

Engineering thesis

Michał Stefaniuk

major: **applied computer science**

Web application development using WebToolkit C++ on the example of a banking service

Supervisor: **Dr Grzegorz Gach**

Cracow, January 2021

Oświadczenie studenta

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.”, a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.”, oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

Streszczenie pracy

Celem niniejszej pracy dyplomowej było zaimplementowanie aplikacji pozwalającej wykorzystać w procesie tworzenia rozmaite narzędzia programistyczne. Narzędzia, o których mowa to przede wszystkim programy wspomagające inżynierię oprogramowania i są one wykorzystywane w dzisiejszych czasach przez wielu programistów.

Tematem programu jest internetowa aplikacja bankowa umożliwiająca trywialne funkcjonalności takiego serwisu, jak na przykład przelewanie pieniędzy czy przeglądanie logów transakcyjnych przez administratora. Całość została zaimplementowana w języku C++ ze wsparciem biblioteki WebToolkit. Narzędzia, które wykorzystano do wspomoczenia procesu inżynierii oprogramowania to system kontroli wersji GitHub, automatyczny serwer regresyjny GitHub oraz oprogramowanie do śledzenia postępów i zarządzania projektami JIRA.

Abstract

The goal of this thesis was implementing an application that would allow to use various software engineering tools during the development process. Those tools are especially the ones that are enhancing the process and nowadays they are used by almost every programmer.

The topic of the program is a web application as a banking service that provides simple features like wiring money or going through service logs to the users and admin. The code was written in C++ language with support of Web Toolkit library. The tools used in this project to conduct the software engineering processes were version control system GitHub, automatic regression GitHub server and issue tracking tool JIRA.

Contents

Introduction	7
1 Projects assumptions	9
1.1 Application's blue-print	9
1.2 Programming environment	9
2 WebToolkit C++ library	15
2.1 Library overview	15
2.2 WebToolkit Features	17
2.3 Hello World Application	20
2.4 Widgets gallery	22
3 DevOps layer	23
3.1 Introduction	23
3.2 Distributed version-control system	24
3.3 GitHub	25
3.4 Proprietary issue tracking	26
3.4.1 JIRA Software	27
3.5 Automation server and regression testing	29
3.5.1 GitHub CMake	30
4 Implementation	37
4.1 Web Banking core class	38
4.2 Widgets	40
4.3 Session	46
4.4 User entity	48
4.5 Database	50
5 Summary	53

Introduction

Software engineering is a very wide area of engineering which particularly concerns developing and maintaining programming products. A development process itself is a major challenge to all people involved, starting with engineers creating the code, continuing with product owners who are managing teams and coordinating the work flow and ending-up with managers who are setting the direction of the whole process.

The main motivation to create this thesis was simply to present a development process that includes basic and nowadays necessary tools which are significantly helpful in such process. Simultaneously creating a C++ web application was an equally important factor. To carry out the development and present all tools it was decided that an example C++ web application will be created with a GUI library in modern C++ called **Web Toolkit**.

Nowadays in a software engineering world there is a trend to migrate desktop applications to the internet, which has also impacted a lot GUI desktop libraries in decreasing their utilities. The secondary reason behind choosing C++ to create a web application, which is quite unusual, was to show the tremendous possibilities that this language still provides and to exhibit the capabilities of open source libraries shared among developers and engineers. The bottom line is - next chapters are a review of possibilities that C++-based web library provides.

Chapter 1

Projects assumptions

1.1 Application's blue-print

The application is a single-page service run on a local server with an **http://localhost:8080/** address. After starting, logging panel appears, giving the user an opportunity to type their credentials and log-in.

The idea of the service is to provide basic functionalities of a banking service like for instance transferring money or checking their balance to the user. These required implementation of secure back-end layer with a database containing information about the users and transactions and a front-end layer which is transparent to the user.

Regarding the credentials a user will see different features after logging into the service. There might be many accounts created and held up in the database of the service but there are only two rights of access.

USER Transferring money and seeing current account's balance.

ADMIN Seeing details about every account in the service as well as having access to the logs from the application.

After user decides to quit the service, a possibility of logging out is provided and whilst doing that the application returns to the logging page where session is refreshed and database is updated. If the user was an administrator of the service, an access to the console logs that are gathering and sniffing network traffic like HTTP methods for RESTful APIs is also granted.

