# QtScraper – Uma Biblioteca Multiplataforma para WebScraping com Qt

Eliakin Costa
Instituto Federal da Bahia
Rua Emídio dos Santos, S/N, Barbalho
Salvador-Ba, Brasil
eliakin.costa@ifba.edu.br

Sandro Andrade
Instituto Federal da Bahia
Rua Emídio dos Santos, S/N, Barbalho
Salvador-Ba, Brasil
sandroandrade@ifba.edu.br

## RESUMO

O *web scraping* é uma alternativa comumente utilizada quando não deseja-se realizar o desenvolvimento completo de novas soluções para integração entre sistemas. Isso pode significar redução de tempo de desenvolvimento, burocracia e utilização de recursos das organizações, mas sua realização muitas vezes pode ser desafiadora, quando a plataforma de desenvolvimento em que se desejar integrar este tipo de solução não oferece uma solução pronta para *web scraping*. Um caso particular dessa dificuldade é o uso de *web scraping* no contexto do *toolkit* Qt. Neste contexto, este trabalho visa apresentar o QtScraper, uma biblioteca que auxilia no desenvolvimento de soluções de *web scraping* em Qt. Foram desenvolvidos componentes QML que abstraem a extração de dados de uma página Web e integração dentro do contexto do Qt. Para avaliar a solução foram realizados experimentos com e sem o QtScraper, visando analisar a produtividade e desempenho em cada cenário.

## Plavras-chave

*Web Scraping; Web; Qt; Mobile;*

## 1. INTRODUÇÃO

Ao longo dos últimos trinta anos, a *Web* se tornou cada dia ainda mais importante na vida das pessoas, sendo fonte de informação diária para muitas pessoas [1]. Entretanto, nos últimos dez anos ocorreu uma popularização dos *smartphones*, inclusive ultrapassando a quantidade de computadores de mesa (*desktop*) [2]. Essa popularização desencadeou uma alta demanda por aplicativos móveis desenvolvidos diretamente para cada plataforma. Atualmente, as pessoas passam mais tempo utilizando aplicativos do que navegadores de Internet [3]. Contudo, desenvolver uma mesma aplicação para diferentes plataformas é altamente custoso, o que fez com que várias soluções multiplataforma fossem desenvolvidas. Uma destas soluções é o *Qt* [4].

A partir deste cenário, ainda existem muitas aplicações Web desenvolvidas por terceiros que não possuem a sua versão *mobile*. Também existem casos de aplicações desenvolvidas por diferentes empresas, mas que possuem informações em comum que pode ser interessantes para o desenvolvimento de novas soluções. Um exemplo disso são *sites* para pesquisa de preço [5] [6], que funcionam como indexadores de uma série de outras aplicações *Web* do varejo, através da extração dos dados destas páginas. Essa técnica é chamada de *web scraping*. O *web scraping* hoje é muito popular em aplicações desenvolvidas em Python [7].

Apesar da popularidade do *web scraping* e do desenvolvimento de aplicações móveis usando *Qt*, hoje não existe uma solução pronta para o desenvolvimento de aplicações utilizando a técnica de *web scraping* para o *framework Qt*. Contudo, é possível através dos recursos do *Qt*, desenvolver uma solução deste tipo, apesar da complexidade.

Neste contexto, o presente trabalho tem como objetivo a implementação e avaliação de uma solução baseada em componentes para facilitar o desenvolvimento de soluções baseadas em *web scraping* com *Qt* para dispositivos móveis. Esta biblioteca, chamada QtScraper, realiza a abstração da complexidade do *web scraping* através de componentes declarativos e que possuem total integração com as capacidades multiplataforma do *Qt*. A solução requer a implementação de componentes *QML*, que são então convertidos em requisições HTTP de forma transparente. Estes componentes implementados permitem o acesso aos dados obtidos através das requisições de forma totalmente compatível com componentes nativos do *QML*.

Para avaliar a solução proposta neste trabalho, foram realizados experimentos comparativos entre dois cenários distintos: no primeiro, uma aplicação *mobile* originalmente desenvolvido apenas em *Qt* e *C++* para realizar o *web scraping* de páginas *Web*; no segundo, utilizou-se o QtScraper para realizar a refatoração de três funcionalidades desta mesma aplicação existente. Foram feitas análises com base em métricas de complexidade e desempenho para avaliar a efetividade da solução.

O restante deste trabalho está organizado como segue. A Seção 2 apresenta o material introdutório ao tema abordado nesse trabalho, tais como definição de termos utilizados e apresentação de conceitos necessários ao entendimento deste trabalho. A Seção 3 apresenta os trabalhos relacionados à solução proposta neste artigo. A Seção 4 apresenta a solução proposta e as tecnologias utilizadas no seu desenvolvimento. A Seção 5 apresenta a análise do estudo proposta, bem como considerações sobre os resultados. A Seção 6 descreve as conclusões obtidas, seguidas das limitações encontradas e os possíveis trabalhos futuros.

## 2. REFERENCIAL BIBLIOGRÁFICO

Esta seção apresenta os principais assuntos necessários ao entendimento deste trabalho. A Subseção 2.1 apresenta o conceito de *web scraping*. Visto que este trabalho tem como objetivo disponibilizar uma biblioteca para *web scraping* com Qt, a Subseção 2.2 irá apresentar esta solução que foi base para o desenvolvimento da biblioteca. É importante entender quais as possibilidades do Qt, quando se trata de

*web scraping*. A Subseção 2.3 tem como objetivo apresentar essas possibilidades, a fim de facilitar o entendimento da importância deste trabalho neste contexto.

## 2.1 O que é Web Scraping?

Segundo Ryan Mitchell (2018), "*web scraping* é o ato de obter, processar e formatar dados de páginas *Web* por meio de agentes não-humanos". O conceito de *web scraping* não é recente na Internet. Ele também é conhecido como *screen scraping* [8] ou mineração de conteúdo *Web* [9]. Apesar da simplicidade da definição, trata-se de uma área multidisciplinar, por envolver engenharia de *software*, ciência de dados e legislação de direitos autorais.

Quando fala-se de *web scraping*, é preciso entender a sua importância e os contextos em que se faz útil. Se a única maneira de acessar um conteúdo é através de um *browser*, você está limitado ao tamanho da sua tela, uma página de cada vez. *Scrapers* podem ser ferramentas poderosas para processar milhares de páginas e suas informações em uma fração reduzida de tempo, quando comparado com o ser humano. Além disso, soluções de *web scraping* podem mostrar informações que vão muito além dos resultados de uma *engine* de busca como o *Google*, podendo exibir análise sobre os dados de páginas semelhantes e até mesmo previsões. Um caso de uso interessante seria descobrir qual a melhor data de compra de um produto, com base em análise feita, ou apenas saber se um produto está de fato em promoção ou não de acordo com a evolução dos preços.

Todas as informações descritas poderiam ser obtidas através de uma *API*, mas como demonstrado por Ryan Mitchell (2018), pode não ser interessante pelos seguintes motivos: *i)* quando precisa-se acessar dados de muitos sites diferentes que não possuem uma *API* coesa; *ii)* os dados requeridos não são suficientes para justificar a existência de uma *API*; *iii)* a fonte não possui uma estrutura ou capacidade técnica para disponibilizar uma *API*.

Mesmo quando uma *API* existe, ela pode ser totalmente incompatível com os requisitos desejados. É neste momento que o *web scraping* se faz importante, pois se a informação é acessível através de um *browser*, na maioria dos casos será possível acessar através de um *script*.

Estes obstáculos são bem conhecidos pela indústria e partindo de uma análise baseada em [10, 9, 7, 11], podem ser divididos em quatro tarefas principais:

- **Projeto e Análise:** é uma parte muito importante do processo, por decidir as estratégias mais importantes de *web scraping* para um determinado contexto. Dentre essas estratégias, destacam-se: a frequência com que os dados serão lidos, quantidade de itens que serão coletados e possibilidade combinação de dados diferentes e de fontes diferentes no mesmo contexto.

  No que diz respeito a frequência com que os dados serão obtidos, é importante pensar na necessidade de cada aplicação e o intervalo em que os dados da fonte serão atualizados. Essa decisão terá impacto na carga aplicada sobre os servidores da fonte. *Web scraping* para passagens de avião geralmente exige uma atualização mais constante do que a consulta a cardápios de restaurante por exemplo.

  A quantidade de itens também é importante, dependendo do objetivo que se deseja alcançar. A evolução do preço de um produto irá exigir uma quantidade de itens menor do que a comparação de preço de um segmento inteiro de itens.

  Um dos pontos mais importantes do *web scraping* é fazer agrupamento de dados semelhantes a partir de fontes diferentes. O desafio é criar padrões de busca com seletores para cada fonte de dados. Um exemplo muito comum é a busca por passagens aéreas de um trecho específico. As pessoas querem saber os preços mais baixos comparando a maior quantidade possível de companhias.

- **Acesso aos recursos:** geralmente a comunicação com páginas é estabelecida através do protocolo HTTP. Os comandos HTTP mais comumente utilizados para *web scraping* de dados são *GET* (obtenção de dados) e *POST* (envio de dados). O acesso ao conteúdo das páginas é um processo difícil, devido a cuidados com sobrecargas de servidores, banimento de *IPs*, autenticação e mesmo quanto à legalidade do acesso a determinado site ou informação.

