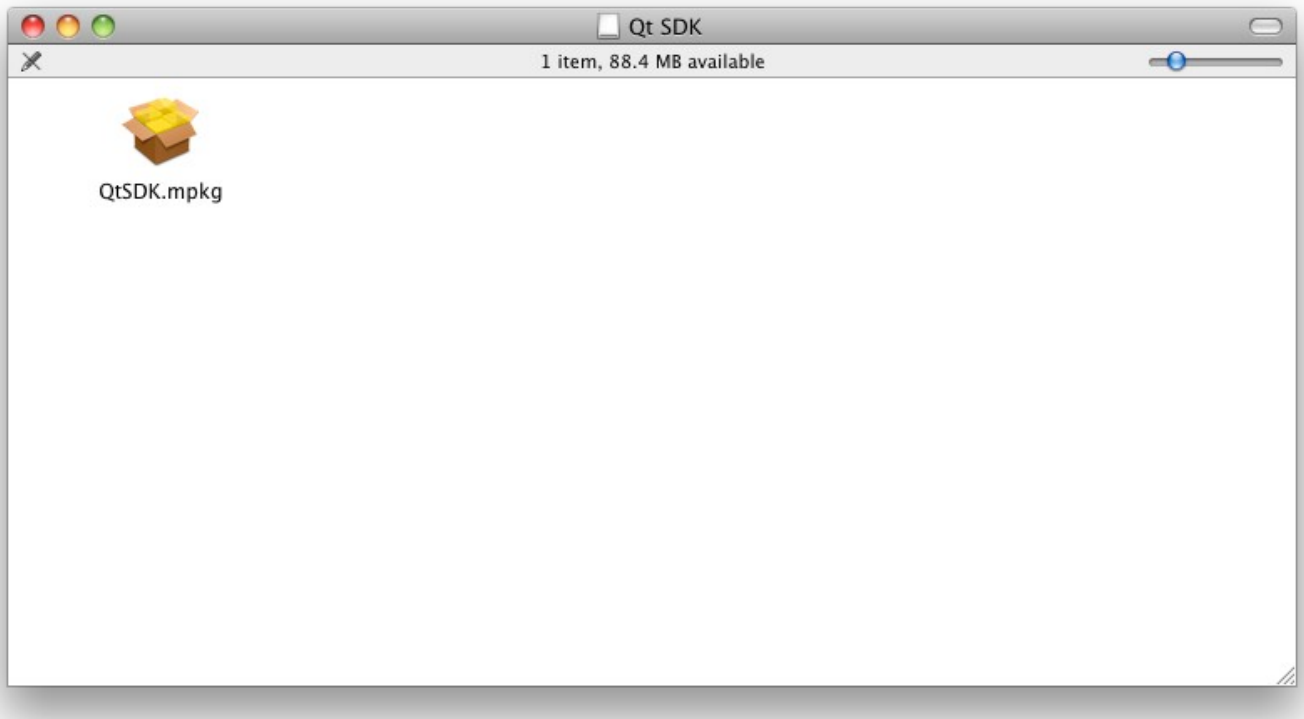


1.4 – A velocidade do download vai depender da sua banda larga, agora a ordem é esperar.



2 – Iniciando a instalação

O download foi efetuado com sucesso e o arquivo QtSDK.mpkg irá aparecer no desktop do seu computador, ele irá abrir automaticamente, caso isto não aconteça clique no icone.



2.1 – A tela do instalador aparece e o processo se inicia.

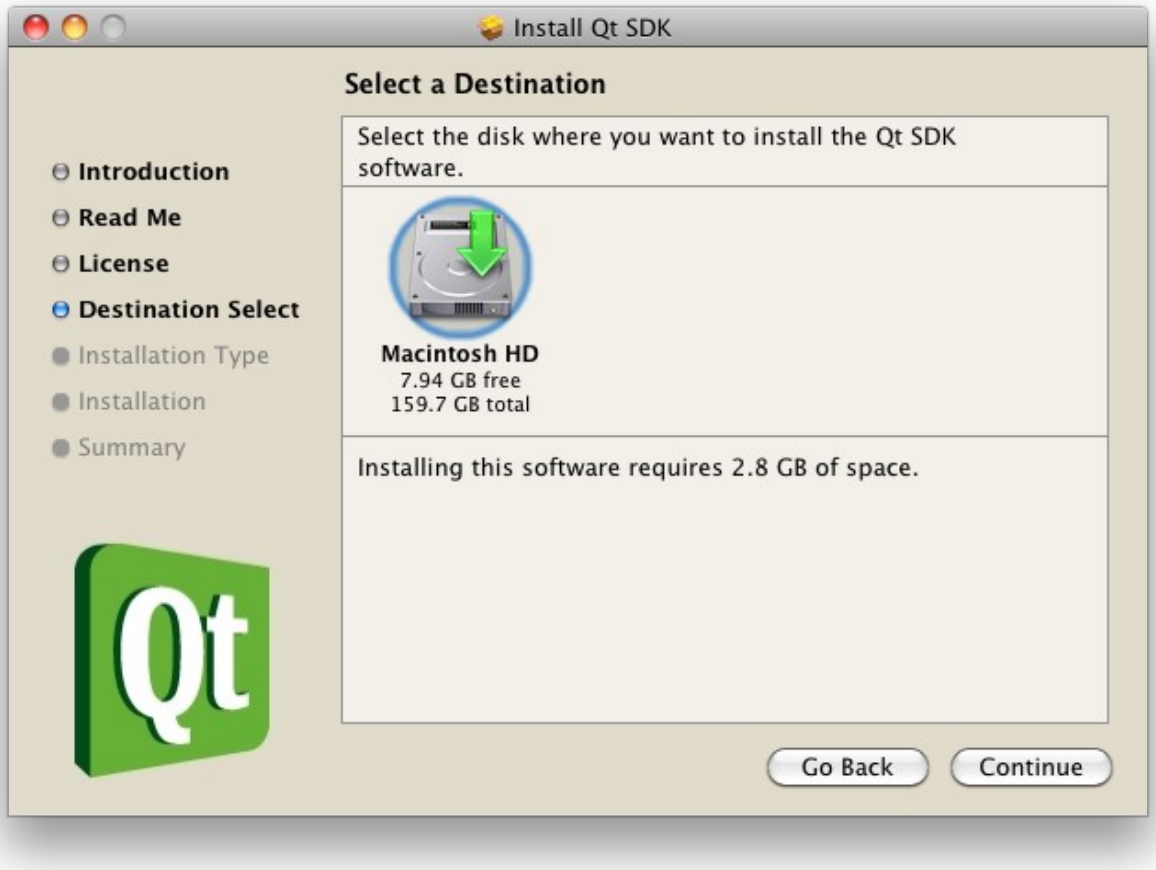
2.2 – Tela de boas vindas com algumas informações sobre o produto.

2.3 – Um read me com informações sobre a versao do QT e informações em geral, podemos seguir em frente.

2.4 – Informações sobre o licenciamento do software, caso tenha interesse leia e siga em frente.

2.5 – Esta janela, quer uma confirmação se você aceita os termos da licença clique em Agree caso você concorde em usar.

2.6 – O instalador irá mostrar as unidades de disco e o espaço disponível , e o espaço que a instalação do QT vai ocupar no disco.



2.7 – Se quiser mudar o local de instalação. No tutorial deixamos no padrão.

2.8 – Como esta instalando um software, por motivos de segurança o Mac Os vai solicitar o seu usuário e senha do administrador.

2.9 – O processo de instalação foi iniciado, agora é esperar.

