C++ Web Framework: A Web Framework for Web Development using C++ and Qt

Herik Lima[1,2] and Marcelo Medeiros Eler[1]
[1]*University of São Paulo (EACH-USP), São Paulo, SP, Brazil*
[2]*XP Inc., São Paulo, SP, Brazil*

Keywords: Web, Framework, Development, C++.

Abstract: The entry barrier for web programming may be intimidating even for skilled developers since it usually involves dealing with heavy frameworks, libraries and lots of configuration files. Moreover, most web frameworks are based on interpreted languages and complex component interactions, which can hurt performance. Therefore, the purpose of this paper is to introduce a lightweight web framework called C++ Web Framework (*CWF*). It is easy to configure and combine the high performance of the C++ language, the flexibility of the Qt framework, and a tag library called CSTL (C++ Server Pages Standard Tag Library), which is used to handle dynamic web pages while keeping the presentation and the business layer separated. Preliminary evaluation gives evidence that *CWF* is easy to use and present good performance. In addition, this framework was used to develop real world applications that support some business operations of two private organizations.

1 INTRODUCTION

Web Applications have been adopted as the *de facto* platform to support business operations of all sort of organizations for a long time. This has been even more stimulated by the availability of modern frameworks and tools (Chaubey and Suresh, 2001) along side with the growth of technologies related to Software as a Service (Tsai et al., 2014), Cloud Computing (Buyya et al., 2009) and Mobile Platform Applications (Charland and Leroux., 2011; Freitas and Maia, 2016). The benefits of Web Applications are indeed very attractive for companies of all sizes and sectors, but the increasing complexity of web applications and the underlying development infrastructure comes with a price.

According to Sinha et. al (Sinha et al., 2015), programming for the web may be quite intimidating because writing even conceptually simple applications requires inordinate amount of effort: learning multiple languages; diving into details of low level frameworks/libraries (e.g. JSF and Strutts for Java-based Web frameworks, Ruby on Rails are Ruby-based, Grails is Groovy-based, .NET Framework, CakePHP is for PHP-based frameworks, and Lift is for Scala (del Pilar Salas-Zarate et al., 2015)); keeping libraries and frameworks updated and compatible (Raemaekers et al., 2014), dealing with several configuration files; and writing glue code to make multiple layers inter-operate (Vuorimaa et al., 2016). In addition, many web frameworks present poor documentation (Constanzo and Casas, 2016; Constanzo and Casas, 2019).

Those peculiar characteristics of web development environments have many practical consequences. First, the entry barrier for web programming is high even for skilled programmers due to the complex configuration (Sinha et al., 2015; Swain et al., 2016; Tuan et al., 2016), which can make the learning curve too steep. Next, web applications may present low maintainability once finding and fixing bugs, creating new features, integrating solutions and keeping configuration files, frameworks and libraries updated and compatible may require a lot of effort. Some solutions try to use single pages that contain business and presentation layer logic to reduce complexity, but it also render application less maintainable (Shklar and Rosen, 2004; Srai et al., 2017). Finally, several web frameworks rely on a heavy combination of other frameworks, components and libraries and on interpreted languages and scripts, which may jeopardize the overall performance of the application and increase computational resource consumption. Although performance is not the main concern of web development, there might be applications or specific functions that requires good performance (e.g. trad-

Lima, H. and Eler, M.
C++ Web Framework: A Web Framework for Web Development using C++ and Qt.
DOI: 10.5220/0010457700760087
In *Proceedings of the 23rd International Conference on Enterprise Information Systems (ICEIS 2021) - Volume 2*, pages 76-87
ISBN: 978-989-758-509-8

ing systems).

Despite the fact that C++ is a well known language that presents high performance and low resource consumption (Game, 2021), and it is one of the four most used programming language in the world (TIOBE, 2021), it has not been used to develop web applications (Millares, 2015). So far, few approaches have been proposed to support web development using the C++ language. In particular, CGI (Common Gateway Interface) solutions offer a standard protocol for web servers to execute programs like console applications that runs on the server and generate web pages dynamically (CGI). There are, however, other solutions available, such as POCO Libraries (Obiltschnig, 2005), C++ Server Page (CSP), Wt (Dumon and Deforche, 2008), Crow [1] e QWebApp (Frings, 2010). All of them, however, make business, control and presentation layer entangled for the sake of simplification, however it renders applications less adaptable, maintainable and reusable (Srai et al., 2017).

In this context, this paper introduces a new web development framework called C++ Web Framework (*CWF*) which was built to mitigate many of the issues related to web development (e.g complex configuration, steep learning curve, low maintainability, limited performance, and high computational resource consumption(del Pilar Salas-Zarate et al., 2015; Sinha et al., 2015; Swain et al., 2016; Raemaekers et al., 2014; Shklar and Rosen, 2004; Srai et al., 2017)) by providing developers with a lightweight infrastructure to develop web applications using the C++ programming language. *CWF* was built upon the Qt[2] framework, a cross-platform framework for the development of desktop, embedded and mobile applications in C++.

CWF is lightweight because it uses only one configuration file and relies on a few libraries provided by the Qt framework. Web applications implemented in a compiled language such as C++ and upon the optimized infrastructure provided by Qt tend to use less computational resources such as memory and processor, and to present high performance. A tag library called CSTL (C++ Server Pages Standard Tag Library) was created to generated dynamic web pages that keeps the presentation and the business layer separated. Architectural and design decisions concerning our solution is presented in Section 3.

Preliminary evaluation gives evidence that this framework is easy to use and understand, which makes it a good choice as an entry level web framework. It was also found that the *CWF* applications present good performance. In particular, two real world applications were developed using this framework and they have been running for several months. This first set of evaluations motivates further improvement and assessment of the *CWF*.