  Quando trata-se de sobrecarga, deve-se tomar cuidado com a quantidade de requisições feitas para não tornar indisponível um determinado site. Geralmente quando uma página é excessivamente acessada à partir de um mesmo IP, existe um mecanismo de segurança que adiciona este IP em uma lista de endereços IP bloqueados. Isso torna difícil a execução de milhares de requisições em um pequeno intervalo de tempo, fazendo-se necessário o uso de servidores *proxy*, que não é nada mais que uma outra máquina que irá efetivamente realizar a requisição para a sua aplicação. Dessa forma, o IP real da sua aplicação permanece desconhecido do site desejado. É possível visualizar na Figura 1 como funciona esse tipo de mecanismo. O uso de *proxies* levanta um ponto importante sobre a legalidade do *web scraping*.

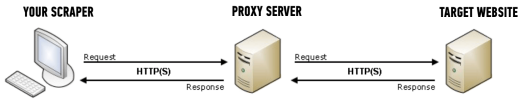


Figura 1: Exemplo de funcionamento de um servidor *proxy* [12]

  Como abordado em [7], as restrições de *web scraping* podem variar para cada país, mas em geral seguem se algumas premissas básicas: fazer mais requisições em horários de menor pico de acessos, respeitar as restrições descritas no *robots.txt* (arquivo que descreve quais páginas não podem ser acessadas por robôs em um determinado site por meio de uma especificação criada pelo *Google*) e a depender da carga, pode ser necessário comunicar o mantenedor.

- **Seleção dos dados:** uma vez que tenha-se obtido a página *HTML*, a próxima etapa é decidir como acessar os dados desejados. É possível executar esta tarefa por meio de expressões regulares [13] ou por meio de criação de estrutura de árvore a partir da página obtida. Estes soluções são chamadas de seletores. *XPath* ou CSS são os mais utilizados para acessar os dados com maior facilidade.

- **Implantação e operação:** Manter uma aplicação *web scraping* é um desafio tão grande quanto implementar. Deve-se preocupar o tempo todo em monitorar a disponibilidade das fontes de dados e garantir que a aplicação permaneça resiliente o suficiente para contornar esse tipo de problema. Outro monitoramento que precisa ser feita é quanto a mudanças de variáveis, pois páginas *Web* mudam o tempo todo. Pior do que não fornecer informações, é prover dados incorretos. Isso pode ser catastrófico para a experiência do usuário.

  Isso tudo exige um esforço de desenvolvimento enorme. Os custos de desenvolvimento e manutenção devem ser muito bem planejados. Garantir resiliência e consistência deste tipo de serviço é extremamente desafiador.

## 2.2 O Toolkit Qt

O Qt [4] é um *toolkit* multiplataforma desenvolvido em C++ [14], desenvolvido e mantido pela *The Qt Company* [15]. O Qt disponibiliza um conjunto vasto de funcionalidades, tais como: componentes para interface gráfica, *Web engine*, armazenamento de dados, ferramentas de desenvolvimento, *APIs mobile*, etc.

O Qt foi concebido por *Haavard Nord* and *Eirik Chambe-Eng* em 1990. Os dois trabalhavam juntos em um projeto de uma aplicação para ultrassom de imagem. Este *software* teria que funcionar no *Windows*, *macOS* e *Unix*. A partir dessa ideia foi criada uma empresa chamada *TrollTech*. Em maio de 1995 foi feito o lançamento da primeira versão do *toolkit* pela empresa recém fundada. Em 1998 foi lançado o KDE [16], uma interface gráfica para *Unix* totalmente baseada em Qt. O KDE acabou se tornando um dos casos de maior sucesso do Qt e com uma comunidade ativa de desenvolvedores que passou a participar da evolução do *toolkit*. Em 2008, a *TrollTech* foi adquirida pela Nokia [17]. A Nokia posteriormente mudou suas estratégias de plataformas e vendeu o Qt para a *Digia* [18]. A partir de 2014, o Qt passou a ser administrado pela *The Qt Company*, subsidiária da *Digia*. Em 2016, as duas empresas se separaram e a *The Qt Company* passou a deter os direitos sobre o Qt. A história relatada está baseada em [19].

O Qt hoje possui mais de um milhão de desenvolvedores, sendo utilizado por empresas do mundo todo [20]. O Qt suporta diversas plataformas [21], incluindo plataformas *mobile*, tais como *Android* e *iOS*. O poder do Qt está em compilar e executar aplicativos em todas essas plataformas a partir do mesmo código-fonte e com *look-and-feel* nativo.

Para que os aplicativos possuam o visual de aplicações nativas, ele utiliza de componentes QML. O QML é uma linguagem declarativa que permite desenvolver interfaces baseadas em seus componentes visuais e como eles se relacionam. O QML baseou-se em conceitos de hierarquia e simplicidade de outras linguagem declarativas como CSS e JSON, assim como pode ser visto na Listagem 1. O QML possui outras características que fazem dele uma opção interessante para o desenvolvimento. Primeiro, a sua curva de aprendizado é menos acentuada quando comparada com linguagens como C++ e até mesmo linguagens declarativas como *HTML* [22]. Segundo, a possibilidade de utilizar a sua sintaxe declarativa junto com programação imperativa, através do JavaScript. Por último, é possível combinar toda a simplicidade que essa linguagem declarativa fornece junto com a performance do

```
import QtQuick 2.0

Rectangle {
    id: page
    width: 320; height: 480
    color: "lightgray"

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

Listagem 1: Exemplo de arquivo QML.

C++ através da exposição de *APIs* para o QML. Trata-se de uma solução com foco em rapidez, reuso e performance.

Um dos mecanismos mais importantes da relação do QML com o C++, assim como do Qt como um todo, é o conceito de *signals* e *slots* [23]. *Signals e slots* são usados para a comunicação entre objetos e esse mecanismo é uma funcionalidade central do Qt e que o difere de outros *toolkits*.

Quando um *widget* (elemento visual) é modificado, muitas vezes existem outros elementos visuais que precisam ser notificados. Um exemplo de caso de uso seria o *click* de um botão chamado "Fechar" que deseja que uma função chamada *"close"* da janela atual seja executada.

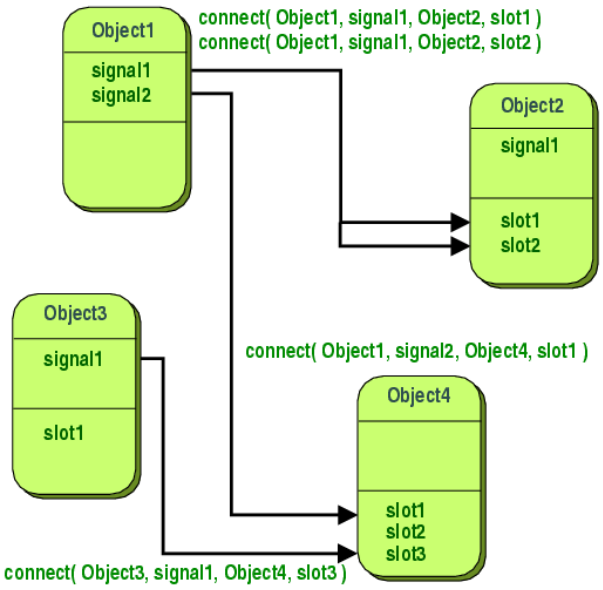


Figura 2: Representação do mecanismo de *signals* e *slots*.

A Figura 2 representa o funcionamento do mecanismo de *signals* e *slots*. Um *signal* é uma notificação que pode ser emitida toda vez que um objeto muda seu estado. O Qt identifica os objetos que devem ser notificados através de um método chamado *connect*, que liga um *signal* à um ou mais *signals* ou *slots*. Os *slots* tratam-se de métodos que irão executar uma determinada ação, como o exemplo citado an-

teriormente do fechamento de uma janela. O que torna a comunicação desacoplada é o fato de que os *slots* desconhecem os *signals* que estão conectados a ele, isso garante que componentes são realmente independentes. Este mecanismo permite que componentes da interface atualizem automaticamente os dados, o que associado com a linguagem QML, torna o desenvolvimento mais simples e totalmente desacoplado.

Apesar do Qt ser uma solução bastante madura, ainda não existe uma biblioteca de *web scraping* para o *toolkit*, seja para desenvolvimento *desktop*, embarcado ou *mobile*.

## 2.3 Web scraping com Qt

O Qt é uma solução voltada principalmente para o desenvolvimento de interfaces gráficas. Ao analisar o *toolkit*, nota-se uma vasta gama de componentes dos mais variados tipos, mas nenhum deles especificamente direcionados para *web scraping*. Entretanto, ao realizar uma análise mais profunda, é possível perceber que existem recursos suficientes para conseguir realizar a extração de dados de páginas *Web*.

Como abordado anteriormente, o Qt possui suporte nativo para desenvolvimento C++, JavaScript e QML. Através dos recursos disponíveis para essas linguagens foi possível identificar duas alternativas para o problema de *web scraping*. A primeira utilizando C++ e uma biblioteca chamada *QNetworkAccessManager* [24]. A segunda com uma biblioteca nativa do JavaScript chamada de *XMLHttpRequest* [25] e componente QML chamado de *XmlListModel* [26].

A *API* de rede do Qt é toda construída a partir do objeto *QNetworkAccesManager* [24]. Este objeto mantém as configurações necessárias para realizações de requisições e obtenção das respostas. Todas as requisições feitas a partir dele serão respondidas de forma assíncrona. Estas respostas serão obtidas através do mecanismo de *signals* e *slots*. Na Listagem 2, é possível visualizar um exemplo básico de como funciona o processo para realização de uma requisição HTTP e acesso do *HTML* . É possível refatorar essa implementação básica para que se possa receber uma *URL* de forma parametrizada.