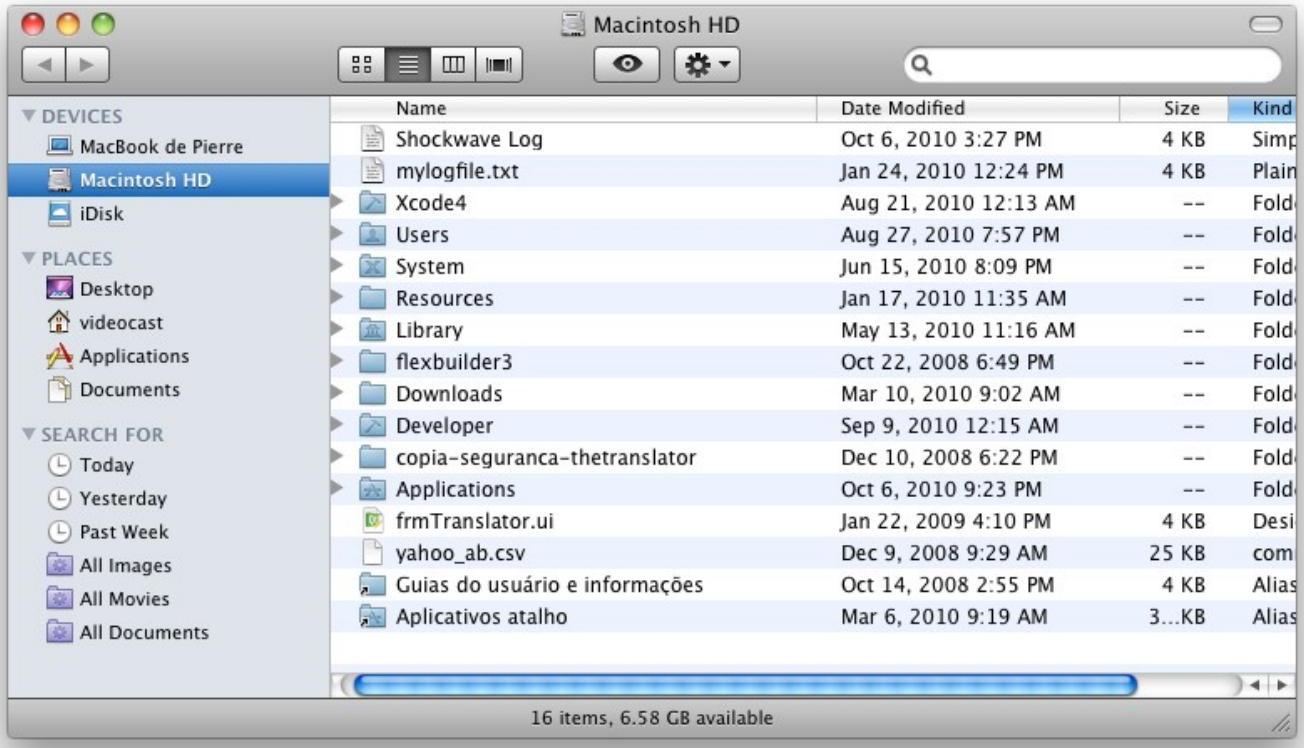
2.10 – Se você chegou a este ponto, significa que o QT foi instalado com sucesso no seu computador.

3.0 - Testando o ambiente instalado

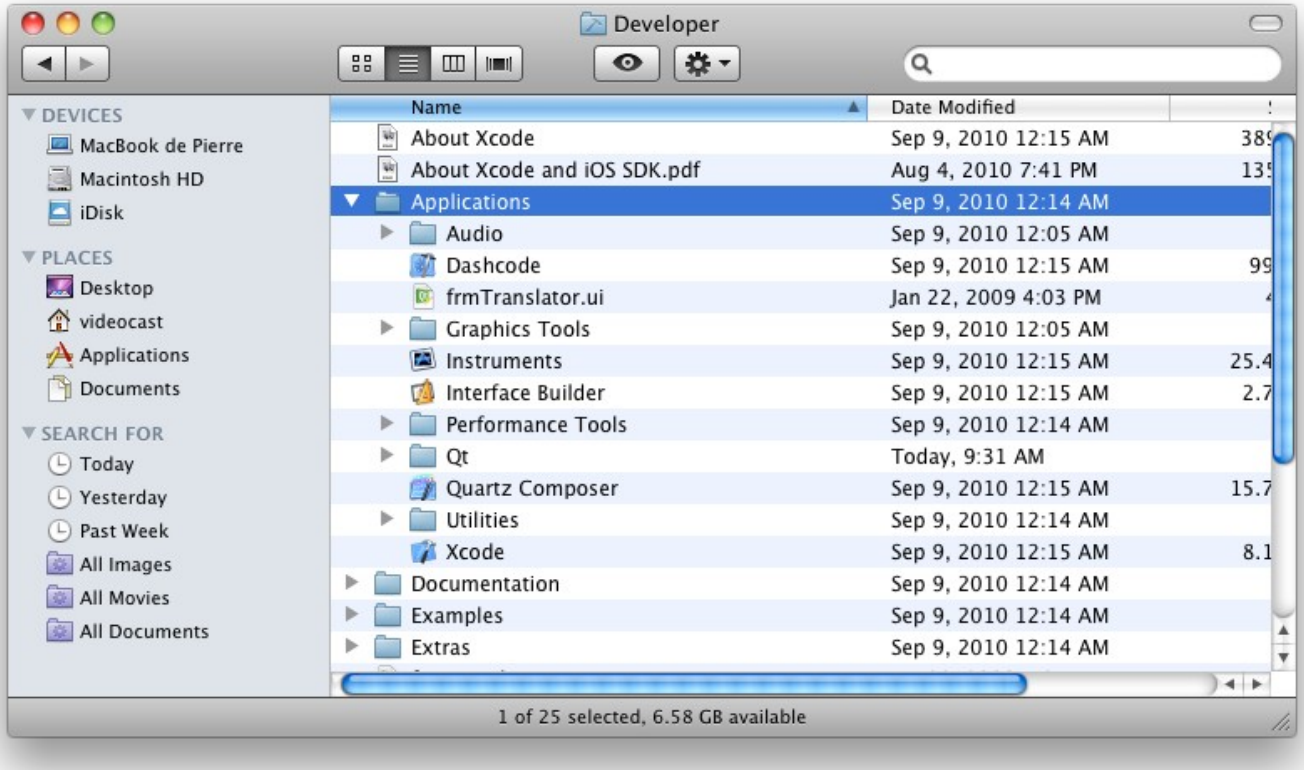
Com o SDK do Qt instalado, vamos agora conhecer a cara do QT.

3.1 – Usando o Finder ou o atalho no seu desktop acesse o seu Macintosh HD.

Temos a pasta Developer onde ficam as ferramentas de desenvolvimento.



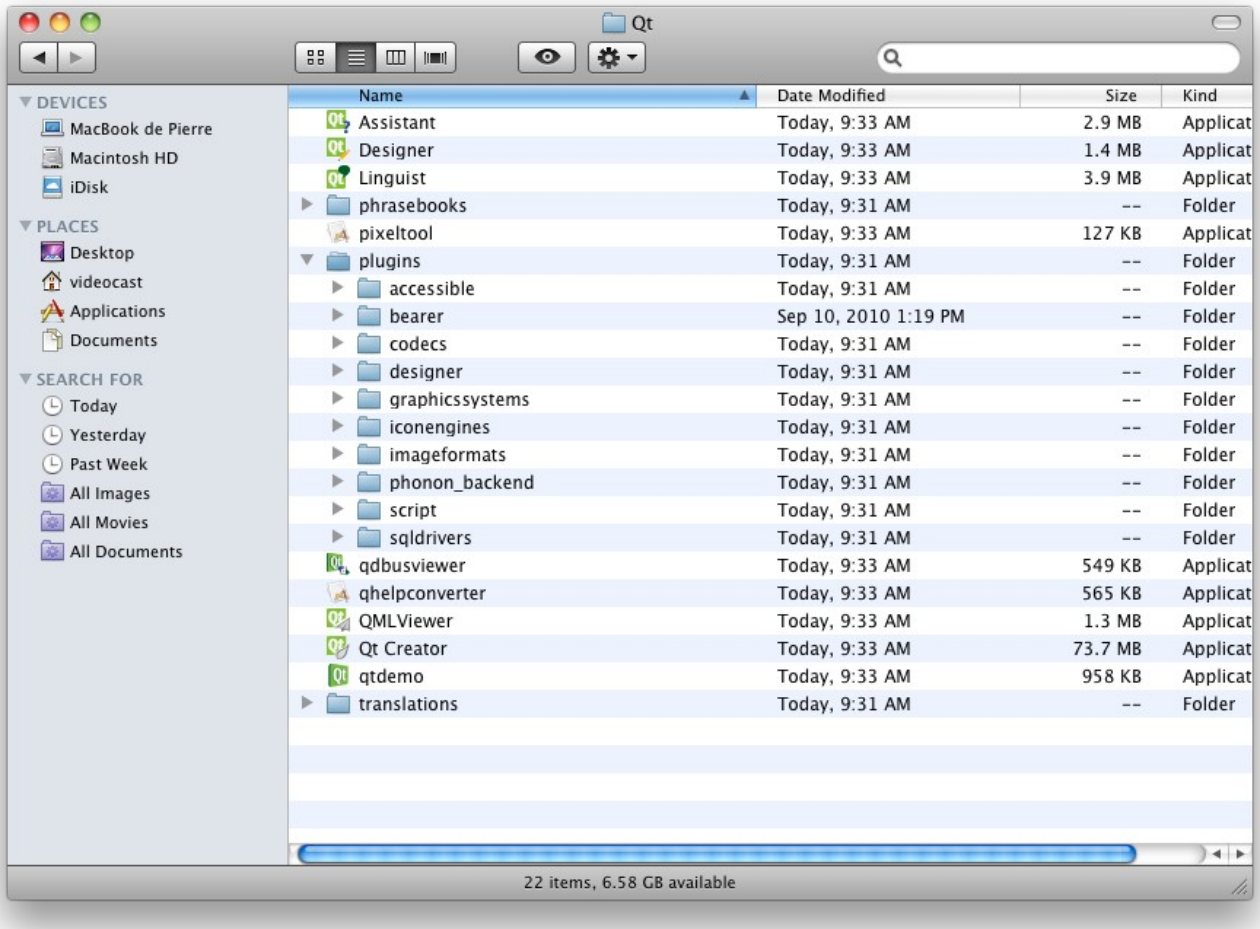
- Dentro da pasta Developer selecione Applications, agora entre na pasta QT.