This paper is organized as follows. Section 2 discuss the basic requirements we collected to develop our web framework and the correspondent architectural and design decisions we made to provide developers with a suitable solution for the development of we applications. Section 3 presents the *CWF* in details and provides examples of how to instantiate a new web application. Section 4 shows preliminary evaluations we have conducted to asses the *CWF* – user evaluation, performance evaluation and usage in real world scenarios. Section 5 discuss some related work. Finally, Section 6 presents the concluding remarks and future directions.

2 REQUIREMENTS, ARCHITECTURAL AND DESIGN DECISIONS

The overall concept and structure of a web application is quite simple. A web application is a software that runs on a web server to provide their clients with different types of services, such as web-mail or online shopping. The web application handles requests received from its client and sends back an answer. Such communication is based on a agreed protocol, such as HTTP (Hyper Text Transfer Protocol).

As web applications have became more complex, handling clients requests also become more complex and, in many cases, requires the interaction with many components, services and other applications. Therefore, it is expected that a web framework provides web developers with several other facilities to make it easy to build complex and robust web applications. Considering such scenario, we decided that the *CWF* would implement the following resources in its first versions: support to MVC (Model View Controller) architecture, session management, filters, web services, XML and JSON manipulation, secure HTTPS, data base access and ORM (Object-Relational-Mappers).

Many architectural and design decisions were made during the *CWF* development to mitigate some of the issues related to web development aforementioned. The general idea was to implement a flexible but yet a lightweight solution for developing web applications. We briefly discuss some of our decisions as follows.

[1] https://github.com/ipkn/crow

[2] https://www.qt.io/

Programming Language. We decided to implement the *CWF* using the C++ language and provide support to the development of web applications using C++ for many reasons: high performance and low usage of computational resources (Ramana and Prabhakar, 2005; Hundt, 2011; Game, 2021); it is a widely used programming language – it is one of the top four most used language in the world (TIOBE, 2021); it is constantly updated and no conflict is introduced with previous versions; and many issues related to older versions of the language, such as memory leaks, have been solved in recent versions.

Underlying Framework. Many resources and facilities required by web applications are not natively supported by programming languages. Therefore, web applications require external libraries to implement many of its functionalities. Such libraries are usually provided or orchestrated by web frameworks. Therefore, we decided to built the *CWF* upon the Qt framework for many reasons: it provides several libraries that can be used to develop a web application (e.g. socket, XML and JSON manipulation, reflection, data base access, and so forth); it is constantly updated and new versions are compatible with older versions; it is cross-platform as it can run on Windows, Linux, Mac, Android, iOS, tvOS, watchOS, WinRT and embedded environments; it is widely used by C++ developers and big companies; and it has a detailed and good quality documentation.

Configuration Files. When many configuration files must be set it makes the development and evolution process more complex (Sinha et al., 2015; del Pilar Salas-Zarate et al., 2015; Vuorimaa et al., 2016). Therefore, *CWF* requires the developer to set a single configuration file in which 19 options are available. Details on the configuration files are presented in Section 3.

Separation between Business and Presentation Layer. Entangling business and presentation layer, although it has been a simple solution adopted by many approaches to present dynamic information, makes applications less understandable, testable and maintainable. Therefore, we created the CSTL (C++ Server Pages Standard Tags Library), a tag library that generates dynamic content but keeping the presentation and the business layer separated. It was largely inspired by the JSTL (JavaServer Pages Standard Tags Library) from the Java EE platform.

Application Container. Typically, web applications run on web containers (e.g. TOMCAT, IBM WebSphere, JBoss) that run on web servers. However, *CWF* applications cannot be deployed to web containers currently available in the market, unless it is implemented under protocols such as CGI or FastCGI, but it would hurt performance. Therefore, we decided to create a specific web container for the *CWF* applications and make each application its own web container. That way, *CWF* applications can run on any platform in which Qt framework can run.

Java Servlets Inspiration. CGI-based solutions tend to present low performance and high use of computational resources. FastCGI is a solution that try to reduce the overhead related to interfacing between web server and CGI programs, but it does not completely solve the overhead issues. Java servlets, on the other hand, can handle requests in an optimized way by using a pool of threads to deliver better performance and save computational resources. Thus, *CWF* controllers handle requests based similarly to Java Servlets solutions.

3 THE C++ WEB FRAMEWORK (CWF)

The C++ Web Framework (*CWF*)[3] was devised to support the development of web applications using the C++ language and the Qt framework. Qt is a cross-platform application development framework for desktop, embedded and mobile. Qt is not a programming language by its own. It uses a preprocessor called MOC (Meta-Object Compiler) that extends the C++ language and parses the source files written in Qt-extended C++ to generate sources that can be compiled by any standard compliant C++ compiler.

The combination of the flexibility of the Qt framework and the power of the C++ language is one of the appealing characteristics of the *CWF*. Developers can use the Qt facilities to access databases, create web services, parse XML and JSON, use testing frameworks, and so forth. Developers can also use external libraries. Accordingly, the web applications developed in this environment are intended to present the following characteristics:

- High performance since it is based on the C++ language and it doesn't use interpreted languages
- Low use of computational resources (memory and processor)
- It requires only a simple configuration file

[3]https://github.com/HerikLyma/CPPWebFramework

- Dynamic content can be presented using dynamic pages created with CSTL (see Section 3.5)
- Separation between the presentation (HTML, for example) and the business layer (back-end code)

The *CWF* may be especially appealing for C++ developers that want to develop web applications but cannot afford learning a new programming language or dealing with heavy frameworks. In fact, a recent study[4] has shown that the population of C++ developers is past 5 million people, which is one of our motivation to develop a framework suitable for this specific population.