```
QNetworkAccessManager *manager =
    new QNetworkAccessManager(this);
connect(
    manager, &QNetworkAccessManager::finished,
    this, &MyClass::replyFinished
);

manager->get(QNetworkRequest(
    QUrl("http://qt-project.org")
));
```

Listagem 2: Exemplo de uso do QNetworkAccessManager no Qt

Após obter o *HTML*, é necessário verificar se as suas *tags* estão bem formadas, como todo *XML* exige, o que é algo incomum. A maioria das páginas visitadas durante o estudo precisariam ser convertidas antes de fazer uma busca na sua estrutura. A Listagem 3 mostra como fazer essa busca com *XPath* em um *HTML* que é um *XML* válido.

A segunda opção disponível é através de requisições utilizando o objeto *XMLHttpRequest*, objeto este que é criado por meio do JavaScript. Através dele é possível realizar uma

```
QString html = "<html><body><h1>My First Heading"
" </h1><p id=\"test\">My first paragraph.</p>"
" </body></html>";

QXmlQuery query;
query.setFocus(html);
query.setQuery("//*[@id=\"test\"]");

QString result;
query.evaluateTo(&result);
```

Listagem 3: Exemplo de uso do QXmlQuery no Qt

requisição para uma página *Web* e obter seu conteúdo como texto. Na Listagem 4 é apresentado um exemplo de como realizar essa requisição e envio de dados através do método HTTP *POST*.

```
var httpRequest = new XMLHttpRequest();
httpRequest.setRequestHeader(
    'Content-type',
    'application/x-www-form-urlencoded'
);

function get(url) {
    httpRequest.open("GET", url);
    httpRequest.send()
}

function post(url, params) {
    httpRequest.open("POST", url);
    httpRequest.send(params)
}
```

Listagem 4: Exemplo de uso do XmlHttpRequest no Qt

Supondo que o *HTML* é um *XML* válido, é possível realizar a conversão e obter os dados que serão exibidos na interface. No trecho 5, o *XmlListModel* carrega o *XML* e a *query* que precisa ser feita no *HTML*. Os objetos *XmlRole* serão os itens retornados dentro de cada padrão definido no *XmlListModel*.

```
XmlListModel {
    id: xmlModel
    xml: "<html>
            <body>
               <h1>My First Heading</h1>
               <p>My first paragraph.</p>
               <p>My second paragraph.</p>
            </body>
          </html>"
    query: "/html/body/p"