-.Dentro da pasta Qt e vários aplicativos estarão disponíveis:

- **Assistant,**
- **Qt Creator**
- **qtdemo (Exemplos feitos em Qt)**

Clique no Qt Creator para conhecer a IDE de desenvolvimento.

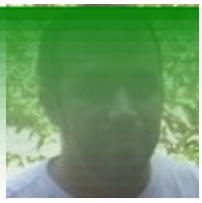


3.2 – O Qt Creator é a IDE de desenvolvimento oficial.

Chegamos ao final deste tutorial, onde abordamos a instalação das ferramentas de desenvolvimento para Mac.

O próximo passo agora é o seu, estudar, pesquisar e conhecer este fascinante mundo do QT.

Estilo de codificação QT



Por: André Vasconcelos
alovasconcelos@gmail.com

Esta é uma tradução livre do documento *Qt Coding Style* com algumas convenções utilizadas no desenvolvimento do Qt propriamente dito. Trata-se de uma pequena lista de recomendações àqueles que queiram participar ativamente do projeto, baixando os fontes do Qt e criando novas classes, novos módulos, etc. Achei interessante reproduzir aqui como sugestão de estilo. Vamos ao texto:

Estilo de Codificação Qt

Este é um resumo da convenção de codificação que usamos para escrever código do Qt. Estes dados foram recolhidos pela "mineração" dos fontes do Qt, fóruns de discussão, tópicos de email e através da colaboração dos desenvolvedores.

Indentação

- Quatro (4) espaços são usados para identação
- Espaços, não Tabs!

Declaração de variáveis

- Declare cada variável em uma linha separada
- Evite usar nomes curtos (por exemplo: “a”, “rbarr”, “nughdeget”) sempre que possível
- Nomes de variáveis com apenas um caractere só devem ser usadas para contadores e variáveis temporárias, quando o propósito da variável seja óbvio
- Declare uma variável no momento em que ela seja necessária

```
 // Errado
int a, b;
char *c, *d;

// Correto
int height;
int width;
char *nameOfThis;
char *nameOfThat;
```

• Nomes de variáveis devem começar com letras minúsculas. Cada palavra seguinte no nome de uma variável deve começar com uma letra maiúscula
• Evite abreviações

```
// Errado
short Cntr;
char ITEM_DELIM = '\t';

// Correto
short counter;
char itemDelimiter = '\t';
```

• Classes sempre começam com uma letra maiúscula. Classes públicas do Qt começam com uma letra Q (maiúscula). Funções públicas na maioria das vezes, começam com uma letra q (minúscula)

Espaços em branco

• Use linhas em branco para agrupar declarações onde for adequado
• Sempre use apenas uma linha em branco
• Sempre use um espaço único depois de uma palavra-chave e antes de chaves.

```
// Errado
if(foo){
}

// Correto
if (foo) {
}
```

• Para ponteiros ou referências, sempre use um único espaço entre o tipo e o caracter '*' ou '&'*', mas nenhum espaço entre o '*' ou '&' e o nome da variável.

```
char *x;
const QString &myString;
const char * const y = "hello";
```

• Não use espaço depois de um cast.

• Evite conversões (*casts*) no estilo de C quando possível.

```
// Errado
char* blockOfMemory = (char* ) malloc(data.size());

// Correto
char *blockOfMemory = (char *)malloc(data.size());
char *blockOfMemory = reinterpret_cast<char *>(malloc(data.size()));
```

Chaves

• Como regra básica, a chave da esquerda vem na mesma linha do começo da declaração:

```
// Errado
if (codec)
{
}

// Correto
if (codec) {
}
```

• Exceção: Implementações de funções e declarações de classes sempre têm a chave esquerda no começo de uma nova linha:

```
static void foo(int g)
{
    qDebug("foo: %i", g);
}

class Moo
{
};
```

• Use chaves quando o corpo de uma declaração condicional contiver mais de uma linha, ou quando contiver apenas uma linha que seja um pouco mais complexa

```
// Errado
if (address.isEmpty()) {
    return false;
}

for (int i = 0; i < 10; ++i) {
    qDebug("%i", i);
}

// Correto
if (address.isEmpty())
    return false;

for (int i = 0; i < 10; ++i)
    qDebug("%i", i);
```

• Exceção 1: Use chaves também se a declaração contiver várias linhas

```
// Correto
if (address.isEmpty() || !isValid()
    || !codec) {
    return false;
}
```

• Exceção 2: Use chaves também em blocos if-then-else caso o bloco de código do if ou do else tenha mais de uma linha

```
// Errado
if (address.isEmpty())
    return false;
else {
    qDebug("%s", qPrintable(address));
    ++it;
}

// Correto
if (address.isEmpty()) {
    return false;
} else {
    qDebug("%s", qPrintable(address));
    ++it;
}

// Errado
if (a)
    if (b)
        ...
    else
        ...

// Correto
if (a) {
    if (b)
        ...
    else
        ...
}
```

• Use chaves quando o corpo de uma declaração for vazia

```
// Errado
while (a);

// Correto
while (a) {}
```

Parênteses

• Use parênteses para agrupar expressões:

```
// Errado
if (a && b || c)

// Correto
if ((a && b) || c)

// Errado
a + b & c

// Correto
(a + b) & c
```

Declarações Switch

• Os rótulos "case" das declarações switch estão sempre na mesma coluna da declaração switch

• Todo caso deve ter uma declaração break (ou return) no final ou um comentário para indicar que intencionalmente não foi colocado o break

```
switch (myEnum) {
case Value1:
    doSomething();
    break;
case Value2:
    doSomethingElse();
    // fall through
default:
    defaultHandling();
    break;
}
```

Quebras de linhas

• Mantenha as linhas com menos de cem (100) caracteres; utilize quebras se necessário.

• Vírgulas vão no final das linhas quebradas; operadores vão no início de uma nova linha. O operador estará no final de uma linha para evitar ter que rolar (*scroll*) a tela se o editor for muito estreito.

```
// Correto
  if (longExpression
      + otherLongExpression
      + otherOtherLongExpression) {
  }

  // Errado
  if (longExpression +
      otherLongExpression +
      otherOtherLongExpression) {
  }
```

Exceção geral

• Sinta-se livre para quebrar uma regra se esta fizer o seu código parecer ruim.

No dia 18 de novembro, a Nokia irá realizar um webinar em português sobre o desenvolvimento de apliações em Qt para dispositivos com Symbian.

Já no dia 9 de dezembro, será a vez de um webinar sobre desenvolvimento de interfaces gráficas de usuários com Qt 4.7 e Qt Quick, também em português.