3.1 The CWF Architecture

Figure 1 shows an overview of the architecture of the *CWF*. A *CWF* application (*CppWebApplication*) is built upon the Qt framework and libraries, and the C++ language. The application is wrapped by its own web server or container (*CppWebServer*), so the developer does not need to deploy the web application in a web container such as Apache Tomcat, for example. This is very useful to keep the infrastructure simple and to avoid security and conflicts issues. The servlets inside the *Controllers* components answer the client requests and present dynamic content through web pages and *CSTL* (C++ Server Pages Standard Tag Library). The servlets can also use standard C++ classes. The details of each component of the `CWF` and simple usage examples are presented as follows.

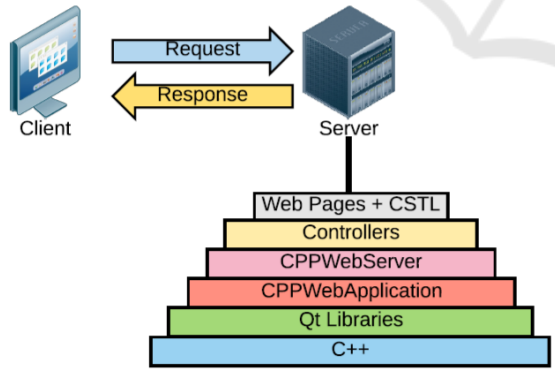

Figure 1: Overview of the *CWF* architecture.

Developers must use the Qt environment to create *CWF* applications. The new project must use the Qt modules `core`, `network` and `xml` and import folders `cwf` and `server` from the *CWF*. If the application uses a database connection, for example, it must also import the *sql* module.

[4]https://adtmag.com/articles/2018/09/24/developer-economics-survey.aspx

Figure 2 shows an example of how the folder hierarchy of a *CWF* application will look like for a simple Hello World application: `HelloWorld` is the main folder of the project; `HelloWorld.pro` is the Qt project file; `Sources` stores all source code of the project; `Other files` stores all supporting files; `server/config` keeps the configuration file of the project; `cppwebserverpages` keeps the default pages of the project (index.html, 403 and 404, for example); `log` keeps the server's log information; and `ssl` stores the files to configure the HTTPS protocol. Listing 1 shows an example of the file `CppWeb.ini` to set the configuration of the *CWF* web server, which is built-in each application.

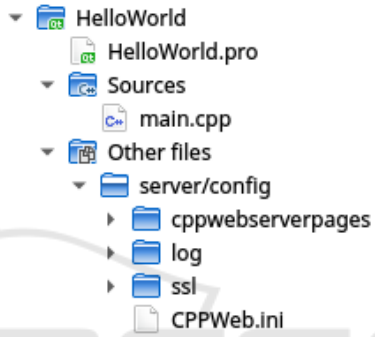

Figure 2: The folder structure of a *CWF* application.

Listing 1: CppWeb.ini file.

```
[config]
host=127.0.0.1
port=8080
maxThread=200
cleanupInterval=60000
timeOut=60000
maxUploadFile=500000000
path=/CPPWebFrameworkTest/server/
suffix=.xhtml
logFilePath=/config/log/
sslKeyFile=/config/ssl/my.key
sslCertFile=/config/ssl/my.cert
indexPage=/config/cppwebserverpages/index
accessCPPWebIni=false
accessServerPages=false
```

As mentioned before, *CWF* application requires only one configuration file. The details of each element of the `CppWeb.ini` file is presented as follows: `host` and `port` set the IP and the port number the web server will use, respectively; `maxThread` sets the maximum amount of thread the server is allowed to create; `cleanupInterval` sets the periodicity to clean sessions; `timeOut` sets the maximum amount of time the server can use to answer a request; `sessionExpirationTime` sets the time a session takes to expire; `maxUploadFile` sets the max-

imum size of a request; `path` points to the server folder; `suffix` sets the suffix to be added to the page requests; `logFilePath` points to a log file; `sslKeyFile` points to a `.key` file and `sslCertFile` points to a `.cert` file, which are related to digital certifications; `indexPage` sets the main and first page of the project to be shown; `accessCPPWebIni` sets whether the `CPPWeb.ini` file can be remotely accessed; `accessServerPages` sets whether the default pages of the *CWF* can be accessed.

3.2 CppWebApplication and CppWebServer

The class `CppWebApplication` is the main class of the web application. It encapsulates the `CppWebServer` and other core classes of the Qt framework. The `CppWebApplication` creates a server using the parameters (e.g. host, port) set by the developer and the application can be accessed using the address host:port. Listing 2 shows a view that is called by the web application presented in Listing 3. The `CppWebApplication` receives `argc` and `argv` as parameters. A server is created using `argc`,`argv` and the path to the `server` working directory. The `HelloWorldController` is added to the server, which is started in the sequence. After that, this application can be accessed using the address `host:port`. Listing 3 also shows the class `HelloWorldController`, which inherits from `CWF::Controller` and overrides the method `doGet` o answer requests by sending the view *helloworld.view* as answer.