    XmlRole {
        name: "paragraph"
        query: "string()"
    }
}
```

Listagem 5: Exemplo de uso do XmlListModel no Qt

O problema das duas implementações disponíveis no Qt é que elas dependem de documentos *HTML* bem formatados,

o que não é o caso de muitas páginas na *Web*. Uma vez que se tenha uma página bem formatada, só é possível acessar o conteúdo por meio de um seletor *XPath* ou através de expressões regulares.

# 3. TRABALHOS RELACIONADOS

Com o objetivo de buscar trabalhos relacionados à solução aqui proposta, foram encontrados algumas ferramentas utilizadas na indústria. Estes projetos são muito conhecidos na área de *web scraping*. O objetivo desta seção é apresentar as principais soluções da indústria para resolver o problema de *web scraping*, detalhando o contexto em que essas soluções são utilizadas. Ressaltando que essa análise foi feita com base em artigos publicados por empresas atuantes na área [27][28]. A Tabela 1 apresenta uma análise comparativa das principais característica das soluções mais populares e a biblioteca desenvolvida para este trabalho.

A primeira ferramenta analisada é a *Scrapy* [29]: um *framework open source* para extração de dados. Esta solução disponibiliza funcionalidades, tais como: *proxies*, enfileiramento de *requests*, escalabilidade, etc. É a solução mais utilizada pela comunidade hoje e que possui mais conteúdo disponível na Internet. Por ser uma ferramenta mais complexa voltada para desenvolvedores, possui uma curva maior de aprendizagem.

A ferramenta seguinte é a *pyspider* [30]: é um *application framework* para construção de *web crawlers* [10] que executa *scripts Python*. Esta solução fornece uma interface fácil de usar para editar os *scripts*, monitorar tarefas em andamento e visualizar os resultados. O *pyspider* é uma solução complexa e sofre com o mesmo problema da uma curva de aprendizado mais íngreme e a sua comunidade ainda é muita pequena.

A terceira solução encontrada é o *Cheerio* [31]. O *Cheerio* é uma biblioteca JavaScript desenvolvida a partir do núcleo do *jQuery* [32] e desenhada especificamente para o lado do servidor. Esta solução é especializada em obter o HTML de uma determinada página e converter em conteúdo acessível através da sintaxe *jQuery*. Por conta de seu escopo mais reduzido, esta é uma ferramenta muito mais simples de aprender a utilizar e que se baseia em uma biblioteca muito conhecida na comunidade JavaScript. Entretanto, ela não executa o JavaScript da página ou mesmo obtém qualquer conteúdo atualizado via AJAX [33]. Seletores *XPath* [34] também não são suportados pelo *Cheerio*.

A *Beautiful Soup* [35]. Semelhante ao *Cheerio*, trata-se de uma biblioteca *Python* para conversão de documentos *HTML* e *XML* (incluindo documentos mal formados) em uma estrutura mais facilmente acessível. Uma estrutura de árvore é criada a partir da conversão. Esta ferramenta destaca-se por conta de sua baixa curva de aprendizado e resiliência diante de estruturas mal formatadas. Acaba sofrendo dos mesmos problemas da solução *Cheerio*.

A *Scraper API* [36], que tem como objetivo apoiar a construção de ferramentas para *web scraping*. Esta *API* lida com toda a infraestrutura necessária para obter uma página, como: *proxies*, *captcha*, etc. Esta solução utiliza uma tecnologia chamada de *headless browser* [37]. *Headless browsers* são navegadores sem interface gráfica que garantem a execução de JavaScript da página, dessa forma, conseguem acessar quaisquer alterações feitas via *AJAX*. Por tratar-se de uma ferramenta paga, possui muitas limitações em relações ao número de requisições que podem ser realizadas. Essa limitação pode ser um complicador muito grande para soluções com orçamento mais baixo. A documentação ainda é muito insuficiente o que dificulta desenvolver soluções mais customizadas.

A *ParseHub* [38], que é voltada para profissionais que trabalham com análise de dados, mas que não programam. É uma ferramenta web que permite que você ensine como fazer *web scraping* de um site a partir de sua interface. A partir destas seleções, são criadas associações. O resultado final pode ser obtido a partir de uma API RESTful com dados transitados em formato JSON. O *ParseHub* lida com os principais problemas de *web scraping* e oferece o processamento desses dados para um formato muito mais acessível. Apesar de suas qualidades, não oferece opções de customização para estender as suas funcionalidades, possui pouca documentação e *requests* limitados de acordo com pagamento.

O *Apify* [39] possui uma proposta semelhante a *Scrapper API*. Esta solução consegue resolver os todos os problemas resolvidos pela *Scrapper API* com o adicional de uma boa documentação e agendamento de tarefas. Entretanto, é uma ferramenta paga com preços acima das outras e que possui uma *API* complexa para desenvolvedores.

A partir do que foi apresentado nesta seção, o objetivo deste trabalho foi propor uma solução voltada para *Qt* e o nicho de aplicações híbridas. Aplicativos híbridos são aplicações multiplataforma que possuem o mesmo código-fonte, portanto o desenvolvedor apenas precisará manter uma único aplicativo. Hoje o *Qt* não fornece uma solução voltada para o problema de *web scraping*. A partir disso, a ideia desse projeto foi fornecer um conjunto de componentes para o Qt, que tornem o *web scraping* uma tarefa mais simples neste contexto. A arquitetura proposta não estende-se ao tratamento de mudanças nas páginas realizadas através da linguagem JavaScript.

# 4. QTSCRAPER

Nesta seção será apresentado o QtScraper, uma biblioteca multiplataforma para para *web scraping* de páginas *Web*. A Subseção 4.1 apresenta as informações e diagramas da arquitetura implementada. Em seguida, a Subseção 4.2 elucida, de forma detalhada, a utilização dos componentes da biblioteca nos principais casos de uso. Por fim, a Subseção 4.3 apresenta os aspectos de implementação da biblioteca.

## 4.1 Arquitetura do QtScraper

A solução apresentada neste trabalho visa apresentar uma solução mais simples e completa para desenvolvimento de aplicações *mobile* baseadas em *web scraping* de páginas da *Web*. A definição da localização e de quais dados serão extraídos das respectivas páginas é feita através da linguagem *QML*.

A Figura 3 apresenta o fluxo de execução do QtScraper. A partir da etapa "Execução de *requests*", todo o processamento é feito de forma assíncrona. A seguir, tem-se uma breve descrição de cada uma dessas etapas:

1. **Carregamento de objetos**: que acontece no momento em que o elemento *QML* é criado. O seu processamento é síncrono com o objetivo de garantir consistência das informações antes das próximas etapas;
2. **Execução de requests**: que enfileira e processa as requisições. Esta etapa será mais detalhada durante a Subseção 4.3;

| | Baixa complexidade | Requests Ilimitados | Proxies | AJAX | Autenticação | CSS | XPath | UI |
|---|---|---|---|---|---|---|---|---|
| Scrapy | - | X | X | - | X | X | X | - |
| pyspider | - | X | | X | - | X | X | X |
| Cheerio | X | - | - | - | - | - | X | - |
| Beautiful Soup | X | - | X | - | - | - | X | - |
| Scraper API | - | - | X | X | X | - | - | - |
| ParseHub | X | - | X | X | X | - | - | X |
| Apify | - | - | X | X | X | - | X | X |
| QtScraper | X | X | - | - | X | - | X | X |

Tabela 1: Análise dos trabalhos relacionados

3. **Conversão de HTML em XML**: que é um dos diferencias da biblioteca e que permite que *HTMLs* mal formatados sejam processados;
4. **Extração dos dados**, onde os dados a serem extraídos são localizados e convertidos em estrutura compatível com a biblioteca e o *QML*;
5. **Disponibilização dos dados**: é feita de forma transparente para o desenvolvedor que apenas precisa se preocupar com a forma como irá exibir as informações.


Figura 3: Fluxo de execução do QtScraper.