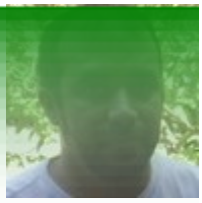
Para inscrever-se no webinar sobre Qt para dispositivos Symbian, acesse:

http://forumnokia.emea.acrobat.com/e93667679/event/registration.html

Para inscrever-se no webinar sobre desenvolvimento de interfaces gráficas de usuários com QT 4.7 e Qt Quick, acesse:

http://forumnokia.emea.acrobat.com/e60819127/event/registration.html

Tutorial QML / Layout Anchor-based em QML / Introdução à Linguagem QML



Por: André Vasconcelos
alovasconcelos@gmail.com

Mais uma da série “Versão Brasileira”. Desta vez trago uma tradução do *QML Tutorial* disponível no endereço http://doc.qt.nokia.com/4.7-snapshot/qml-tutorial.html. O Tutorial QML está dividido em três partes. De lambuja, a tradução de mais dois tutoriais relacionados ao Tutorial QML. Um sobre layouts baseados em âncoras (Anchor-based) e outro sobre a linguagem QML. Vamos lá...

Tutorial QML

Este tutorial traz uma introdução ao **QML**, a linguagem de marcação para o **Qt Quick**. Ele não cobre todos os aspectos; a ênfase é no ensino dos princípios fundamentais, e os recursos são apresentados à medida em que sejam necessários.

Através dos diferentes passos deste tutorial nós aprenderemos sobre os tipos básicos, criaremos nosso próprio componente QML com propriedades e sinais (signals), e criaremos uma animação simples com a ajuda dos estados (states) e transições (transitions).

O primeiro capítulo começa com um programa "Hello world!" mínimo e o capítulo seguinte apresenta novos conceitos.

O código fonte do tutorial pode ser encontrado no diretório:

$QTDIR/examples/declarative/tutorials/helloworld

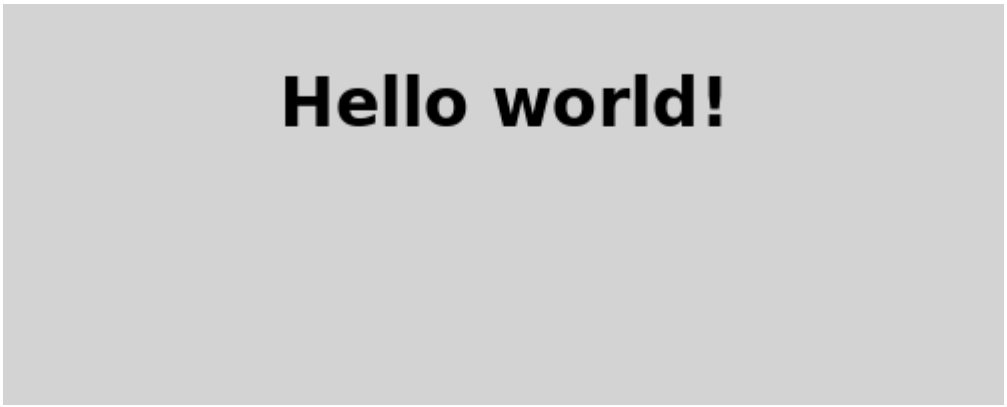
onde $QTDIR corresponde ao diretório no qual você instalou o SDK do Qt 4.7.

Capítulos do tutorial:

1. Tipos básicos
2. Componentes QML
3. Estados (States) e transições (Transitions)

Tutorial QML 1 - Tipos básicos

O primeiro programa é um exemplo de "Hello world!" muito simples que apresenta alguns conceitos básicos do QML. A figura abaixo é um screenshot deste programa.



Aqui está o código QML da aplicação:

```
import QtQuick 1.0

Rectangle {
    id: page
    width: 500; height: 200
    color: "lightgray"

    Text {
        id: helloText
        text: "Hello world!"
        y: 30
        anchors.horizontalCenter: page.horizontalCenter
        font.pointSize: 24; font.bold: true
    }
}
```

Passo a passo

Import

Primeiro nós precisamos importar os tipos necessários a este exemplo. Na maioria dos arquivos QML importaremos os tipos QML embutidos (como Rectangle, Image, etc) que vêm com o Qt usando:

```
import QtQuick 1.0
```

Elemento Rectangle

```
Rectangle {
    id: page
    width: 500; height: 200
    color: "lightgray"
```

Declaramos um elemento raiz do tipo Rectangle. Este é um dos blocos de construção básicos que você pode usar para criar uma aplicação em QML. Nós atribuimos um id para poder fazer referência ao mesmo mais tarde. Neste caso, nós o chamamos de "page". Nós também setamos as propriedades width (tamanho), height (altura) e color (cor). O elemento Rectangle contém muitas outras propriedades (como x e y), mas estes são deixados com seus valores default.

Elemento Text

```
    Text {
        id: helloText
        text: "Hello world!"
        y: 30
        anchors.horizontalCenter: page.horizontalCenter
        font.pointSize: 24; font.bold: true
    }
```

Adicionamos um elemento Text como filho do elemento raiz Rectangle que exibe o texto "Hello world!".

A propriedade y é usada para posicionar o texto verticalmente a 30 pixels do topo de seu pai.

A propriedade anchors.horizontalCenter refere-se à centralização horizontal de um elemento. Neste caso, especificamos que nosso elemento Text deve ser centralizado horizontalmente no elemento page (veja Layout baseado em âncora "Anchor-based Layout").

As propriedades font.pointSize e font.bold são relativas às fontes e usam a notação de ponto.

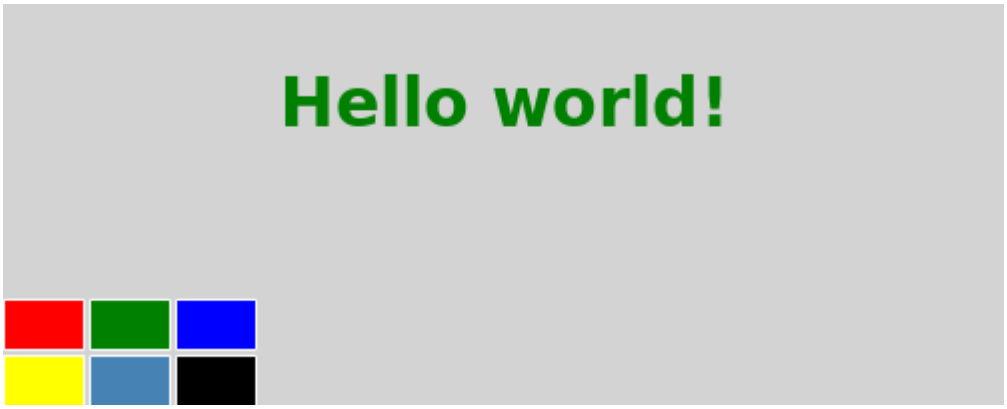
Visualizando o exemplo

Para ver o que você criou, execute a ferramenta de visualização QML (localizada no diretório bin do SDK do Qt) com o nome do arquivo como primeiro argumento. Por exemplo, para executar o exemplo completo do Tutorial 1 a partir do local de instalação, você deveria digitar:

bin/qmlviewer $QTDIR/examples/declarative/tutorials/helloworld/tutorial1.qml

Tutorial QML 2 - Componentes QML

Este capítulo acrescenta um seletor de cores (color picker) para alterar a cor do texto.



Nosso seletor de cores é feito de seis células com diferentes cores. Para evitar escrever o mesmo código várias vezes para cada célula, criamos um novo componente Cell. Um componente proporciona uma forma de definir um novo tipo que podemos reutilizar em outros arquivos QML. Um componente QML é como uma caixa preta e interage com o mundo exterior através de propriedades, sinais (signals) e funções e é geralmente definido em seu próprio arquivo QML. (Para mais detalhes veja a seção "Definindo novos componentes"). O nome do arquivo do componente deve sempre começar com uma letra maiúscula.