1.2 Programming environment

Developing any kind of programs usually requires specific environmental variables therefore a programmer's task is to choose a specific set of tools which will be the most handy during the process.

Regarding the specifics of the project, the chosen set of tools looks like following:

IDE Microsoft Visual Studio Community 2019,

LIBRARY Web Toolkit 4.3.1,

LIBRARY Boost C++ lib [latest version],

TOOL CMake VERSION 2.4,

TOOL Git BASH.

The application was developed under Windows 10 64-bit operating system, but it could have been developed under any other platform that is supported by WebToolkit library.

IDE - Integrated Development Environment

IDE is a program or a set of programs merged into one that usually gathers tools, libraries, debuggers, run time scripts and any other stuff necessary for the developer to write the code. The purpose of IDE is to allow easily and swiftly create code, but at the same time test it, compile it and run it in one place.

The advantage of this solution is that it gives the developer an opportunity to set up and scale his development environment adjusting it to the project needs. Usually IDE, as well as modern text editors, also support plugins which are nice addition to the program, like for instance syntax highlighters, semantics hints, code analyzers or refactoring scripts.

The chosen IDE for this project is **Microsoft Visual Studio Community 2019**. Web Toolkit library is supported on various platforms including Linux distributions (even less popular ones like ArchLinux, Slackware or Opensuse), Windows or other operating systems like Android, Raspberry Pi or even OS X. The consequence of choosing MSV IDE was using Windows platform and prebuilt Windows binaries of the WT library.

Microsoft Visual Studio is an IDE produced by Microsoft Company and it allows creating cross-platform software with graphical user interface. It basically supports every programming language but the basic package contains support for

- Microsoft Visual C,#
- Microsoft Visual C++,
- Microsoft Visual Basic,
- Microsoft Visual J,#
- Microsoft Visual Web Developer ASP.NET,
- Microsoft Visual F.#

Microsoft Visual Studio (MSVC) also provides a lot of built-in features. Many of them occurred to be significantly useful in the process of developing this application. A few most important ones are

- debugger, linker and compiler,
- projects and build systems,
- writing and refactoring C++ code,
- code analysis overview,
- unit tests support,
- universal windows apps like command line applications.

Boost C++ library

Boost is a collection of C++ libraries that enhances capabilities of C++ code development, which is also licensed by **Boost Software License**. For the project Boost is necessary to build Web Toolkit library as it's implementation uses Boost functionalities.

The most important features provided by Boost are

- algorithms,
- concurrent programming,
- complex containers,
- correctness validating and enhanced unit testing,
- additional data structures (like bimap, fusion, tuple etc),
- high level programming and functional objects,
- parsers and graphs,
- meta-programming with templates.

Boost library as a collection is not used in this project explicitly, as the application was developed in Microsoft Visual Studio environment with pre-built Windows binaries. However, for automation server, the project is built with linux machine where WebToolkit needs to be built manually, therefore boost is required.

CMake

CMake is a cross platform tool that provides automatic management of compiler that builds the code of an application. It's role is to create a configuration for project files of popular programming environments, which then are used in a process of compilation. The main advantage of using CMake is it's independence of the compiler and the platform. Moreover CMake as a standalone program creates files with rules for compilation dedicated to another program like IDE and it forms a unified building environment. CMake stands for *Cross-platform Make*.

The most important features provided by CMake are

- platform independence,
- cross compilation,
- out-of-source building,
- building projects with complex catalog structure,
- unit testing support,
- detecting dependencies and outer libraries.

This project required including CMake tool because of various dependencies and complex building due to including Web Toolkit and Boost.

To be able to explore favors that CMake offers one must create **CMakeLists.txt** file placed in the desired catalog of the project. The core of this file is a simple scripting language that describes rules and defines variables telling the compiler how to link files and what should be the outcome of the compilation process.

Let's have an example project[1] which structure looks like below 1.1.

```

1  .
2  |-- CMakeLists.txt
3  |-- build
4  |-- include
5  |   |-- Student.h
6  |-- src
7  |   |-- Student.cpp
8  |   |-- mainapp.cpp

```

Listing 1.1: Example CMake project structure.