O QtScraper é uma biblioteca que adota dois padrões de arquitetura principais, sendo o primeiro estilo *publisher-subscriber* [40], em que objetos interessados em eventos se registram e aguardam notificações de mudança de estado. O segundo é padrão scheduler-agent-supervisor [41], utilizado para executar uma série de ações assíncronas, mas garantindo ordem e tratamento de falhas.

A Figura 4 representa o ciclo de processamento dos requests, desde a inicialização dos componentes até o término da execução das requisições, sendo:

- ***UI Component:*** componentes *QML* que irão utilizar das funcionalidades da biblioteca para fazer *web scraping* dos dados;
- ***In-memory state:*** após os componentes da biblioteca serem inicializados, o estado deles é salvo em memória. Esse estado pode ser acessado a qualquer momento por qualquer elemento da interface;
- ***Scheduler* ou Escalonador:** garante a ordem das requisições e compartilhamento de contexto entre todas as etapas da execução do fluxo de requisições;
- ***Supervisor:*** é responsável pelo controle de status das operações do *Scheduler*, assim como por disparar sinais das mudanças feitas para os *UI Components*;
- ***Http Agent:*** cada passo do *Scheduler* é representado por este objeto que faz requisições para páginas *Web*. Cada *Agent* contém lógica para tratamento de falhas, redirecionamentos e *retry*.

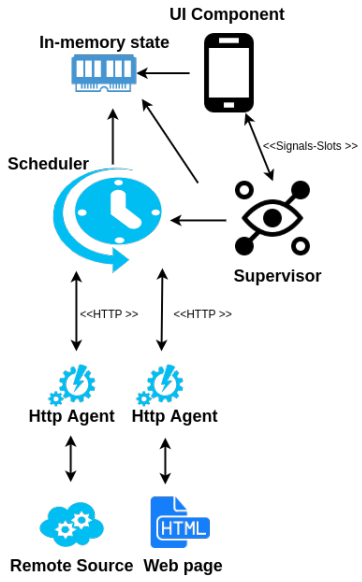


Figura 4: Arquitetura de execução dos requests.

A Figura 5 apresenta um diagrama de sequência do QtScraper, demonstrando o fluxo de controle entre os objetos

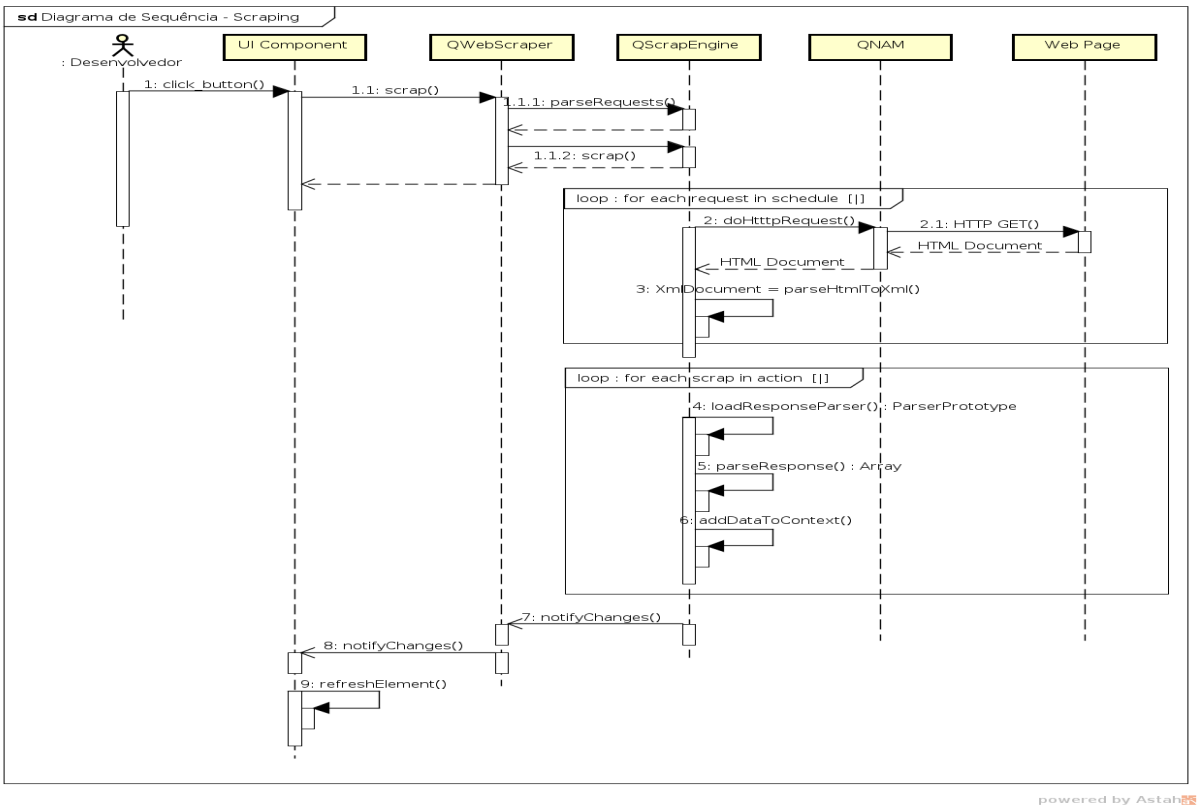


Figura 5: Diagrama de Sequência: *web scraping* de página *Web*.

durante a execução de um *web scraping* de uma página *HTML* na *Web*. O ator, que neste caso será o desenvolvedor, inicia a ação ao realizar o clique de um botão que acionará o método *scrap*.

O `UIComponent` executa a função `scrap` de `QWebScraper` que delegará à `QScrapEngine` o *parsing* dos objetos em modelos de requisição HTTP, através de `parseRequests` (síncrono). Em seguida é iniciado a execução do requisições através do método *scrap* (assíncrono).

O Escalonador em `QScrapEngine` será responsável por executar de forma ordenada os *requests* por meio de `QNetworkAccessManager` (`QNAM`).A classe `QNAM` permite que uma aplicação envie *requests* e receba *replies* através da rede. Para cada *request*, texttt`QNAM` irá retornar um documento *HTML* que é então convertido em *XML* pela `QScrapEngine`.

Essa conversão do documento HTML em *XML* é feita através de uma biblioteca escrita na linguagem C, que é chamada de *libtidy* [42]. Essa biblioteca foi escolhida por conta da fácil compatibilidade com C++, que foi a linguagem utilizada para desenvolvimento do Qt e do QtScraper. Foi apenas necessário compilar o *libtidy* para as plataformas *mobile*.

O objeto da classe *QScrapEngine* irá em seguida processar todos os pares de *name* e *query* (*scraps*), utilizando o devido *parser* para cada um. Cada conjunto de dados encontrados é então convertido em um objeto serializável e adicionado ao contexto compartilhado da solução. Após a finalização do processo anterior, é emitido um sinal avisando que existem mudanças nos objetos de contexto e/ou do *QWebScraper*. Todos os componentes da interface conectado ao *QWebScraper* irão atualizar os seus dados posteriormente.

## 4.2 Definições de utilização da biblioteca

Os arquivos *QML* do projeto que utiliza o QtScraper deverão conter pelo menos um objeto *QWebScraper* para cada fluxo de requisições que precisa ser executado em sequência. Além disso, são necessárias informações sobre a localização da página *Web*, bem como indicações de quais valores precisam ser extraídos e como deseja-se formatar os dados. A seguir, serão descritos os componentes que precisam ser utilizados e uma breve descrição da sua utilidade. Os atributos estão organizados na mesma estrutura em que precisam ser declarados.

- `QWebScraper:` objeto que representa um conjunto de requisições HTTP que precisam ser executadas na order definida pelo desenvolvedor.
  - ***id:*** indica o identificador do objeto no contexto do arquivo *QML*. É interessante usar o identificador para que não existam conflitos entre diferentes objetos do tipo *QWebScraper* e outros tipos que possam conter atributos com mesmos nomes.
  - ***keepAlive:*** indica se a sessão será mantida ou não após a realização do *scraping*. É muito comum precisar utilizar esse recurso em casos em que o desenvolvedor não quer pedir as credenciais do usuário toda vez que ele abrir o aplicativo, assim como nos navegadores. Esta propriedade é opcional e tem o valor padrão *false*.

  - ***actions:*** deverá conter uma lista de objetos do tipo *QWebScraperAction*, com um registro para cada requisição HTTP na ordem em que precisa ser feita. É importante salientar que tratamento de erros, redirecionamento e novas tentativas são transparentes para o desenvolvedor.
  - ***onStatusChanged:*** é um *signal handler* para o *signal statusChanged*. Sinal este que é disparado toda vez que ocorrem mudanças no estado atual das requisições (*actions*). Atualmente existem três mudanças de estado que são disparadas: Início do processamento das *actions* (*Loading*), falha ao processar a lista de requisições (*Error*) e o último para quando todas as requisições são processadas com sucesso (*Ready*).

- ***QWebScraperAction*:** objeto que representa um uma requisição HTTP para uma página *Web*. Este componente terá toda a informação para extração dos dados.
  - ***endpoint*:** representa a URL para a página *Web* de qual uma ou mais dados serão extraídos. Este valor é obrigatório.
  - ***method:*** *string* para o comando HTTP da requisição. Os valores aceitos são *GET* e *POST*. *GET* é o comando padrão, caso nenhum valor seja fornecido.
  - ***headers:*** um dicionário que contém um conjuntos de pares – chave e valor – que serão então convertidos em *HTTP headers* para a requisição. Este campo é opcional.
  - ***data:*** lista de dicionários que serão processados e enviados no corpo do *request*. Este campo é esperado quando *method = "POST"*. O valor padrão é uma lista vazia.
  - ***scraps:*** propriedade que dita quais dados e como eles serão extraídos da página *Web*.
    - ***name:*** nome da propriedade que será salva no contexto compartilhado entre todos os objetos *QWebScraper*.
    - ***query:*** query *XPath* que indica o caminho para encontrar um ou mais elemntos em um documento *HTML*.
    - ***responseParser:*** propriedade do tipo de conversão dos dados. Atualmente aceita o dois valores: *DefaultParser* para operações mais comuns de extração e *TableParser* para transformar tabelas em lista de dicionários compatíveis com modelos esperados por elementos *QML*.
    - ***indexes:*** exclusivo para quando for utilizado um *responseParser* do tipo *TableParser*. Ele determina quais índices de colunas ou linhas serão acessados.
    - ***headers:*** exclusivo para quando for utilizado um *responseParser* do tipo *TableParser*. Define um apelido para cada lista de valor retornado por cada índice.
  - ***validator:*** trata-se de um dicionário semelhante ao utilizado em *scraps*, mas com o único propósito de validar se uma requisição foi bem sucedida. O resultado dessa verificação está disponível através da propriedade *valid*. Caso não seja fornecido um valor para essa propriedade, subentende-se que uma requisição é válida quando executada com sucesso.
    - ***name:*** nome da propriedade que será utilizado internamente pela biblioteca.
    - ***query:*** *query XPath* que indica o caminho para encontrar o elemento necessário para a validação.
  - ***valid:*** propriedade apenas de leitura que, uma vez que uma *action* seja processada, informa se ela executou com sucesso. O retorno pode ser verdadeiro (*true*) ou falso (*false*).