Aqui está o código QML do arquivo Cell.qml:

```
import QtQuick 1.0

  Item {
      id: container
      property alias cellColor: rectangle.color
      signal clicked(color cellColor)

      width: 40; height: 25

      Rectangle {
          id: rectangle
          border.color: "white"
          anchors.fill: parent
      }

      MouseArea {
          anchors.fill: parent
          onClicked: container.clicked(container.cellColor)
      }
  }
```

Passo a passo

O componente Cell

```
  Item {
      id: container
      property alias cellColor: rectangle.color
      signal clicked(color cellColor)

      width: 40; height: 25
```

O elemento raiz do nosso componente é um Item com o id container. Um Item é o elemento visual mais básico em QML e é frequentemente usado com um container para outros elementos.

```
property alias cellColor: rectangle.color
```

Declaramos uma propriedade cellColor. Esta propriedade é acessível de fora do nosso componente, o que nos permite instanciar as células com diferentes cores. Esta propriedade é apenas um apelido (alias) para uma propriedade existente - a propriedade color de um retângulo que compõe a célula (veja Adicionando novas propriedades).

```
signal clicked(color cellColor)
```

Nós precisamos que nosso componente tenha também um signal que chamaremos de clicked com um parâmetro do tipo color. Usaremos este signal para mudar a cor do texto no arquivo QML principal mais tarde.

```
 Rectangle {
          id: rectangle
          border.color: "white"
          anchors.fill: parent
}
```

Nosso componente cell é basicamente um retângulo colorido com o id rectangle.

A propriedade anchors.fill é uma forma conveniente de atribuir o tamanho de um elemento. Neste caso, o retângulo terá o mesmo tamanho de seu pai (veja Layout baseado em âncora "Anchor-based Layout").

```
MouseArea {
          anchors.fill: parent
          onClicked: container.clicked(container.cellColor)
}
```

Para mudar a cor do texto quando clicar em uma célula, criamos um elemento MouseArea com o mesmo tamanho de seu pai.

Um MouseArea define um signal chamado clicked. Quando o signal é disparado precisamos emitir nosso próprio signal clicked com a cor como parâmetro;

O arquivo QML principal

Em nosso arquivo QML principal, usamos nosso componente Cell para criar o seletor de cores:

```
import QtQuick 1.0

Rectangle {
   id: page
   width: 500; height: 200
   color: "lightgray"

   Text {
     id: helloText
     text: "Hello world!"
     y: 30
     anchors.horizontalCenter: page.horizontalCenter
     font.pointSize: 24; font.bold: true
   }

   Grid {
     id: colorPicker
     x: 4; anchors.bottom: page.bottom; anchors.bottomMargin: 4
     rows: 2; columns: 3; spacing: 3
     Cell { cellColor: "red"; onClicked: helloText.color = cellColor }
     Cell { cellColor: "green"; onClicked: helloText.color = cellColor }
     Cell { cellColor: "blue"; onClicked: helloText.color = cellColor }
     Cell { cellColor: "yellow"; onClicked: helloText.color = cellColor }
     Cell { cellColor: "steelblue"; onClicked: helloText.color = cellColor }
     Cell { cellColor: "black"; onClicked: helloText.color = cellColor }
   }
}
```

Criamos o seletor de cores colocando 6 células com diferentes cores em uma tabela.

```
Cell { cellColor: "red"; onClicked: helloText.color = cellColor }
```

Quando o signal clicked de nossa célula é disparada, precisamos atribuir a cor do texto para a cellColor passada como seu parâmetro. Podemos reagir a qualquer signal do componente através de uma propriedade de nome "onSignalName" (veja Manipuladores de sinais).

Tutorial QML 3 - States e Transitions

Neste capítulo, nós criamos este exemplo um pouco mais dinâmico pela introdução de states e transitions.

Queremos que nosso texto mova-se para a base da tela, rode e fique vermelho quando clicado.

Aqui está o código QML:

```
import QtQuick 1.0

Rectangle {
   id: page
   width: 500; height: 200
   color: "lightgray"

   Text {
      id: helloText
      text: "Hello world!"
      y: 30
      anchors.horizontalCenter: page.horizontalCenter
      font.pointSize: 24; font.bold: true

      MouseArea { id: mouseArea; anchors.fill: parent }

      states: State {
         name: "down"; when: mouseArea.pressed == true
         PropertyChanges { target: helloText
                           y: 160; rotation: 180
                           color: "red"
          }
      }

      transitions: Transition {
         from: ""; to: "down"; reversible: true
         ParallelAnimation {
            NumberAnimation { properties: "y,rotation"
                              duration: 500
                              easing.type: Easing.InOutQuad
            }
            ColorAnimation { duration: 500 }
         }
       }
   }

   Grid {
       id: colorPicker
       x: 4; anchors.bottom: page.bottom; anchors.bottomMargin: 4
       rows: 2; columns: 3; spacing: 3

       Cell { cellColor: "red"; onClicked: helloText.color = cellColor }
       Cell { cellColor: "green"; onClicked: helloText.color = cellColor }
       Cell { cellColor: "blue"; onClicked: helloText.color = cellColor }
       Cell { cellColor: "yellow"; onClicked: helloText.color = cellColor }
       Cell { cellColor: "steelblue"; onClicked: helloText.color = cellColor }
       Cell { cellColor: "black"; onClicked: helloText.color = cellColor }
   }
}
```

Passo a passo

```
states: State {
   name: "down"; when: mouseArea.pressed == true
   PropertyChanges { target: helloText; y: 160; rotation: 180; color: "red" }
}
```

Primeiro nós criamos um novo state para nosso elemento text. Este state será ativado quando o MouseArea for pressionado e desativado quando for solto.

O state down inclui um conjunto de mudanças de propriedades em relação ao state default (os itens como são inicialmente definidos no QML). Especificamente, nós setamos a propriedade y do texto para 160, a rotação para 180 e a cor para vermelho.

```
transitions: Transition {
     from: ""; to: "down"; reversible: true
     ParallelAnimation {
     NumberAnimation { properties: "y,rotation"
                       duration: 500
                       easing.type: Easing.InOutQuad
        }
        ColorAnimation { duration: 500 }
     }
}
```

Como não queremos que o texto apareça na base da tela instantaneamente, mas ao invés disse mova-se suavemente, adicionamos uma transition entre os dois states.

As propriedades from e to definem os states entre os quais a transition vai ser executada. Nesse caso, queremos uma transition do state default oara nosso state down.

Como queremos que a mesma transition seja executada ao contrário quando voltando do state down para o state default, setamos a propriedade reversible para true. Isto equivale a escrever as duas transitions separadamente.

O elemento ParallelAnimation garante que os dois tipos de animação (number e color) comecem ao mesmo tempo. Podeos também executá-las (as animações) uma após a outra usando SequentialAnimation ao invés de ParallelAnimation.

Para mais detalhes sobre states e transitions, veja States QML e o exemplo de states e transitions.

Layout Anchor-based em QML

Além dos mais tradicionais - Grid, Row e Column, QML também proporciona um modo de organizar itens usando o conceito de âncoras (anchors). Cada item pode ser imaginado como se tivesse um conjunto de sete linhas-âncoras invisíveis: left, horizontalCenter, right, top, verticalCenter, baseline e bottom.

A linha-base (não mostrada acima) corresponde à linha imaginária na qual o texto estará assentado. Para itens sem texto, ela é a mesma que "top".

O sistema de ancoragem QML permite-lhe definir relacionamentos entre as linhas-âncoras de diferentes itens. Por exemplo, você pode escrever:

```
Rectangle { id: rect1; ... }
Rectangle { id: rect2; anchors.left: rect1.right; ... }
```

Neste caso, a borda esquerda de rect2 é vinculada à borda direita de rect1, produzindo o seguinte:

O sistema de ancoragem também permite que você especifique margens e deslocamentos. Margens especificam a quantidade de espaço deixado para o lado de fora de um item, enquanto deslocamentos permitem que você manipule o posicionamento usando as linhas centrais de ancoragem. Observe que margens especificadas usando o sistema de ancoragem tem significado apenas para âncoras; elas não tem qualquer efeito quando usando outros layouts ou posicionamento absoluto.