Listing 2: A simple view using HTML.

```
<html>
    <body>
        Hello World!
    </body>
</html>
```

Listing 3: A simple view using HTML.

```
#include <cwf/cppwebapplication.h>
class HelloWorldController : public CWF::Controller
{
public:
  void doGet(CWF::Request &request,
       CWF::Response &response) const override
  {
    request.getRequestDispatcher("/pages/helloworld.view")
           .forward(request, response);
  }
};

int main(int argc, char *argv[])
{
    CWF::CppWebApplication server(argc, argv, "/server");
    server.addController<HelloWorldController>("/hello");
    return server.start();
}
```

3.3 Controllers

The *CWF* controllers (or servlets) were strongly inspired by the Java servlet, which allow handling requests and generating dynamic content. Implementing controllers in the *CWF* is straightforward: the developer has to create an extension of the `CWF::Controller` and override the virtual methods: `doGet`, `doPost`, `doPut`, `doDelete`, `doTrace` and `doOptions`. Each method receives two parameters which are references to `Request`, which keeps information concerning a request, and `Response`, which provide the mechanisms to answer a request.

Listing 4 shows an example of a controller created for handling sessions. The only rule that developers must comply with to register an information within a session is that the class containing the information must be an extension of `QObject`. Method `fillQObject` is used to set the attributes of the object `usr`, whose values are received via input form. Method `getSession` is used to obtain the current session and method `addAttribute` includes an object into the current session.

Listing 4: Example of the usage of a session.

```
class LoginController : public CWF::Controller
{
public:
    void doGet(CWF::Request &req,
               CWF::Response &resp) const override
    {
        req.getRequestDispatcher("/pages/login.view")
           .forward(req, resp);
    }
    void doPost(CWF::Request &req,
                CWF::Response &resp)  const override
    {
        QString login(req.getParameter("login"));
        QString pwd(req.getParameter("passwd"));
        if(login == "user" && pwd == "pwd")
        {
            User *usr = new User;
            req.fillQObject(usr);
            req.getsession().addAttribute("usr", usr);
            req.getRequestDispatcher("/pages/home.view")
               .forward(req, resp);
        }
        else
        {
            req.getDispatcher("/pages/login.view")
               .forward(req, resp);
        }
    }
};
```

All controllers must be registered so the server can know that they exist and how to answer the requests for each controller. This task can be easily done using the methods `addController` from the class `CppWebApplication`. A possible drawback of this approach is that *CWF* applications would be less flexible since controllers could not be changed or added dynamically. However, it is possible to use the dynamic linking facilities provided by the Qt framework, which can also help reducing resources consumption.

3.4 Filters

In web applications, filters are used to intercept requests before they reach a controller. They can pre-process requests and check, for example, whether a user session is still valid. In *CWF*, filters can be added by creating instances of `Filter` and overriding the method `doFilter`. As the controllers, filters also need to be registered within the `CppWebApplication`.

Listing 5 shows an example of the usage of filters in the *CWF*. This filter checks whether the user is authenticated and whether the session is still valid. If the session is invalid, the filter will redirect the user to the authentication page. Otherwise, it will present the requested resource.

Listing 5: Example of the usage of a filter.

```
class SessionValidatorFilter:public CWF::Filter
{
public:
 void doFilter(CWF::Request &req,
      CWF::Response &resp, CWF::FilterChain &chain)
 {
    QString url(req.getRequestURL());
    if(url.endsWith(".css") || url.endsWith(".png"))
    {
       chain.doFilter(req, resp);
    }
    else if (url != "/login")
    {
      QObject *obj = req.getSession().getAttribute("usr");
      if (obj == nullptr || reg.getSession().isExpired())
      {
          req.getRequestDispatcher("/pages/login.view")
             .forward(req, resp);
      }
      else
      {
          chain.doFilter(req, resp);
      }
     }
     else
     {
       chain.doFilter(req, resp);
     }
  }
};
```

3.5 The Web Pages and the CSTL

The CSTL (C++ Server Pages Standard Tag Library) is a tag library created to generated dynamic content but keeping the presentation and the business layer separated. It was inspired by the JSTL (JavaServer Pages Standard Tag Library) library from the Java platform. The CSTL allows using C++ objects within xhtml pages using tags instead of programming instructions. So far, the CSTL designed for the *CWF* has the following tags: out, for, and if. Even though there are only three (3) tags currently, their combination are flexible and powerful enough to allow building several kinds of web applications. The complete specification of the CSTL can be found in the *CWF* repository[5].

There are some rules to use the CSTL: i) all objects must be an extension of `QObject`; ii) all methods used by the CSTL must be in the section `public slots`; iii) it is only possible to retrieve information from the objects, not set values; iv) the return of the methods used by the CSTL must be a C++ primitive type, or `std:string`, or `QString`.

Listing 6 shows an example of a class that can be used within view pages. Notice that the class costumer inherits from QObject and the getter methods are declared within the `public slot` section. The declaration of the setter methods are hidden.

Listing 6: A class prepared to be used by the CSTL.

```
class Customer : public QObject
{
    Q_OBJECT
private:
    int id;
    char gender;
    QString name;
    QString address;
public:
    explicit Customer(QObject *o = nullptr): QObject(o)
    {}
public slots:
    int getId() const  { return id; }
    char getGender() const  { return gender; }
    QString getName() const { return name; }
    ...
};
```

Listing 7 shows a controller that passes a list of objects to the view page. Following, Listing 8 presents the usage of the list of objects received. When using the CSTL, methods are called using `#{class.method}` as a template and the values are shown using the tag `out`. The tag `for` can be used to iterate over a collection of objects over numeric values (e.g, `<for var="i" from="1" to="10" increment="1">`). In this example, the tag `for` is used to access each object within the customer list received from `CustomerServlet`. In this example, the tag `if` imposes a condition: only male customers (gender=='M') will be shown.