```
QWebScraper {
    id: scraper
    keepAlive: true
    actions: [

        QWebScraperAction {
            endpoint: "https://www.test.com/login/"
            scraps: [{
                "name": "token",
                "query":"/html/body/main/token/string()"
            }]
        },

        QWebScraperAction {
            endpoint: "https://www.test.com/login/"
            method: "POST"
            headers: {
                "referer": "https://www.test.com/login/"
            }
            data: [
                {"csrfmiddlewaretoken": "%%token%%"},
                {"login": usernameText.text},
                {"password": passwordText.text}
            ]
            validator: {
                "name": "username",
                "query": "/html/body/main/string()"
            }
        }
    ]
}

Button {
    id: loginButton
    text: "Login"
    onClicked: {
        scraper.clearCookies();
        scraper.scrap();
    }
}
```

Listagem 6: Exemplo de uso do QtScraper para autenticação.

A Listagem 6 apresenta um exemplo do arquivo de declaração dos componentes do QtScraper, seguindo as definições apresentadas nesta seção. Consiste em um caso muito comum de autenticação de usuário e validação feita por meio de *token*, este que é obtido na através da página *HTML* e posteriormente utilizado para validação junto as credenciais (usuário e senha).

A propriedade *keepAlive* garante que outros componentes *QWebScraper* que forem instanciados durante a execução da aplicação poderão utilizar a mesma sessão. Caso as credenciais não sejam mais válidas, o desenvolvedor pode utilizar o método *clearCookies* para que o usuário precise se autenticar novamente.

Por último, é realizado o processamento da segunda ação – utilizando método *POST* – utilizando valores que podem ser preenchidos a partir de outros elementos *QML* ou do próprio contexto, como é o caso do exemplo em questão. É possível acessar *scraper.actions[1].valid* para consultar se a autenticação foi bem sucedida.

Todas os eventos do exemplo são disparados através do botão com *id loginButton*. No evento *onClicked*, o *clearCookies* irá limpar uma sessão de login anterior e o método *scrap* irá realizar a execução do *QWebScraper* declarado acima.

## 4.3 Detalhes da implementação

### 4.3.1 Padrão *singleton*

O QtScraper foi desenvolvido utilizando como base o *Qt*, um *framework* que oferece uma série de soluções para o mais variados contextos de acordo com o apresentado na seção 2. Uma dessas soluções é o *QNetworkAccessManager* [24], que mantém todas configurações comuns para as requisições que processa. Ele possui configurações de *proxy* e *cache*, assim como sinais para acompanhamento do progresso de operações de rede. Uma única instância é o suficiente para toda uma aplicação. Para garantir que todas as intâncias de *QWebScraper* utilizam o mesmo objeto de QNetworkAccessManager, foi necessário implementar o padrão de projeto *singleton* [43] conforme representado na Figura 6.

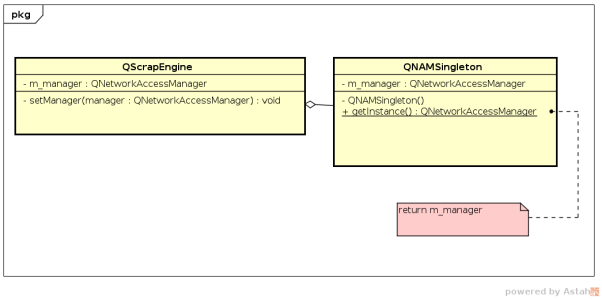


Figura 6: Diagrama de classe de QNAMSingleton.

Utilizar esse padrão proporciona uma série de benefícios para a implementação realizada. O primeira vantagem é em relação a utilização de recursos de memória, já que apenas um objeto precisará ser instânciado para todos os objetos *QWebScraper*, o que para muitos dispositivos móveis ainda é um fator relevante.

Utilizar o padrão *singleton* também facilita a reutilização da sessão por diferentes objetos. Na Figura 7 por exemplo: temos um objeto *loginObject* que irá autenticar um usuário e salvar isso em uma sessão no *manager*, posteriormente o objeto *getCourses* pode usar as mesmas credenciais para obter as disciplinas desse mesmo aluno sem que seja necessário autenticar-se novamente.

Por último percebe-se que o objeto *Context* também é único e é compartilhado entre todas as instâncias de *QScrapEngine*. Levando em consideração que o contexto é global, um ponto importante é tomar cuidado com a repetição de nomes para valores que precisam ser extraídos, para que assim não haja choque de nomes e consequentemente um valor que já estava no contexto seja sobrescrito.

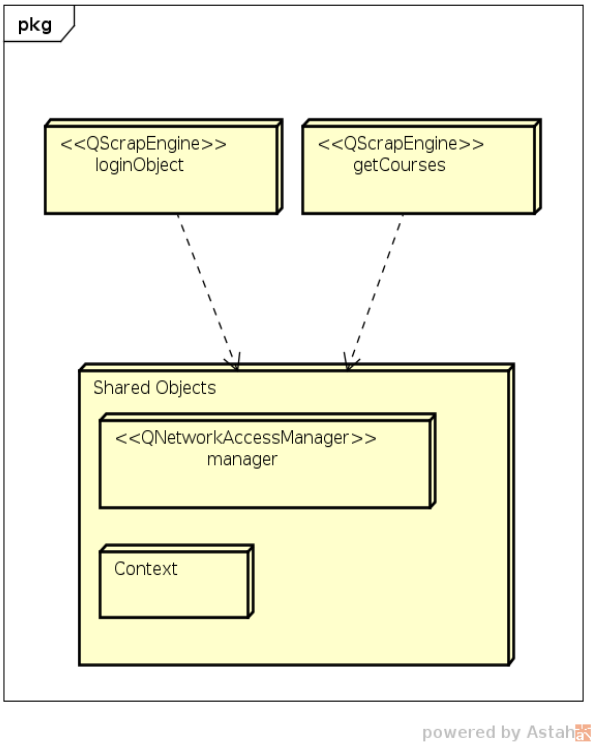


Figura 7: Diagrama dos objetos compartilhados.

### 4.3.2 Implementação do *Scheduler*

Um dos desafios iniciais de implementação da biblioteca foi decidir como implementar o processo de execução das requisições, uma vez que elas são assíncronas. A partir dessa premissa foi implementado o *Scheduler*, com a ideia de inicialmente apenas garantir que as requisições sejam executadas na ordem. A principal motivação para isso foi que dentre os diversos casos de uso levantados durante as fases iniciais do projeto – usando o *POST* principalmente –, a grande maioria precisava de requisições encadeadas e que compartilhassem dados entre si.

A Figura 8 apresenta um diagrama da solução de execução do *requests* pelo *Scheduler*. O fluxo inicia-se a partir da invocação do método *scraper*, que verifica as requisições pendentes, caso existam requisições pendentes de acordo com o indexador atual, o objeto de *request HTTP* é encontrado por meio deste indexador e fica responsável pela execução do *request*.

Assim que a requisição se encerra um sinal é emitido chamado *requestFinished*. O *Scheduler* captura esse sinal e executa o processo de tratamento da resposta. Caso o *status code* da requisição seja inválido, será enviado um sinal *ERROR* para objetos que estejam conectados. Caso contrário, será feito o *parsing* do *HTML* em *XML* e o processo volta à verificação inicial de *requests* pendentes. Se não houver mais pendências, o objeto irá emitir um sinal *Ready* para avisar a objetos interessados, de que a execução foi bem sucedida. Esse fluxo foi implementado por meio de recursão, que tem

como condição de parada, a quantidade de requisições pendentes igual a zero.

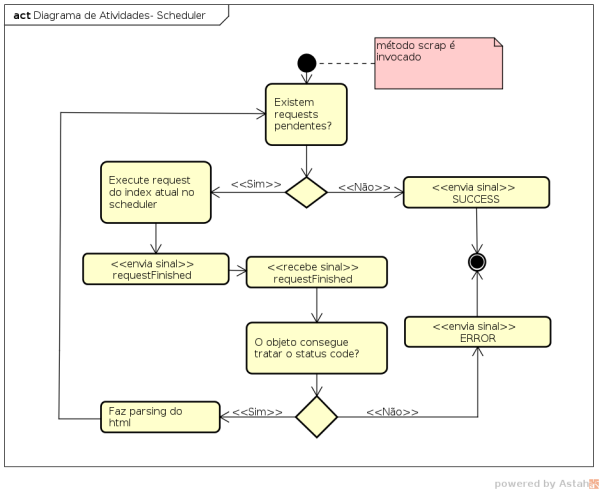


Figura 8: Diagrama de atividades do *Scheduler*.

# 5. AVALIAÇÃO E RESULTADOS

Nesta seção serão apresentadas informações relativas aos experimentos realizados, com o objetivo de testar e avaliar a solução proposta neste trabalho. A Subseção 5.1 apresenta o objeto de estudo em um cenário real em que a solução poderia ser utilizada. A Subseção 5.2 apresenta o processo de execução dos experimentos de forma detalhada, abordando os aspectos de cada cenário analisado. Por fim, a Subseção 5.3 demonstra os resultados obtidos e as avaliações das hipóteses que foram investigadas.

## 5.1 Objeto de estudo: Emile Client

O *Emile Client* [44] é um aplicativo *mobile* e *open-source* para acompanhamento da vida acadêmica, desenvolvido no âmbito do GSORT (Grupo de Sistemas Distribuídos, Otimização, Redes e Tempo Real), grupo de pesquisa do IFBA (Instituto Federal da Bahia). Este aplicativo foi projetado para permitir com que os alunos e professores tenham acesso as suas informações disponíveis no sistema *Web* SUAP (Sistema Unificado de Administração Pública)[1], tendo em vista a inexistência de uma solução para dispositivos móveis fornecida pela instituição.

O sistema é composto pelo aplicativo móvel propriamente dito e do SUAP, que tem seus dados extraídos para consumo no aplicativo. Dentre as principais funcionalidades para os alunos, estão a consulta das disciplinas em que o estudante está matriculado, consulta de faltas, consulta de boletim e visualização de notícias da instituição. Os professores também têm acesso ao seu cronograma de aula, assim como podem registrar a presença/falta de cada aluno. O projeto teve início em 11 de janeiro de 2019, e o desenvolvimento da sua versão inicial ocorreu em cerca de 6 meses. O aplicativo móvel foi desenvolvido em *Qt*. A aplicação móvel tem suporte para os sistemas operacionais *Android* e *iOS*.

No início do projeto, a equipe de desenvolvimento não obteve acesso a uma *API* para consultar os dados dos alunos. Por isso, os desenvolvedores decidiram fazer *web scraping* do site atual da instituição. Dessa forma, o desenvolvimento avançou mais rápido e todos os serviços disponibilizados para os usuário são obtidos de páginas *HTML* do sistema *Web* SUAP.

O problema dessa abordagem é que precisa-se implementar do zero toda a lógica de *requests*, autenticação, sessões, *parsing* de *HTML* e extração de dados. Isso acaba tornando o desenvolvimento mais complexo, pois será necessário ter um bom conhecimento do domínio de desenvolvimento de um *web scraper*, além da complexidade de escrever muito mais código *C++* que é uma linguagem de mais baixo nível do que *QML*. O projeto foi desenvolvido quase que em sua totalidade por um desenvolvedor com mais de 18 anos de experiência com *Qt* e *C++*, logo muitos das dificuldades foram mais facilmente contornadas devido ao histórico de carreira do desenvolvedor. Entretanto, um desenvolvedor menos experiente e a falta de conhecimento de como aplicações de *web scraping* funcionam pode consequentemente desencadear problemas de performance.

O cenário supracitado ilustra um caso real em que o QtScraper poderia ser aplicado para tornar a integração entre aplicativo móvel e aplicação *Web* por meio de *web scraping* mais simples. Os desenvolvedores precisariam criar componentes *QML*, como descrito na Subseção 4.2, indicando as informações necessárias para o *web scraping* das páginas *HTML*, apenas sendo necessário implementar componentes que irão exibir os dados. Caso as URLs e caminhos para os elementos *HTML* mapeados estejam corretos, a biblioteca irá disponibilizar dados já compatíveis com os elementos da interface.