O seguinte exemplo especifica uma margem esquerda:

```
Rectangle { id: rect1; ... }
Rectangle { id: rect2; anchors.left: rect1.right; anchors.leftMargin: 5; ... }
```

Neste caso, a margem de 5 pixels é reservada para a esquerda de rect2, produzindo o seguinte:

Você pode especificar âncoras múltiplas. Por exemplo:

```
Rectangle { id: rect1; ... }
Rectangle { id: rect2;
            anchors.left: rect1.right;
            anchors.top: rect1.bottom; ...
}
```

Especificando múltiplas âncoras horizontais ou verticais você pode controlar o tamanho de um item. Por exemplo:

```
Rectangle { id: rect1; x: 0; ... }
Rectangle { id: rect2
            anchors.left: rect1.right
            anchors.right: rect3.left
            ...
}
Rectangle { id: rect3; x: 150; ... }
```

Limitações

Por razões de performance, você só pode ancorar um item ao seus irmãos e pais diretos. Por exemplo, a segunte âncora pode ser considerada inválida e produziria uma advertência:

```
Item {
  id: group1
  Rectangle { id: rect1; ... }
}
Item {
  id: group2
  Rectangle { id: rect2; anchors.left: rect1.right; ... }    // âncora inválida
}
```

Introdução à linguagem QML

QML é uma linguagem declarativa projetada para descrever interfaces de usuário de um programa: sua aparência e como ele se comporta. Em QML, uma interface de usuário é especificada árvore de objetos com propriedades.

Esta introdução destina-se àqueles com pouca ou nenhuma experiência de programação. Javascript é usado como linguagem de script em QML, então você pode querer aprender um pouco mais sobre o assunto antes de mergulhar fundo no QML. Também ajuda ter um conhecimento básico de outras tecnologias web como HTML e CSS, mas não é necessário.

Sintaxe básica QML

QML parece com isso:

```
import QtQuick 1.0

Rectangle {
   width: 200
   height: 200
   color: "blue"

   Image {
       source: "pics/logo.png"
       anchors.centerIn: parent
   }
}
```

Objetos são especificados pelo seu tipo, seguido por um par de chaves. Tipos de objetos sempre começam com uma letra maiúscula. No exemplo acim, existem dois objetos, um Rectangle e um Image. Entre as chaves, podemos especificar informações sobre o objeto, como suas propriedades.

Propriedades são especificadas como propriedade: valor. No exemplo acima, podemos ver que Image possui uma propriedade chamada source, à qual foi atribuido o valor "pics/logo.png". A propriedade e seus valores são separadas por uma dois pontos.

Propriedades podem ser especificadas uma por linha:

```
Rectangle {
      width: 100
      height: 100
}
```

ou você pode colocar múltiplas propriedades em uma única linha:

```
Rectangle { width: 100; height: 100 }
```

Quando múltiplos pares propriedade/valor são especificados em uma única linha, eles devem ser separados por um ponto e vírgula.

A declaração import importa o módulo Qt, que contém todos os elementos padrão do QML. Sem esta declaração import, os elementos Rectangle e Image não estariam disponíveis

Expressões

Além de atribuir valores às propriedades, você pode também atribuir expressões escritas em Javascript.

```
Rotation {
    angle: 360 * 3
}
```

Estas expressões podem incluir referências a outros objetos e propriedades, caso em que um vínculo é estabelecido: quando o valor da expressão muda, a propriedade à qual a expressão foi atribuída é automaticamente atualizada para aquele valor.

```
Item {
    Text {
        id: text1
        text: "Hello World"
    }
    Text {
        id: text2
        text: text1.text
    }
}
```

No exemplo acima, o objeto text2 mostrará o mesmo texto do objeto text1. Se text1 mudar, text2 é automaticamente alterado para o mesmo valor.

Observe que para referir-se a outros objetos, usamos os valores de seus ids (veja abaixo mais informações sobre a propriedade id).

Comentários QML

Comentar em QML é semelhante ao JavaScript.

* Comentários de única linha começam com // e terminam no final da linha
* Comentários multilinhas começam com /* e terminam com */

Comentários são ignorados pelo *engine*. Eles são úteis para explicar o que você está fazendo; para referência futura ou para outros que leiam seus arquivos QML.

Comentários também podem ser usados para evitar a execução de código, o que às vezes é útil para rastrear problemas.

```
Text {
      text: "Hello world!"
      //opacity: 0.5
  }
```

No exemplo acima, o objeto Text será sua opacidade normal, posto que a linha opacity: 0.5 foi transformada em um comentário.

Propriedades

Nomeando propriedades

Propriedades começa com uma letra minúscula (com exceção das propriedades anexadas).

Tipos de propriedades

QML suporta propriedades de muitos tipos (veja Tipos básicos QML). Os tipos básicos incluem int, real, bool, string, color e lists.

```
  Item {
      x: 10.5             // uma propriedade 'real'
      ...
      state: "details"    // uma propriedade 'string'
      focus: true         // uma propriedade 'bool'
  }
```

Propriedades QML são o que é conhecido como type-safe. Isto é, elas apenas permitem atribuir um valor que combine com o tipo da propriedade. Por exemplo, a propriedade x do item é um real, e se você tentar atribuir uma string a ela, você obterá um erro.

```
  Item {
      x: "hello"  // ilegal!
  }
```

A propriedade id

Cada objeto pode receber uma propriedade especial e única chamada id. Nenhum outro objeto dentro do mesmo documento QML pode ter o mesmo valor de id. Atribuir um id permite que o objeto seja referido por outros objetos e scripts.

O primeiro elemento Rectangle abaixo tem um id, "myRect". O segundo elemento Rectangle define seu próprio tamanho (width) por referência a myRect.width, o que significa que ele terá o mesmo valor de width que o primeiro elemento Rectangle.

```
Item {
    Rectangle {
        id: myRect
        width: 100
        height: 100
    }
    Rectangle {
        width: myRect.width
        height: 200
    }
}
```

Observe que um id deve começar com uma letra minúscula ou um underscore e não pode conter outros caracteres que não sejam letras, números e underscores.

Propriedades List

Propriedades List parecem com isso:

```
Item {
    children: [
        Image {},
        Text {}
    ]
}
```

A lista é encerrada entre colchetes, com uma vírgula separando os elementos da lista. Em casos onde você esteja atribuindo um único item à lista, você pode omitir os colchetes:

```
Image {
    children: Rectangle {}
}
```

Propriedades Default

Cada tipo de objeto pode especificar uma de suas listas ou propriedades de objeto como sua propriedade default. Se a propriedade foi declarada como propriedade default, a tag da propriedade pode ser omitida.

Por exemplo, este código:

```
State {
    changes: [
        PropertyChanges {},
        PropertyChanges {}
    ]
}
```

pode ser simplificado para:

```
State {
    PropertyChanges {}
    PropertyChanges {}
}
```

porque changes é a propriedade default do tipo State.

Propriedades agrupadas

Em alguns casos, propriedades formam um grupo lógico e usam um ponto ou notação agrupada para demonstrar isso.

Propriedades agrupadas pode ser escritas assim:

```
Text {
    font.pixelSize: 12
    font.bold: true
}
```

ou assim:

```
Text {
    font { pixelSize: 12; bold: true }
}
```

Na documentação do elemento, propriedades agrupadas são mostradas usando a notação de ponto.

Propriedades anexadas

Alguns objetos anexam propriedades a outro objeto. Propriedades anexadas são da forma Tipo.propriedade, onde Tipo é o tipo de elemento que anexa a propriedade.

Por exemplo:

```
Component {
    id: myDelegate
    Text {
        text: "Hello"
        color: ListView.isCurrentItem ? "red" : "blue"
    }
}
ListView {
    delegate: myDelegate
}
```

O elemento ListView anexa a propriedade ListView.isCurrentItem para cada delegate que ele cria.

Outro exemplo de propriedade anexada é o elemento Keys que anexa propriedades para manipulação de teclas pressionadas para um item visual, por exemplo:

```
Item {
      focus: true
      Keys.onSelectPressed: console.log("Selected")
}
```

Manipulação de sinais (signals)

Manipuladores de sinais permite que sejam tomadas ações em resposta a um evento. Por exemplo, o elemento MouseArea possui manipuladores de sinais para manipular o pressionamento, soltura e click dos botões do mouse:

```
  MouseArea {
      onPressed: console.log("mouse button pressed")
  }
```

Todo manipulador de sinais começa com "on".

Alguns manipuladores de sinais incluem um parâmetro opcional, por exemplo o manipulador do sinal onPressed de MouseArea tem um parâmetro mouse:

```
MouseArea {
   acceptedButtons: Qt.LeftButton | Qt.RightButton
   onPressed: if (mouse.button == Qt.RightButton)
                 console.log("Botão direito do mouse pressionado")
}
```

O leitor Cárlisson Galdino escreveu:

Primeiro, acho que houve um equívoco. Uma biblioteca GPL não permite de certa forma linkagem, sem que interfira na licença do software que está sendo desenvolvido.