Listing 7: Passing objects to web pages.

```
class CustomerServlet : public CWF::HttpServlet
{
public:
  virtual ~CustomerServlet(){}
  void doGet(CWF::HttpServletRequest &req,
             CWF::HttpServletResponse &resp)
  {
     ...
     CWF::QListObject qListObject;
     qListObject.setAutoDelete(true);
     qListObject.add(customerOne);
     qListObject.add(customerTwo);
     qListObject.add(customerThree);
     qListObject.add(customerFour);
     req.addAttribute("customersList", &qListObject);
     req.getRequestDispatcher("/pages/customer").
                                forward(req, resp);
   }
};
```

[5]https://github.com/HerikLyma/CPPWebFramework

Listing 8: Using the CSTL tag for.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<html>
  <body>
    <table width="550px" border="1">
     <tr>
       <td width="100" height="40">
          <b>Name</b>
       </td>
     </tr>
     <for items="customersList" var="customer">
      <if var="#{customer.getGender}" equal="M">
       <tr>
        <td height="25">
         <out value="#{customer.getId}"></out>
        </td>
        <td height="25">
         <out value="#{customer.getName}"></out>
        </td>
       </tr>
      </if>
     </for>
    </table>
  </body>
</html>
```

4 EVALUATION

A preliminary assessment of the *CWF* was carried out to evaluate the framework considering three forms: user evaluation, performance testing and applicability in real world scenarios. The details of each evaluation are presented as follows.

4.1 User Evaluation

The purpose of this evaluation is to check whether developers could install, configure and create dynamic web pages using the *CWF* framework. Therefore, we design an experiment as follows:

- Subjects: C++ developers with or without knowledge in web development.
- Subjects Selection: an invitation was sent to all developers or former students registered in the database of an I.T. company that develops software and provides training for C++ developers.
- Location: the experiments will take place on the laboratories of the I.T. company that develops software and provides training for C++ developers.
- Execution: experiments lasted four hours at most. Subjects received instructions[6] on how to use the *CWF* framework to develop dynamic web applications during two hours. The following two hours was used to execute four activities.
- Data Collection: we will collect the following information on each subject: education, experience with software development in general and web development, opinion on how easy was to install and configure the *CWF* framework, and the time spent to execute the proposed activities.

The four activities the users must carried out are the following:

1. Create a controller, register it in the Web server and implement the GET method. This method must return and HTML page containing an input field so that the user can insert a number and submit a form using the POST method.
2. Move all the HTML code written in the Controller of the previous activity to a HTML page. Modify the GET method so it calls this HTML page using the *CWF* functions.
3. Implement the POST method of the Controller to read the number sent through the input field, calculate the factorial of that number and return a HTML page with the result.
4. Use the Model View Controller pattern. Move all HTML code of the POST method to a View and use the CSTL to show the result.

In total, only 22 out of 1000 developers we invited agreed to be part of the *CWF* evaluation. During the experiment, the 22 participants received a two hour training in which the basis of general web development was explained in details, along with how to install, configure and implement dynamic web pages using the *CWF*. After that, the participants had two hours to install, configure and execute the four activities described before. Following they had to fill out an online form with profile details and the time spent in each activity. During the experiment, one participant had technical problems with his computer and one of them arrived late and missed the training part. The data collected on these two participants were discarded, hence we analysed the data on only 20 subjects.

The participant's profiles show the diversity of education, experience in software and web development. When it comes to education of the participants, 55% has a bachelor degree, 20% has also a certificate program degree, 20% has a MSc. degree and 5% has a PhD degree. Table 1 shows the experience of the

[6]Unfortunately, extensive documentation was not available since it is in early stages of development and feedback from this preliminary evaluation will be used to improve the design and tailor the user documentation.

Table 1: Experience of the participants (in years) when it comes to software, web and C++ development.

Type of exp.	>5y	3 - 5y	1 - 3y	<1y	None
Sw Dev.	65%	0%	20%	10%	5%
Web Dev.	10%	10%	25%	5%	50%
C++ Dev.	30%	5%	35%	10%	20%

Table 2: Time spent to execute the activities (in minutes).

Act.	<10	10-20	20-30	>30	Unsolved	Expect.
1	15%	50%	25%	10%	0%	20-30
2	15%	55%	15%	15%	0%	20-30
3	55%	30%	10%	5%	0%	10-20
4	15%	25%	20%	30%	10%	>30

participants regarding software development, web development and C++ development. Notice that, even though most participants has large experience in software development, most of them does not have much experience in web development.

After executing all proposed activities, all participants gave their opinion on how easy is to install and configure the *CWF* framework. They also informed how long they take to finish each activity. Around 25% of participants found the framework very easy to install and configure, 55% found it easy, and 20% found it moderately easy. We found this a satisfactory response since most developers do not have much experience with web development.

Table 2 shows the results when it comes to time spent to execute each one of the proposed activities. For the first activity, it was expected that the participants would take from 20 to 30 minutes because they had to install, configure and create a controller. Surprisingly, 65% of the participants spent less than 20 minutes to finish the activity. In the second activity, 70% took less than 20 minutes to provide a solution, less than the amount of time expected. For the third activity, 85% finished the activity within the expected time. The expected time to spent in the fourth activity was greater than 30 minutes since the participants had to refactor the application into a MVC architecture and use the CSTL. However, it was expected that they would take less than two hours. Fortunately, 60% of the participants finished the activity in less than 30 minutes, while 30% took more than 30 minutes but less than two hours, which was the total time they had to complete the four activities. In the forth activity, 10% of the participants were not able to finish to provide a satisfactory solution on time.