The content of ***.cpp** and ***.h** files is irrelevant here, at this point it could be any code. In the main directory there is a simple **CMakeLists.txt** which is used to build the project. The code of the **CMakeLists** script 1.2:

```

1  cmake_minimum_required(VERSION 2.8.9)
2  project(directory_test)
3
4  #Specifies include directories to use when compiling a given target
5  include_directories(include)
6
7  #Can manually add the sources using the set command as follows:
8  #set(SOURCES src/mainapp.cpp src/Student.cpp)
9
10 #However, the file(GLOB...) allows for wildcard additions:
11 file(GLOB SOURCES "src/*.cpp")
12
13 add_executable(testStudent ${SOURCES})

```

Listing 1.2: CMakeLists.txt of example project.

The most important things while building the project that above CMake will perform (in order from the top) are

- Include directories to specify headers files directories
- Sources are also set, but the line is commented out as each file needs to be manually added in place
- file() command to add source files to project's GLOB SOURCES

- `add_executable()` which uses `SOURCES` variable in order to build executable program

Now to invoke building using CMake:

```
1  $ cd build
2  $ cmake ..
3  $ make
4  $ ./testStudent
```

First current directory is changed to `build`, where output files will be stored. From there `cmake` will create **Makefile** containing references to all sources and headers, which ultimately will create an executable that is ready to be run. This as a very basic example of how **CMake** works. In next chapters there will be more details about **CMake** and what is the main concept behind basing on the one used in project.

Chapter 2

WebToolkit C++ library

2.1 Library overview

Web Toolkit is an open-source C++ library that allows developers to create Single Page web applications without writing single line of JavaScript code. The main concept behind SAP [2](Single Page Applications) is they do not render HTML files, but instead they use asynchronous JavaScript to reload contents of the page in real time without refreshing the whole page. To some extent it might be a bit misleading, as with loading different content, the URL of the page changes as well, like in ordinary multi-page service.



Figure 2.1: Traditional Page vs SPA.

As a SPA, web-banking application works on widgets which are the core of WebToolkit library. A widget itself is a specific type of user-computer interaction with a graphical interface such as button or a scroll bar, which in WebToolkit case is wrapped in C++ code. A developer should create a main widget, which is not visible to the user. Instead it works as a widget container that will hold other template widgets that will be plugged in or plugged out depending on what content will be loaded on the page. This logic is presented on figure 2.1 and 2.2.

There are a lot of modern services that use SPA technology like facebook [7], airbnb [8], twitter [9], paypal [10], gmail [11] or even netflix [12]. The biggest benefits of this web application model are quick loading time, good caching abilities, improved user experience or rapid front-end development.

The most important thing about Single Page Application model is very rapid and fast user experience as the page does not refresh or re-render. However, since the page works on widgets