Para garantir espaço às bibliotecas livres no mercado, concorrendo com as privativas, foi criada a Library General Public License, que depois foi rebatizada para Lesser General Public License.

Posso estar enganado, mas acho difícil que uma biblioteca sob LGPL interfira na licença do software que a utiliza, caso se trate de meros includes. Seria bom você contactar alguém da FSF (o Alexandre Oliva, por exemplo) para informações mais precisas a respeito.

Gostei da iniciativa. Embora me pareça algo um tanto restrito.

Continue a linha "how-to simples" e vá evoluindo o tipo de aplicação. Como criar uma agenda de contatos, um cliente twitter, um editor de textos... Isso é muito bom pra quem está estudando!

Seria bom também um comparativo-não-tendencioso de recursos do Qt frente aos toolkits que existem: Gtk+, Fltk, wxWindow...

Outra seria uma série apresentando, em cada edição, um how-to similar ao que você fez, mas focando o uso do Qt em outra linguagem. Um para Python, na edição seguinte para Perl, etc... Seria uma forma de agradar também programadores fora do mundo C++ (ou que preferem outras linguagens).

Qt e Androide têm muito ou nada a ver?

Bem, era isso. Muito sucesso na iniciativa!

[]s,

Resposta do editor:

É verdade, Cárlisson, eu até publiquei um pedido público de desculpas com a explicação na segunda-feira (dia 20):

http://revistaqt.blogspot.com/2010/09/mea-culpa.html

A primeira edição foi bem limitada mesmo, porque tive que fazer tudo sozinho. Até as charges liberadas pelo Oliver da Geek and Poke eu tive que editar no Gimp pra traduzir o texto... :) Não tenho muita habilidade com a diagramação também. A idéia é que outros se empolguem e participem do projeto.

Com relação ao Android, existe um "port" do Qt para ele (http://gitorious.org/~taipan/qt/android-lighthouse), mas não é um projeto da Nokia.

Um abraço e muito obrigado.

O leitor Eduardo Pereira escreveu:

Olá André Vasconcelos,

Achei muito interessante seu trabalho com a revista porque são poucas informações em português sobre Qt que acho na internet. Gostei bastante da parte QT + PHP. Seguinte, algumas imagem estão com a resolução muito baixa e ficam com aparência ruim.

Até mais!!!!

Att,

Resposta do editor:

Oi Eduardo.

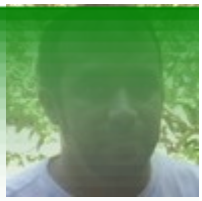
Você tem razão sobre a qualidade das imagens. Minha idéia foi reduzir a quantidade de páginas para aqueles que (como eu) preferissem imprimir a revista, mas no final das contas, olhando a versão impressa, algumas imagens não ficaram boas mesmo.

Vou melhorar isso na próxima edição, "espalhando" um pouco mais o conteúdo. Nesta primeira edição, acabei fazendo tudo sozinho e escaparam alguns detalhes.

Preciso muito desse tipo de retorno, para ir melhorando a revista.

Um grande abraço, Eduardo.

Distribuindo suas aplicações Qt para Linux



Por: André Vasconcelos
alovasconcelos@gmail.com

Dica para montar um pacote para distribuição de suas aplicações escritas em Qt

A história começa com um programador e sua aplicação em Qt pronta e testada. O próximo passo é distribuir a aplicação, provavelmente disponibilizando-a para download em algum servidor Web. Moleza!
No dia seguinte, nosso herói recebe o e-mail de alguém que fez o download do arquivo, mas ao tentar executar... nada!
O usuário seguiu as instruções e deu permissão de leitura para o arquivo, mas ao tentar executar, recebeu a seguinte mensagem:

```
error while loading shared libraries: libQtGui.so.4:
cannot open shared object file: No such file or directory
```

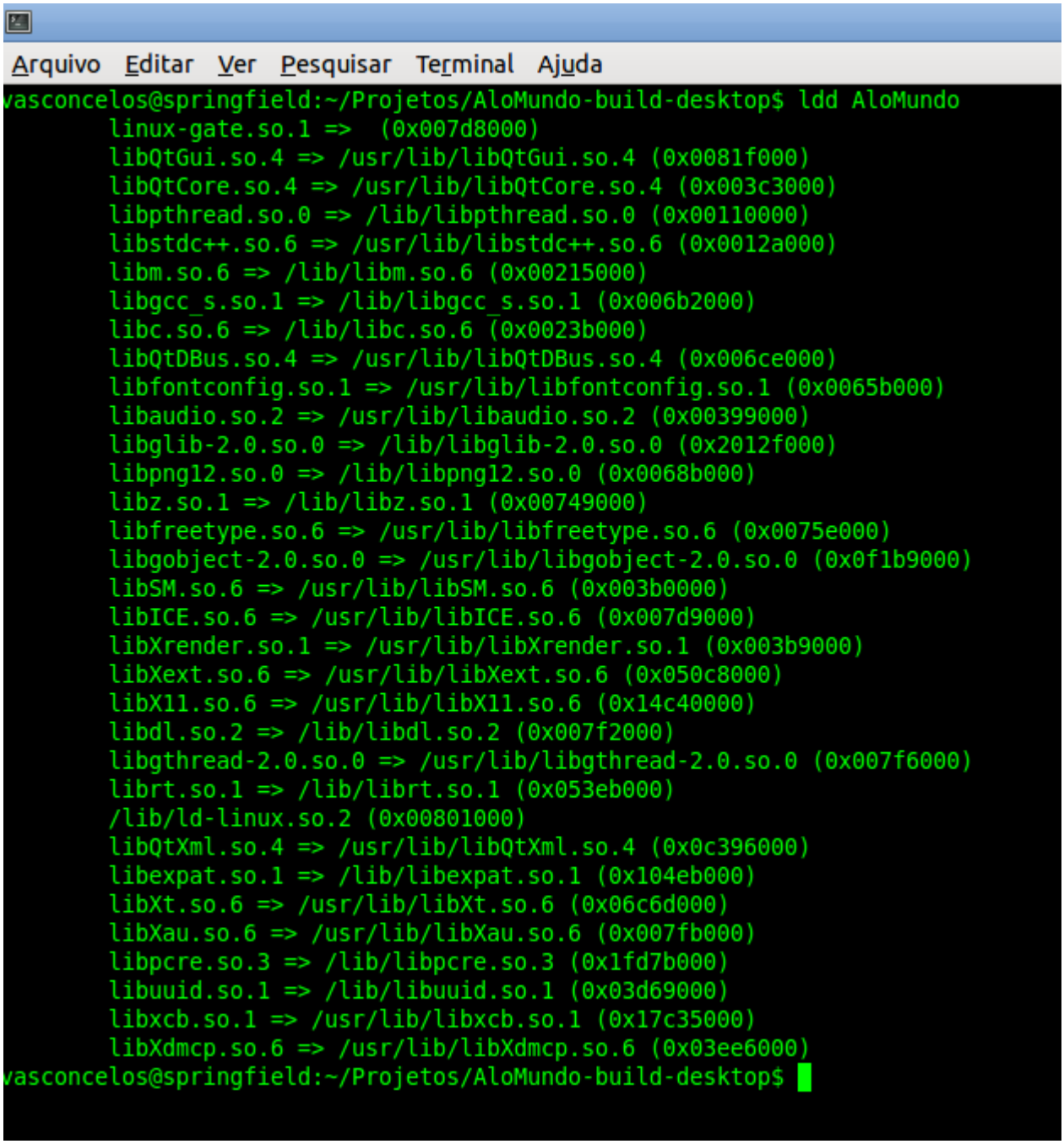
Estamos diante da típica situação de "na minha máquina funciona" . Acontece que na máquina de desenvolvimento funciona tudo porque normalmente temos um trilhão de bibliotecas instaladas nela.
O Qt nos dá duas opções de compilação de nossas aplicações: estática e dinâmica.
Para que possamos utilizar a compilação estática, o próprio Qt tem que ter sido compilado estaticamente. Acreditem... esta não à a mais simples das tarefas. Além de levar muitas horas pra compilar o Qt estaticamente, existem muitas dependências.
Com a compulação estática, temos a geração de um executável muito maior do que o gerado com a compilação dinâmica, porém com tudo embutido nele. Se o programador da nossa historinha tivesse usado esta forma de compilação, o executável disponibilizado para download teria funcionado.