The number of samples are not enough to establish a significant correlation between the experience in software development and/or web development and the time spent to finish the proposed activities. However, a closer look at the data showed that experience does not necessarily influences the time the participants took to execute the proposed activities.

4.2 Load Testing

We compared the performance of a *CWF* application with the performance of a Java web application. We decided to compare the performance of our framework against the Java framework because, even though they have different purposes and are based on different set of technologies and components, because it is one of the most popular platforms to develop web applications when good performance is required (Millares, 2015). We also decided to deploy Java applications on the *Tomcat* because it is one of the most popular web server (Salnikov-Tarnovski, 2017). Our performance tests were inspired by the performance comparisons already used to evaluate web frameworks(TechEmpower, 2018).

The first three comparisons consisted in evaluating the performance of REST services designed to calculate the following mathematical routines: factorial, prime numbers and Fibonacci. The REST API should receive a request via *HTTP - GET* method, read the *number* parameter and process the math routine. After that, a JSON response should be created including the result in a specific attribute. The last comparison consisted in creating a REST services to generate a dynamic *HTML* page with 10 thousand chars. For each one of the four tests, 10 thousand requests have been generated for the web service to measure the processor and memory usage. All tests were executed three times and a report was built considering the mean of the three executions.

All tests were executed in a client-server environment using two computers in a local network. The server was a Laptop Samsung RV 411 running Ubuntu 17.10 (64 bits), SSD Kingston 120 GB, 8 GB RAM e and the Intel Core i5 M 480 2.67 GHz processor. On the server side, we used the Process Monitor GCC 5.3.1, Jemalloc, Qt 5.10.1, CWF Commit 174, Java 1.8.0, Tomcat 8.0.51. All Java code was optimized for the Tomcat server. On the C++ compiler, the optimizer flag was set to -03, which is the highest value. The client was a Laptop Samsung RV 420 running Deepin 15.4.1 (64 bits), SSD Kingston 120 GB, 8 GB RAM and the Intel (R) Celeron (R) CPU B800 1.50 GHz processor. On the client side, the Weighttp Commit 29 was used to perform the load testing. The router to connect each computer on the network is an Archer C7 Router AC 1750 - TP-LINK, and all computers used wire connections. Log functions were disabled in both the Java and the CWF server. Before each test, both the server and the client were restarted

Table 3: Results of the comparison considering the calculation of the Factorial (10).

	Test 1		Test 2		Test 3	
	CWF	**Java 8**	**CWF**	**Java 8**	**CWF**	**Java 8**
RAM (start) (KB)	792,00	105.779,20	796,00	105.062,40	792,00	104.038,40
RAM (end) (KB)	1.228,80	154.828,80	1.228,80	133.632,00	1.228,80	147.968,00
Processor % (start)	0,00	0,00	0,00	33,00	0,00	28,00
Processor % (end)	17,00	94,00	17,00	91,00	17,00	81,00
Response time (ms)	628,00	3.019,00	667,00	3.011,00	641,00	3.865,00

to make sure all tests had the same start point.

Table 3 shows the results obtained for the calculation of the factorial of the number 10. Table 4 shows the results of the algorithm that checks whether number 999 is a prime number. Table 5 shows the results of the test that consisted in finding the 20th number of the Fibonacci sequence. Notice that in all tests, the memory consumption at the *CWF* side was significantly lower than at the Java side. *CWF* also used much less processor than the Java application. In all cases, the response time of *CWF* applications was much faster.

In the last test, each application had to load a *HML* page with 10 thousand chars. Table 6 shows the results of this test. Although the memory consumption and the memory usage of the *CWF* application have been significantly lower than the Java application, there was no significant difference between the response time, with a slightly advantage of the *CWF* application.

It is very clear that the web application developed with the *CWF* uses less memory and processor than the Java application. This resulted was expected since a simple web server is run for each *CWF* application while Java applications run in heavy web servers such as Tomcat, for example. *CWF* is also faster to attend the requests. We believe it is important to show that the *CWF* applications consumes less resource than Java applications once it can lead to energy saving. Some studies pointed out that memory usage may represent more than 30 or 40% of the energy used by a server (Barroso and Hoelzle, 2009; Appuswamy et al., 2015). Nevertheless, further evaluation is required to know whether those results scale for bigger and more complex applications.

4.3 Real World Scenarios

Two real world applications were developed using the *CWF*. One application was developed by the creators of the *CWF* to deliver a solution to a private company. The other application was developed by an European organization. The name of both organizations and details on it are redacted due to non-disclose agreements.

The first usage is a REST service implemented in *CWF* that integrates the fiscal documents of 110 subsidiaries of a private company. This services has been running in this organization for at least 10 months now and it is performing well. The second usage was made by an European enterprise that develops software and prototypes new ideas. This organization used the *CWF* to develop four applications: two simple dynamic web pages; one application designed to observe, record, analyze and create an activity chronicle; and one application to test neural networks. Details on such scenarios cannot be presented in details due to to non-disclose agreements.