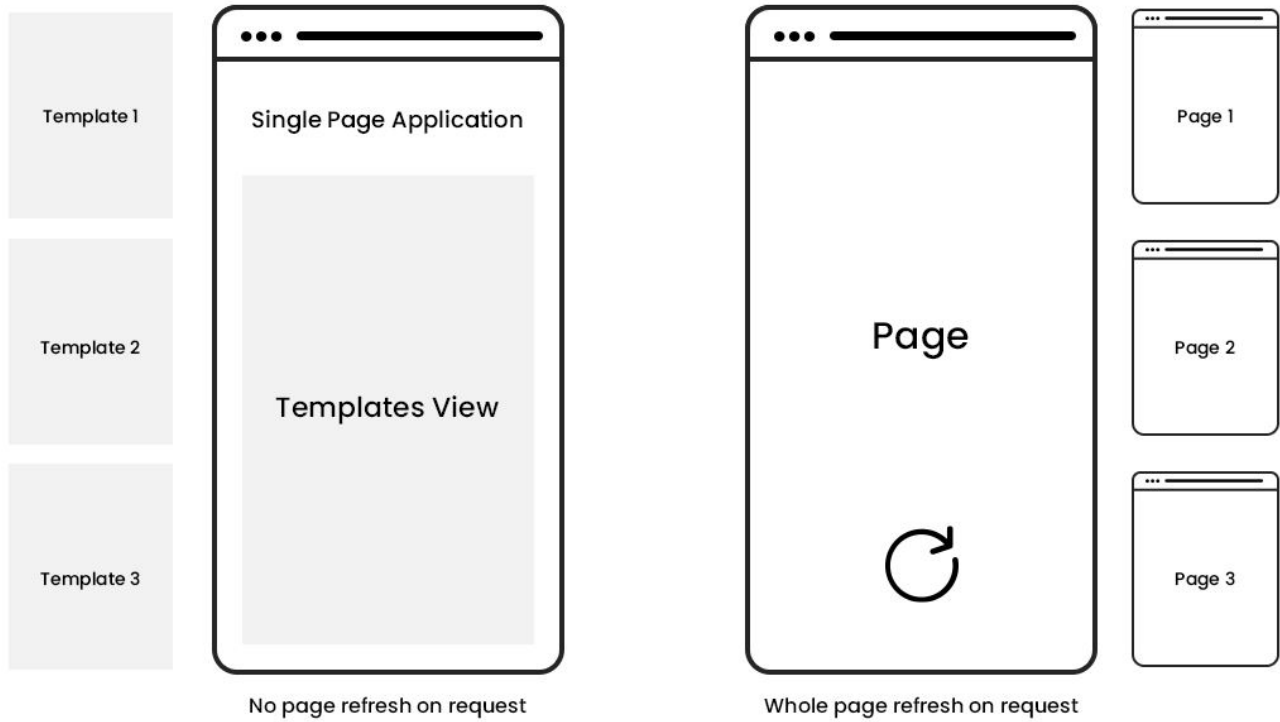


Figure 2.2: Views in Traditional Page and SPA[2].

usually the HTML contents of the page is rather empty. Often it's just a few tags, scripts and imports while the rest of the contents is loaded dynamically for instance by asynchronous javascript. This is a big disadvantage when it comes to search engines that are using crawlers. Instead of seeing the web contents they will see a short HTML file with not too much to process, since crawlers never wait for asynchronously loaded data. This SEO problem makes Single Page applications a bad choice for services relying on search engines and positioning.

On the other hand a regular static page has a lot to offer too. It serves HTML file instantly, but it requires refreshing and rendering new page. It reduces the amount of javascript code that is needed but it also allows crawlers to see the same content that users can see. The disadvantage of this type of page should be pretty obvious now - it allows to create services with limited dynamic content. It might be a poor choice to go for when a service needs to serve content that changes quite frequently.

Beyond usual frameworks dedicated for SPA services are

- Angular.js,
- React.js,
- Backbone.js,
- Ember.js.

It is no secret that preferable programming technology stack for such development would be JavaScript [13], TypeScript [14], HTML [15], CSS [16] and some back-end in Java [17], Python [18] or PHP [19]. With that being said, WebToolkit library seems to be very unorthodox tool for this task. The whole development process starts and ends with C++ code, which makes it interesting and unusual.

2.2 WebToolkit Features

Web Toolkit library has a lot to offer. Now of course - creating web application in C++ language comes with a prize. It's biggest disadvantage is that while programming we have no control of HTML code. There is also a necessity of re-compiling the code with every change introduced where there is no such need when using standard web development tools (like javascript application with node.js or php application with XAMPP). However, having said that it is worth mentioning that Web Toolkit 2.3 is appreciated for plenty features that it provides to the users.



Figure 2.3: WebToolkit logo and features [3].

Core Library

What is really important is that core library is fully open source and still under development. It's implementation allows to integrate it with third party JavaScript libraries, which is a nice addition for developers that create services with more advanced front-end layer. What is more, Web Toolkit is fully compatible with HTML5 [20] and HTML4 [21] browsers. Thus, as a hybrid single page framework it supports browser history navigation. The bottom line is, WT is a high performance library which is something you would expect from a C++ library. It is optimized when it comes to asynchronous I/O, multi-threading and throughout all rendering.

Event handling is completed with C++11 lambdas [22] or bound object methods to retrieve data that is bound to a function. Those are a part of C++11 signal/slot API for responding to various events. Server-initiated updates are also available through WebSockets and automatic fallback to AJAX. Event handling itself is also completed with efficient synchronization of browser using session, which incrementally renders updates of application states.

Core library also provides 2D and 3D painting support to users and developers. Through Web Toolkit API it is possible to generate simple graphics objects like HTML5 canvas, inline SVG [24] or inline VML. Graphics is also leveraged by rendering common image formats like PNG, GIF or even PDF. WT broads the variety of graphics support by also supporting hardware-accelerated 3D painting API like for instance server-side OpenGL [23].

Security

As a completed web-development library, WebToolkit provides built-in security layer to the applications which contains:

- kernel-level memory protection in isolated sessions,

- TLS/SSL support,
- Cross-Site Scripting (XSS) prevention,
- Request Forgery (CSRF) prevention,
- Application logic attack prevention,
- DoS mitigation,
- **Authentication module** (including support for OAuth 2.0 and OpenID Connect) figure 2.4.