O compilação *default* do Qt é a dinâmica. Com a compilação dinâmica temos executáveis menores, porque as bibliotecas utilizadas não são incluídas. Durante a execução dos programas, estes devem acessar arquivos externos correspondentes às bibliotecas necessárias. No caso do nosso desenvolvedor frustrado, a aplicação está reclamando que não conseguiu achar o arquivo libQtGui.so.4 que contém a biblioteca QtGui usada pelas aplicações em Qt quando incluímos o módulo **gui** no projeto, ou seja, em toda e qualquer aplicação com interface gráfica desenvolvida em Qt.

E tem mais... (parafraseando a Polishop) o programa interrompe a execução na primeira biblioteca que não localiza. Então na verdade, faltam mais arquivos do que o libQtGui.so.4.

Neste artigo veremos como montar um pacote com tudo que uma aplicação nossa em Qt precisa para ser executada no Linux. Em edições futuras veremos como proceder para outras plataformas.

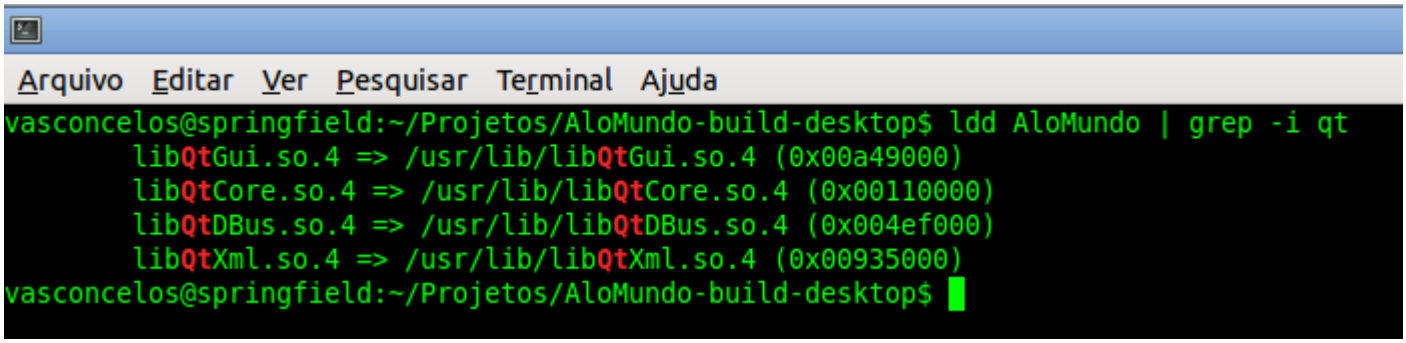
Para este artigo, vamos utilizar o programa Alô, Mundo! publicado na primeira edição da Revista Qt. Nosso primeiro passo será descobrir quais são as bilbliotecas utilizadas pelo executável de nossa aplicação, com o utilitário **ldd:**



33 bibliotecas pra um “Alô, Mundo!”? É o preço que pagamos pelas facilidades que o Qt nos proporciona. Mas não se desespere (ainda...) porque muitas destas bibliotecas estão presentes na instalação de qualquer distribuição atual de Linux.
O resultado do utilitário **ldd** em seu computador pode ser diferente do apresentado aqui, dependendo da sua distribuição Linux e da versão de Qt que esteja utilizando. Eu uso o Ubuntu 10.10 com o Qt 4.7 instalado via **Synaptic**.

Para testar a instalação de minhas aplicações em Qt, instalei uma máquina virtual Ubuntu rodando no Virtual Box. Esta máquina virtual é uma instalação “limpa” do Ubuntu sem as bibliotecas de desenvolvimento adicionais. Dessa forma, ela funciona como a máquina de um usuário que não seja programador.

A princípio podemos nos preocupar apenas com as bibliotecas do Qt, então o **ldd** pode ser executado em conjunto com o **grep** para filtrar apenas os arquivos que pertençam ao Qt:



Agora que sabemos de quais bibliotecas o programa AloMundo precisa para rodar, vamos copiar estes arquivos, juntamente com o arquivo executável AloMundo para um diretório chamado **AloMundoLinux** que usaremos para distribuir a aplicação.

Para indicar a localização das bilbiotecas, usamos devemos setar a variável de ambiente LD_LIBRARY_PATH antes de executar o programa. Agora vem o macete retirado do diretório do Google Earth: um script utilizado para setar a variável de ambiente e chamar o executável. Este script deverá estar no diretório AloMundoLinux.

```
#!/bin/sh
#
# Agente SAA
#

FindPath()
{
    fullpath="`echo $1 | grep /`"
    if [ "$fullpath" = "" ]; then
        oIFS="$IFS"
        IFS=:
        for path in $PATH
        do if [ -x "$path/$1" ]; then
               if [ "$path" = "" ]; then
                   path="."
               fi
               fullpath="$path/$1"
               break
           fi
        done
        IFS="$oIFS"
    fi
    if [ "$fullpath" = "" ]; then
        fullpath="$1"
    fi

    # Is the sed/ls magic portable?
    if [ -L "$fullpath" ]; then
        #fullpath="`ls -l "$fullpath" | awk '{print $11}'`"
        fullpath=`ls -l "$fullpath" |sed -e 's/.* -> //' |sed -e 's/\*//'`
    fi
    dirname $fullpath
}

CPATH="`FindPath $0`"
LD_LIBRARY_PATH=.:${CPATH}
export LD_LIBRARY_PATH

cd "${CPATH}/"

exec "./AloMundo" "$@"
```

Ainda não acabou... no caso do nosso pequeno AloMundo, não utilizamos imagens, apenas um label com o texto “Alô, Mundo!”, mas para aplicações onde sejam utilizadas imagens, devemos incluir os plugins para tratamento dos formatos utilizados. Os plugins dos formatos de imagens ficam no diretório qt/plugins/imageformats a partir do diretório onde esteja instalado o Qt.

Digamos que sua aplicação utilize imagens do tipo jpeg. Nesse caso você deve criar no diretório de sua aplicação um diretório chamado imageformats e copie para ele o arquivo qt/plugins/imageformats/libqjpeg.so para ele.
Com os arquivos da aplicação presentes no diretório AloMundoLinux, basta gerar um pacote, compactando-o. Pode-se utilizar, por exemplo, o utilitário **tar** para isso:

tar cvfz AloMundoLinux.tgz AloMundoLinux

Agora é só baixar o arquivo na máquina cliente, descompactar e executar a aplicação através do script start.

Caso a execução “reclame” da ausência de outra biblioteca, basta incluí-la no diretório AloMundoLinux e gerar o pacote novamente.

Uma vantagem da abordagem de compilação dinâmica é a possibilidade de atualizar as bibliotecas sem atualizar as aplicações.

Ok, você pode a essa altura estar imaginando: se eu tiver 10 aplicações em Qt terei em cada uma delas uma cópia das bibliotecas utilizadas? Não necessariamente. Você pode adotar como regra, a criação de um diretório na máquina do cliente para armazenar as bibliotecas e utilizar este diretório no seu script start. Alguns fabricantes de software usam a estratégia de criar um diretório com o seu nome e dentro deste diretório instalar seus produtos.

Neste caso, o script start setará a variável LD_LIBRARY_PATH para apontar para o diretório compartilhado onde estarão as bibliotecas.

Até a próxima.

Um excelente material sobre Qt encontra-se disponível no site de Antônio Márcio A. Menezes. Slides de seu mini-curso e o código-fonte utilizado. Imperdível para estudantes ou profissionais de Qt.

O mini-curso está disponível em:

http://antoniomenezes.net/?tag=mini-curso-c-qt

Revista Qt

Edição 2 - Novembro-Dezembro-2010

Editor
André Luiz de O. Vasconcelos

Cartoons
Oliver Widder (Geek & Poke)

Colaboradores
Pierre Andrade Freire
Rogério Etsuo Yamamaru

A revista e os aplicativos nela publicados foram criados utilizando o Linux Ubuntu, versão 10.10.

A Revista Qt foi criada e convertida para PDF no BrOffice.org Draw, versão 3.2.1.

As imagens capturadas das telas presentes na Revista Qt foram obtidas através do KSnaphot, versão 0.8.1.

O tratamento de imagens (como a tradução das charges do Geek & Poke) foi feito através do Gimp, versão 2.6.10.