The fact that the *CWF* has been used to create web applications that are supporting real business operations is an evidence of its applicability in real world scenarios. Although such web applications have been running only for a few months, they have been delivering what was expected by their stakeholders.

4.4 GitHub Developers Community

The *CWF* is publicly available on GitHub[7], where developers can give a *star* to the projects they appreciate and make a copy (fork) of those they want to freely modify to accommodate their own requirements without affecting the original project. GiHub users can also submit *pull requests* to fix issues they found or to incorporate new features to the repository. In such cases, all modifications need to be verified and approved before the new feature or fix can be integrated into the original code.

So far, the *CWF* repository has received 344 stars and has been forked 91 times[8]. In addition, seven *pull requests* have been submitted by four distinct users. Even though the interest of the developers community in this framework is not expressive as for very popular web frameworks such as Django[9](55600 stars, 23900 forks) or Flask[10](53900 stars, 14100 forks) for

[7]https://github.com/HerikLyma/CPPWebFramework

[8]Data collected on February 16th of 2021.

[9]https://github.com/django/django

[10]https://github.com/pallets/flask

Table 4: Results of the comparison considering the calculation whether 997 is a prime number.

	Test 1		Test 2		Test 3	
	CWF	**Java 8**	**CWF**	**Java 8**	**CWF**	**Java 8**
RAM start (KB)	792,00	103.116,80	792,00	103.731,20	792,00	109.772,80
RAM end (KB)	1.433,60	141.004,80	1.126,40	158.105,60	1.126,40	150.528,00
Processor % (start)	0,00	0,00	0,00	0,00	0,00	0,00
Processor % (end)	14,00	56,00	17,00	75,00	17,00	59,00
Response time (ms)	649,00	3.178,00	671,00	2.979,00	667,00	2.899,00

Table 5: Results of the test considering the calculation of Fibonacci (20).

	Test 1		Test 2		Test 3	
	CWF	**Java 8**	**CWF**	**Java 8**	**CWF**	**Java 8**
RAM start (KB)	796,00	103.833,60	792,00	107.008,00	792,00	103.833,60
RAM end (KB)	1.433,60	137.420,80	1.228,80	138.342,40	1.228,80	131.276,80
Processor % (start)	0,00	52,00	0,00	52,00	0,00	39,00
Processor % (end)	19,00	70,00	20,00	62,00	20,00	67,00
Response time (ms)	741,00	3.500,00	739,00	3.379,00	715,00	3.163,00

Table 6: Results considering the loading of dynamic web pages with 10000 chars.

	Test 1		Test 2		Test 3	
	CWF	**Java 8**	**CWF**	**Java 8**	**CWF**	**Java 8**
RAM start (KB)	792,00	104.345,60	792,00	102.502,40	792,00	104.550,40
RAM end (KB)	884,00	292.044,80	892,00	283.955,20	884,00	345.292,80
Processor % (start)	0,00	13,00	0,00	34,00	0,00	28,00
Processor % (end)	10,00	51,00	11,00	58,00	10,00	44,00
Response time (ms)	9.157,00	9.400,00	9.199,00	9.482,00	9.223,00	9.503,00

Python, we believe that the number of *stars* and *forks* of the *CWF* repository may be an evidence of the growing interest of developers in creating web applications using the C++ language.

5 RELATED WORK

We have found in the literature initiatives to reduce the complexity of developing web applications using heavy frameworks available in the market. Many of them are focused on devising new abstract layers in which non expert web developers can easily develop and integrate specific applications, but using currently available web frameworks (Sinha et al., 2015; Swain et al., 2016).

Even though C++ is a popular and well established language that has high performance and low resource consumption (Game, 2021), few approaches have been proposed so far to support web development using the C++ language (Obiltschnig, 2005; Frings, 2010; Dumon and Deforche, 2008). Most of such approaches entangle business, control and presentation layer to make web development simple, but, over time, the application tend to become less adaptable, maintainable and reusable (Srai et al., 2017).

Particularly, one common approach is to run CGI scripts in the Apache Server to generate web pages dynamically. The drawback of such an approach is that the HTML pages are generated within the C++ files, which is not a good practice since the presentation and business layer will be entangled making development, maintenance and testing difficult. The same limitation afflicts the CSP (C++ Server Page) and POCO (Obiltschnig, 2005) approaches, once they propose to include C++ programming into the presentation layer. In WebToolKit (Dumon and Deforche, 2008), everything is coded by the components of the framework, which in turn generate the HTML files at each request. Likewise, the presentation and business layers are entangled.

The QtWebApp (Frings, 2010) is the web framework that is most similar to the *CWF*. The main difference between the QtWebAPP and the *CWF* is the CSTL proposed along with the *CWF*. The QtWebApp also uses tags to develop dynamic web pages, but they are very limited. They cannot handle iterations, collections of objects and conditional sentences properly. Moreover, each tag must be previously configured within a C++ file to define the behavior of the

tag, which makes the implementation and the maintenance more difficult than when a specification such as the CSTL is available.

In addition to solutions based on C++, several lightweight frameworks have been developed over the years (Rogowski, 2017). For instances, Vapor and Kitura (Patel, 2018) (Swift), Ruby on Rails, Django and Flask (Python) are frameworks that provide developers with a foundation for the development of web applications, APIs, or cloud platforms projects. Such frameworks are also intended to reduce the complexity of web development by proposing simple structures that allows developers to focus on writing the application rather than on configuration files, libraries, and so forth. Indeed, these frameworks leverage good practices of web development by promoting suitable architectural styles and patterns that promote the separation of concerns that render applications more maintainable, such as the Model View Controller (MVC).

As presented in this paper, the *CWF* is also intended to make web development easier by allowing developers to focus on coding the business rules and by extending base classes provided by the web framework. We did not compare the *CWF* with those lightweight web frameworks because it was out of the scope of our preliminary investigation and because they use languages interpreted such as Ruby and Python. While these languages are flexible and easy to use, they can be up to 300 times slower than C++ (Game, 2021).