## 5.2 Execução dos Experimentos

O processo da avaliação consistiu na implementação de duas funcionalidades do aplicativo existente, *o Emile Client*, abordado na Seção 5.1 utilizando o QtScraper. Para isso, um conjunto de funcionalidades, apresentados na Tabela 2 foram escolhidas como base para utilização em dois cenários com níveis de complexidade diferentes. Estas funcionalidades foram incorporadas à solução já existente e todo o código legado foi removido. A versão original do aplicativo também foi escrita e *Qt/QML*. Durante o desenvolvimento da versão utilizando a solução proposta por este trabalho, todas os componentes visuais existentes foram mantidos.

Foram criados dois repositórios no *Github*[45][46] contendo uma versão do código-fonte do *Emile Client* de acordo com as configurações necessárias para avaliar os experimentos. O primeiro com a versão original e o segundo utilizando o QtScraper para implementação das duas funcionalidades escolhidas.

Uma vez que o desenvolvimento das funcionalidades utilizando o QtScraper foi finalizado, o objetivo da avaliação foi extrair métricas que pudessem oferecer uma análise das vantagens e desvantagens da solução proposta quando comparadas com uma solução utilizando apenas os recursos fornecidos pelo *toolkit Qt*. A seguir, serão apresentadas as descrição dos experimentos conduzidos após a implementação das funcionalidades.

1. Experimento I (produtividade): foi feita uma análise de *LOC* (*Lines of Code*) produzidas para ambas as versões do projeto. O principal objetivo dessa análise foi analisar a complexidade envolvida em: *i)* reimplementar as funcionalidades existentes para utilizar a solução proposta neste estudo *ii)* integrar a aplicação com o

[1]https://suap.ifba.edu.br/accounts/login/?next=/

| Funcionalidade | Complexidade | Descrição |
|---|---|---|
| F1 | Baixa | Retorna lista de notícias do IFBA |
| F2 | Baixa | Retorna detalhes de uma única notícia do IFBA |
| F3 | Alta | Retorna boletim completo do aluno |

Tabela 2: Funcionalidade implementadas para avaliação

QtScraper. Compreende-se que pela linguagem *QML* ser declarativa e de alto nível de abstração, ela possui uma curva de aprendizado menos íngreme e portanto o desenvolvedor consegue chegar em um bom nível de proficiência mais rápido, principalmente quando comparada a uma linguagem de nível de abstração médio como *C++*. Portanto, para um projeto menos complexo de se manter e implementar, o ideal seria manter uma relação em que existe mais código escrito em *QML* e menos linhas de código escritas em *C++*.

2. Experimento II (desempenho): uma vez finalizada a implementação das funcionalidades, desejou-se analisar o quanto o uso de uma solução genérica de *web scraping* para *Qt* poderia afetar o desempenho de uma aplicação. Para isso foi utilizada a classe *QElapsedTimer* [47] do *Qt* para calcular o tempo médio de processamento de cada funcionalidade considerando um tempo fixo de latência de rede para cada *request* realizado.

A implementação e a execução do estudo ocorreram nos dias 14 e 15 de março de 2020. A execução dos experimentos teve duração aproximada de 4 horas. Ambas as funcionalidades de cada versão foram submetidas aos experimentos I e II, na respectiva ordem. Ao final das execuções, os dados foram registrados e salvos em planilhas para posterior análise.

## 5.3 Resultados e Discussão

O presente estudo visa analisar e avaliar os resultados obtidos na execução dos experimentos I e II. As análise feitas foram organizadas e formatadas de forma que pudessem ser convertidas em dados apresentáveis. A Tabela 3 apresenta as hipóteses que serão analisadas com base nos resultados obtidos.

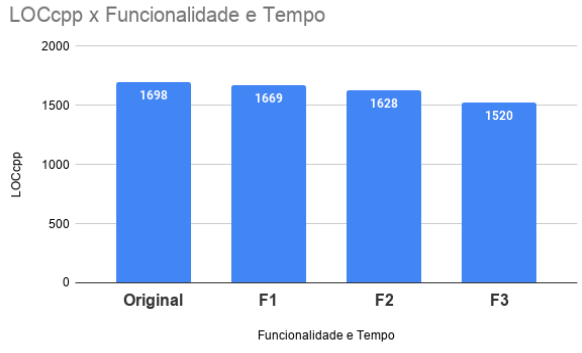


Figura 9: Análise de complexidade das funcionalidade em C++.

As hipóteses H1 e H2, compreendem o uso da métrica *LOC (Lines of Code)* para análise de complexidade dos fatores abordados na Seção 5.2. Para melhor compreensão dos dados analisados, as funcionalidades originais e as desenvolvidas utilizando QtScraper, que estão descritas na Tabela 2, foram separadas em gráficos em formato de barras verticais para cada funcionalidade desenvolvida. Cada funcionalidade foi desenvolvida em ordem cronológica, da esquerda para direita. O estado do software foi mantido entre as implementações de cada funcionalidade.

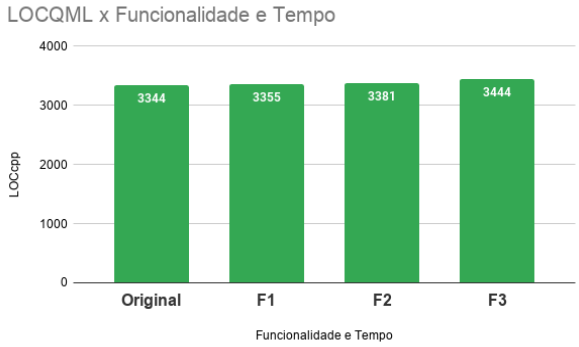


Figura 10: Análise de complexidade das funcionalidade em QML.

As Figuras 9 e 10 apresentam gráficos com as informações da quantidade de linhas C++ e QML para o projeto total antes da utilização do QtScraper e após a implementação de cada funcionalidade, demonstrando o quantitativo de linhas dos arquivos. A separação dos tipos de arquivo foi motivada pela diferença de nível de abstração e abordagem de cada linguagem. Onde *C++* adota um estilo imperativo e *QML* declarativo. Por conta destas diferenças, uma linha de código de cada tipo, deve ter pesos diferentes.

Ainda nas Figuras 9 e 10, é possível visualizar um total de 4 barras para cada gráfico, que estão separadas por funcionalidade, cada uma representando o estado do aplicativo desde a versão original até o desenvolvimento da última funcionalidade F3. O eixo y corresponde as linhas de código em *LOCcpp* (linhas de código *C++*) e *LOCqml* (linhas de código *QML*) da aplicação em determinado momento. O eixo x representa o estado do software após a implementação de cada funcionalidade. Todas as funcionalidades implementadas, estão listadas na Tabela 2.

Nota-se que desde a versão original desenvolvida sem a utilização da solução apresentada neste artigo até o final do desenvolvimento da funcionalidade F3, que houve uma redução considerável de *LOCcpp* e de forma quase proporcional em quantidade, ocorreu um aumento do *LOCqml*, o que é interessante visto que o objetivo da solução é reduzir a complexidade, através da substituição da implementação em *C++* por componentes *QML*. Após o desenvolvimento de F3, houve uma redução de cerda de 10% em *LOCcpp*, quando comparado com a versão original.

| Hipóteses | Métrica | Resultados Esperados |
|---|---|---|
| H1 | Complexidade: LOC | LOCcppQtScraper < LOCcppLOCQt |
| H2 | Complexidade: LOC | LOCqmlQtScraper > LOCqmlQt |
| H3 | Tempo resposta: Tempo | TQtScraper <= TQt |

Tabela 3: Hipóteses avaliadas no estudo.

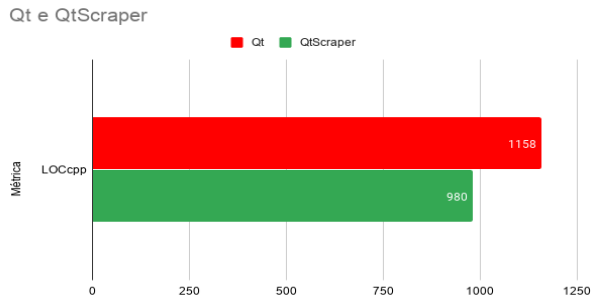


Figura 11: Análise de complexidade das duas versões do aplicativo.

A Figura 11 apresenta os resultado de *LOCcpp*, desconsiderando as linhas de código que não seriam necessárias, caso a aplicação fosse desenvolvida apenas utilizando o QtScraper. Considerando essa abordagem dos dados, nota-se uma redução de cerca de 15% no total, o que é um indício forte de que a partir do momento em todas as funcionalidades fossem adaptadas para o QtScraper, o valor de *LOCcpp* tenderia a um valor próximo de zero.

De acordo com os gráficos apresentado nas Figuras **??** e 11, e com base no que foi analisado e exposto anteriormente, conclui-se que H1 e H2 são verdadeiras. As condições $LOCcppQtScraper < LOCcppLOCQt$ e $LOCqmlQtScraper > LOCqmlQt$ são válidas, visto que que utilizar o QtScraper proporciona a redução de código *C++* e aumento de componentes *QML*.

A hipótese H3, compreende o uso da métrica $T$ (tempo de resposta) para análise de desempenho dos fatores abordados na Seção 5.2 no *Experimento II*. Para melhor compreensão dos dados analisados, as funcionalidades originais e as desenvolvidos utilizando QtScraper, descritos na Tabela 2, serão exibidos em um gráfico de linhas em que é possível compreender a relação de tempo de resposta para cada versão, de cada funcionalidade.

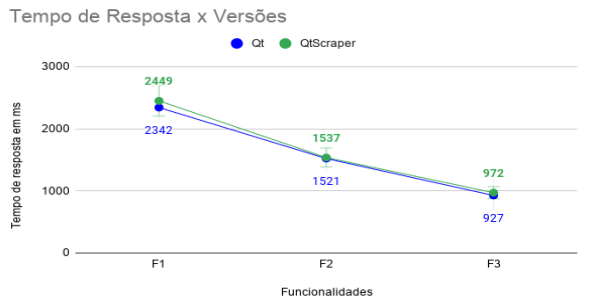


Figura 12: Análise do tempo de resposta por funcionalidade para cada versão.

A Figura 12 apresenta um gráfico com as análises realizadas do tempo de resposta em milissegundos para cada funcionalidade antes (*Qt*) e depois da utilização do QtScraper. Nele, observa-se que o desempenho da solução desenvolvida para a funcionalidade F1 (QtScraper: 2449; *Qt*: 2342) é em média 4,36% mais lenta que a solução original. A funcionalidade F2 (QtScraper: 1537; *Qt*: 1521) apresentou resultados ainda mais próximos entre as duas versões, onde QtScraper foi 1,04% mais lento. Por fim, a funcionalidade F3 (QtScraper: 972; *Qt*: 927) apresentou a maior margem de diferença entre as duas versões, sendo a versão *Qt* 4,62% mais rápido.

No que tange o tempo de resposta referente a cada versão, nota-se que a solução *Qt* foi em média 3,34% mais rápida que o QtScraper. Portanto, a partir da análise dos gráfico disponível nas Figura 12, conclui-se que H3 é falsa. Dessa forma, utilizar o QtScraper para *web scraping* em *Qt* tem maior tempo de resposta — e portanto, gera aplicações um pouco mais lentas — do que desenvolver uma solução apenas com o que o *toolkit Qt* fornece. Contudo, é importante salientar que uma das premissas é que desenvolvedores com pouca experiência com *Qt*, *C++* e *web scraping* podem ter piores performances, o que não é o caso, visto que o desenvolvedor do *Emile Client* possui vasta experiência com as tecnologias. Outro fator relevante que não pode ser desconsiderado é que soluções mais genéricas precisam lidar com tipos mais variados de casos do que uma aplicação e específico precisa, portanto, isso também pode afetar o resultado final.

## 6. CONCLUSÃO

Existem diversas situações em que o *web scraping* é a solução mais viável para implementação de integração de uma nova aplicação com uma aplicação *Web* legada. Entretanto, realizar essa integração perpassa por vários desafios, principalmente considerando-se o acesso aos recursos, como os dados serão selecionados, projeto e análise e por fim a fase de produção.