6 CONCLUSIONS

This paper presented a new web framework called *CWF*, whose main purpose is to support the development of web applications that combine the high performance of the C++ language and the flexibility of the Qt framework. This framework requires only one configuration file, which makes it easier to create web applications. It also provides a tag library called CSTL (C++ Server Pages Standard Tag Library), which generates dynamic web pages and keeps the presentation and the business layer separated.

Preliminary evaluation shows that, at least when executing simple applications, the *CWF* outperforms Java applications in the Tomcat web server. The user evaluation gives evidence that the *CWF* is easy to use and understand since many subjects were able to create simple and complex web applications in C++ using the *CWF*. The *CWF* was also used to develop two real world applications that has been running to support business activities for several months.

We believe that the *CWF* is an appealing web framework for those who want to develop applications with simplicity and high performance, especially C++ developers. Nevertheless, the power of the C++ language combined with the richness of the Qt framework is promising to provide developers with an alternative to develop robust complex web applications, and yet with high performance. The *CWF* might also be very useful for researchers that write C++ code and eventually need to expose operations online but cannot afford investing time to learn new languages or web frameworks.

In future work, we intend to further evaluate the *CWF* by building larger and more complex applications and by conducting further evaluation regarding the *CWF* usability for both specialized and novel web application developers. Furthermore, we aim at investigating the advantages and the drawbacks of evolving web applications that was built based on the *CWF*. In addition, we aim at producing a detailed documentation to provide web developers with guidance on the development of web applications since it is key factor for the usability of the framework (Constanzo and Casas, 2019). Finally, we intend to compare the *CWF* with other lightweight frameworks, such as Vapor, Django, Ruby Rails and Flask.

REFERENCES

Appuswamy, R., Olma, M., and Ailamaki, A. (2015). Scaling the memory power wall with dram-aware data management. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, New York, NY, USA. Association for Computing Machinery.

Barroso, L. and Hoelzle, U. (2009). The datacenter as a computer: An introduction to the design of warehouse-scale. *Morgan Claypool*.

Buyya, R., Yeo, C., Venugopal, S., Broberg, J., and Brandic, I. (2009). Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25:599–616.

Charland, A. and Leroux., B. (2011). Mobile application development: Web vs. native. *Future Generation Computer Systems*, 9.

Chaubey, R. and Suresh, J. K. (2001). Integration vs. development: an engineering approach to building web applications. *IEEE Multimidia*, pages 171–181.

Constanzo, M. A. and Casas, S. (2016). Usability evaluation of web support frameworks. In *2016 XLII Latin American Computing Conference (CLEI)*, pages 1–6.

Constanzo, M. A. and Casas, S. (2019). Problem identification of usability of web frameworks. In *2019 38th In-*

ternational Conference of the Chilean Computer Science Society (SCCC), pages 1–8.

del Pilar Salas-Zarate, M., Alor-Hernandez, G., Valencia-Garcia, R., Rodriguez-Mazahua, L., Rodriguez-Gonzalez, A., and Cuadrado, J. L. L. (2015). Analyzing best practices on web development frameworks: The lift approach. *Science of Computer Programming*, 102:1–19.

Dumon, W. and Deforche, K. (2008). Wt: A web toolkit. *Dr. Dobb's Journal*, 33:55–59.

Freitas, F. and Maia, P. H. M. (2016). A naked objects based framework for developing android business applications. In *Proceedings of the 18th International Conference on Enterprise Information Systems - Volume 1: ICEIS,*, pages 348–358. INSTICC, SciTePress.

Frings, S. (2010). Qtwebapp http webserver in c++.

Game, T. C. L. B. (2021). The computer language benchmarks game.

Hundt, R. (2011). Loop recognition in c++/java/go/scala. Technical report, Google, 1600 Amphitheatre Parkway Mountain View, CA, 94043.

Millares, G. (2015). Top 5 programming languages used in web development.

Obiltschnig, G. (2005). Poco c++ libraries.

Patel, A. (2018). *Hands-On Server-Side Web Development with Swift*. Packt Publishing.

Raemaekers, S., van Deursen, A., and Visser, J. (2014). Semantic versioning versus breaking changes: A study of the maven repository. *IEEE International Working Conference on Source Code Analysis and Manipulation*.

Ramana, U. V. and Prabhakar, T. V. (2005). Some experiments with the performance of lamp architecture. *IEEE*, pages 916–920.

Rogowski, J. (2017). The comparison of the web application development frameworks. *ITCM*.

Salnikov-Tarnovski, N. (2017). Most popular java application servers (2017 edition).

Shklar, L. and Rosen, R. (2004). *Web Application Architecture Principles, protocols and practices*. Wiley.

Sinha, N., Karim, R., and Gupta, M. (2015). Simplifying web programming. In *Proceedings of the 8th India Software Engineering Conference, ISEC 2015, Bangalore, India, February 18-20, 2015*, pages 80–89.

Srai, A., Guerouate, F., Berbiche, N., and Lahsini, H. (2017). Applying mda approach for spring mvc framework. *International Journal of Applied Engineering Research*, 12:4372–4381.

Swain, N. R., Christensen, S. D., Snow, A. D., Dolder, H., Espinoza-Dávalos, G., Goharian, E., Jones, N. L., Nelson, E. J., Ames, D. P., and Burian, S. J. (2016). A new open source platform for lowering the barrier for environmental web app development. *Environmental Modelling & Software*, 85(1):11–26.

TechEmpower (2018). Techempower web framework benchmarks.

TIOBE (2021). The tiobe programming community index for february 2021.

Tsai, W., Bai, X., and Huang, Y. (2014). Software-as-a-service (saas): Perspectives and challenges. *Science China Information Sciences*, 57(5):1–15.

Tuan, A. D., Comyn-Wattiau, I., and Cherfi, S. S.-S. (2016). Structuring guidelines for web application designers. In *Proceedings of the 18th International Conference on Enterprise Information Systems - Volume 1: ICEIS,*, pages 327–335. INSTICC, SciTePress.

Vuorimaa, P., Laine, M., Litvinova, E., and Shestakov, D. (2016). Leveraging declarative languages in web application development. *Springer*, 19.