Embora existam algumas soluções de software voltadas para o *web scraping*, hoje não existe uma solução para *Qt e C++* que consiga lidar com os problemas relacionados ao *web scraping* com um nível mais alto de abstração e com boa performance. Nesse contexto, o presente trabalho implementou e avaliou de uma solução para facilitar a desenvolvimento baseada em *web scraping* para dispositivos móveis. O QtScraper estabelece uma solução de mais alto nível de abstração, através da utilização de componentes que tornam o desenvolvimento deste tipo de solução mais transparente para os desenvolvedores.

A avaliação da solução proposta foi realizada a partir da análise de funcionalidades de um mesmo projeto, primeiramente utilizando apenas *Qt* e *C++*, em seguida fazendo uso do QtScraper.Fora executados dois experimentos distintos: no primeiro, foi analisado a complexidade da solução através da análise das linhas de código produzidas para cada funcionalidade, através da análise de diferentes perspecti-

vas destes dados; No segundo, levou-se em consideração o tempo de resposta para cada funcionalidade para análise de desempenho da solução.

O desenvolvimento do presente trabalho possibilitou uma análise comparativa entre soluções de *web scraping* para dispositivos móveis baseadas em *Qt*. O QtScraper demonstrou ser uma solução que oferece menos complexidade para o desenvolvedor por meio do uso de componentes, resolvendo problemas de forma transparente e com performance muito próxima da solução tradicional baseada apenas em *Qt* e *C++*.

## 6.1 Limitações Deste Trabalho

A primeira limitação deste trabalho refere-se as comparações feitas com partes menores do sistema com graus de complexidade que são subjetivos, portanto podem tornar a avaliação menos confiável. O ideal seria que o aplicativo fosse reescrito por completo, assim a avaliação teria um grau muito menor de subjetividade. Devido ao autor deste do presente estudo não ter participado originalmente do desenvolvimento do aplicativo, que no momento já possui muitas funcionalidades, o entendimento das regras de cada uma delas demandaria um tempo total que poderia inviabilizar a conclusão do trabalho.

No que tange a segunda limitação, observa-se que a pesquisa não apresenta experimentos com participantes inexperientes no uso do QtScraper, comparando os resultados de cada um com a utilização de solução tradicional baseada apenas em *Qt*. Isso poderia avaliar se o uso da solução desenvolvida poderia avaliar agilidade do desenvolvimento, complexidade e performance por meio de outro perspectiva mais condizente com as premissas do trabalho.

## 6.2 Trabalhos futuros

Durante o desenvolvimento da solução proposta, bem como durante a avaliação dos resultados, observou-se possibilidades de nova pesquisas e consequentemente evoluções para a biblioteca. Nesse contexto, destacam-se como trabalhos futuros:

- Realização de novos experimentos com desenvolvedores inexperientes e experientes para comparar como o QtScraper se posiciona nas analises realizadas durantes esse trabalho. Seria possui incluir novas variáveis, como: densidade de *bugs* e tempo de desenvolvimento. Isso tornaria a avaliação mais rica.

- Hoje o *QtSraper* não tem suporte para elementos gerados por JavaScript, isso limita a quantidade de páginas *Web* compatíveis com a biblioteca. Um trabalho relevante, seria o desenvolvimento e implementação de arquitetura para processamento de JavaScript compatível com a solução atual.

# 7. REFERÊNCIAS


[1] N. Brügger and R. Schroeder, *The web as history: Using web archives to understand the past and the present*. UCL Press, 2017.

[2] "Mobile Vs. Desktop Usage (Latest 2020 Data)." https://www.broadbandsearch.net/blog/mobile-desktop-internet-usage-statistics, 2020. [Online; acessado 13-Março-2020].

[3] V. C. Feijó, B. S. Gonçalves, and L. S. R. Gomez, "Heurística para avaliação de usabilidade em interfaces de aplicativos smartphones: Utilidade, produtividade e imersão," *Design e Tecnologia*, vol. 3, no. 06, pp. 33–42, 2013.

[4] "Qt." https://www.qt.io/what-is-qt/, 2019. [Online; acessado 10-Julho-2019].

[5] "Comparador de preços Zoom)." https://www.zoom.com.br/, 2020. [Online; acessado 13-Março-2020].

[6] "Comparador de preços Buscapé)." https://www.buscape.com.br/, 2020. [Online; acessado 13-Março-2020].

[7] O. ten Bosch, D. Windmeijer, A. van Delden, and G. van den Heuvel, "Web scraping meets survey design: Combining forces," in *Big Data Meets Survey Science Conference, Barcelona, Spain*, 2018.

[8] D. Glez-Peña, A. Lourenço, H. López-Fernández, M. Reboiro-Jato, and F. Fdez-Riverola, "Web scraping technologies in an api world," *Briefings in bioinformatics*, vol. 15, no. 5, pp. 788–797, 2013.

[9] Q. Zhang and R. S. Segall, "Web mining: a survey of current research, techniques, and software," *International Journal of Information Technology & Decision Making*, vol. 7, no. 04, pp. 683–720, 2008.

[10] H. Phan, "Building application powered by web scraping," *Helsinki Metropolia University of Applied Sciences*, 2019.

[11] E. Lewerenz, "An example of website "screen scraping".," *MWSUG*, 2009.

[12] "Web Scraping with Proxies: The Complete Guide to Scaling Your Web Scraper." https://blog.hartleybrody.com/web-scraping-proxies/, 2018. [Online; acessado 02-Agosto-2019].

[13] "Regular Expressions." https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions, 2019. [Online; acessado 19-Junho-2019].

[14] "C++." https://isocpp.org/, 2019. [Online; acessado 10-Julho-2019].

[15] "The Qt Company." https://www.qt.io/company, 2019. [Online; acessado 10-Julho-2019].

[16] "KDE." https://kde.org/, 2019. [Online; acessado 10-Julho-2019].

[17] "Nokia." https://www.nokia.com/pt_int/, 2019. [Online; acessado 10-Julho-2019].

[18] "Digia." https://digia.com/en/, 2019. [Online; acessado 10-Julho-2019].

[19] "Qt History." https://wiki.qt.io/Qt_History, 2018. [Online; acessado 10-Julho-2019].

[20] "Qt - Success Stories." https://resources.qt.io/customer-stories-all, 2019. [Online; acessado 10-Julho-2019].

[21] "Qt Supported Platforms." https://doc.qt.io/qt-5/supported-platforms.html, 2019. [Online; acessado 10-Julho-2019].

[22] S. Larndorfer, "Qt qml v html5 – a practical comparison," *Sequality Software Engineering e.U.*, 2017.

[23] "Signals and Slots." https://doc.qt.io/qt-5/signalsandslots.html, 2020.

[Online; acessado 10-Março-2019].

[24] "QNetworkAccessManager Class." https://doc.qt.io/qt-5/qnetworkaccessmanager.html, 2019. [Online; acessado 18-Julho-2019].

[25] "XMLHttpRequest - Web APIs." https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest, 2019. [Online; acessado 18-Julho-2019].

[26] "XmlListModel QML Type." https://doc.qt.io/qt-5/qml-qtquick-xmllistmodel-xmllistmodel.html, 2019. [Online; acessado 18-Julho-2019].

[27] "Best scraper tools. Comparison of the most popular scraper." https://parsers.me/best-scraper-tools-comparison-of-the-most-popular-scraper/, 2018. [Online; acessado 19-Junho-2019].

[28] "The 10 Best Data Scraping Tools and Web Scraping Tools." https://www.scraperapi.com/blog/the-10-best-web-scraping-tools, 2018. [Online; acessado 19-Junho-2019].

[29] "Scrapy | A Fast and Powerful Scraping and Web Crawling Framework." https://scrapy.org/, 2019. [Online; acessado 19-Junho-2019].

[30] "A Powerful Spider(Web Crawler) System in Python.." https://github.com/binux/pyspider, 2019. [Online; acessado 19-Junho-2019].

[31] "Fast, flexible, and lean implementation of core jQuery designed specifically for the server.." https://cheerio.js.org/, 2019. [Online; acessado 19-Junho-2019].

[32] "jQuery." https://jquery.com/, 2019. [Online; acessado 19-Junho-2019].

[33] "AJAX." https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX, 2019. [Online; acessado 10-Julho-2019].

[34] "XML Path Language (XPath)." https://www.w3.org/TR/1999/REC-xpath-19991116/, 2016. [Online; acessado 19-Junho-2019].

[35] "Beautiful Soup 4.4.0." https://www.crummy.com/software/BeautifulSoup/bs4/doc/, 2019. [Online; acessado 19-Junho-2019].

[36] "Scraper API - Easily Build Scalable Web Scrapers." https://www.scraperapi.com/, 2019. [Online; acessado 19-Junho-2019].

[37] "Getting Started with Headless Chrome." https://developers.google.com/web/updates/2017/04/headless-chrome, 2019. [Online; acessado 18-Maio-2020].

[38] "ParseHub." https://www.parsehub.com/, 2019. [Online; acessado 19-Junho-2019].

[39] "The web scraping and automation platform." https://apify.com/, 2019. [Online; acessado 19-Junho-2019].

[40] R. N. Taylor, N. Medvidovic, and E. Dashofy, *Software architecture: foundations, theory, and practice*. John Wiley & Sons, 2009.

[41] "Scheduler Agent Supervisor pattern." https://docs.microsoft.com/en-us/azure/architecture/patterns/scheduler-agent-supervisor, 2018. [Online; acessado 03-Março-2019].

[42] "Introduction to LibTidy." http://tidy.sourceforge.net/libintro.html, 2015. [Online; acessado 03-Junho-2020].

[43] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[44] "Emile: PlayStore." https://play.google.com/store/apps/details?id=br.edu.ifba.gsort.emile, 2020. [Online; acessado 01-Março-2020].

[45] "QtScraper: Experimento 1." https://github.com/EliakinCosta/qtscraper-experimento1, 2020. [Online; acessado 01-Março-2020].

[46] "QtScraper: Experimento 2." https://github.com/EliakinCosta/qtscraper-experimento2, 2020. [Online; acessado 01-Março-2020].

[47] "Qt Docs: QElapsedTimer Class." https://doc.qt.io/qt-5/qelapsedtimer.html, 2020. [Online; acessado 01-Março-2020].