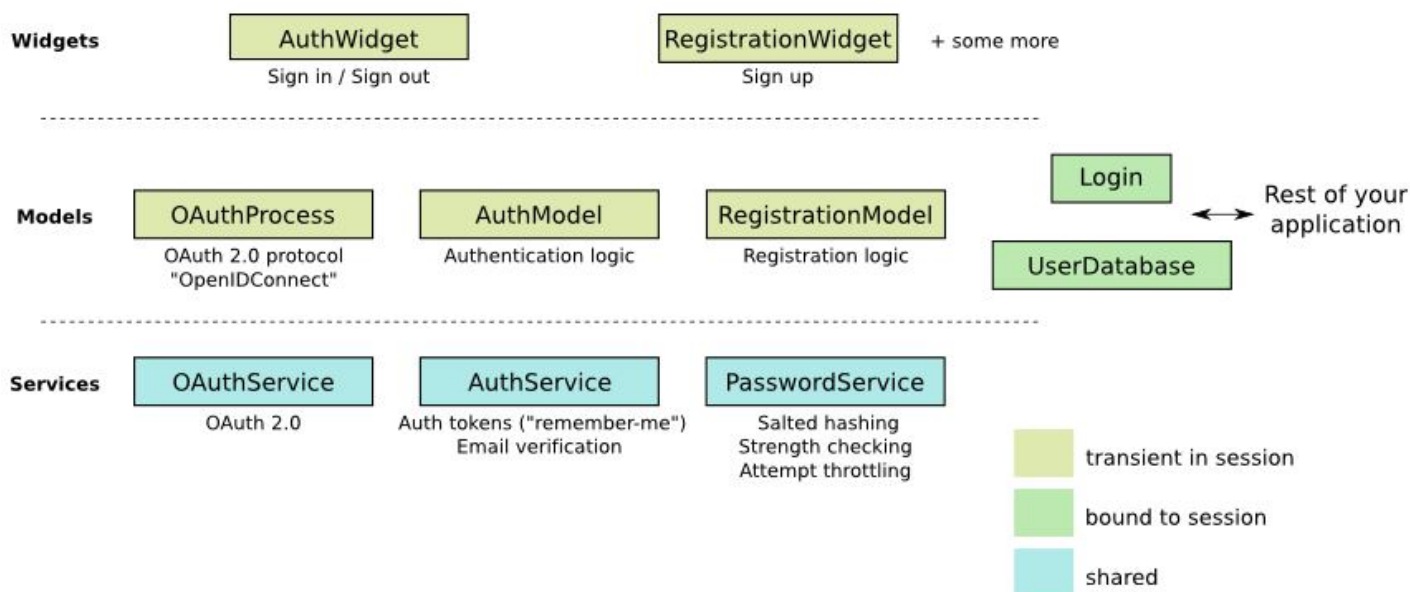


Figure 2.4: Main classes of the authentication module[4].

The authentication module itself occurred to be significantly important and helpful in Web Banking Application. Some of the classes which implement basic user features are presented above. What it means is that secure and robust service functionalities like signing in/out, registration or authentication are already provided.

Database support

As any other modern library, WT supports databases with a sub-library called `Wt::Dbo`, which by implementing ORM (Object-Relational Mapper) becomes a convenient way to communicate C++ application with SQL databases. What is more, `Wt::Dbo` as a self-contained sub-library is independent from `Wt` itself, which means it could be used for standalone back-end service. Other basic `Wt::Dbo` features are:

- Support for
 - SQLite3
 - Firebird
 - MariaDB

- MySQL
 - SQL Server
 - PostgreSQL
- Native SQL to query objects and fields
- Session with first-level cache
- DB transactions
- CRUD operations
- Many-to-One and Many-to-Many relations
- Just C++ without XMLs or Macros

Deployment

Extremely important thing in any application is process of Deployment. WT through various abstract interfaces connects with outer world. The basic one is built-in httpd but among others:

- **Built-in HTTPD**
 - Simple and high performance with multithreading and asynchronous I/O based on C++ asio library
 - Supports WebSockets and HTTP(s) with chunking and compression
 - Available for many platforms like Linux, Windows and any UNIX like systems
- **FASTCGI**
 - Integrated with common web servers like apache
 - Supported for Linux and UNIX
- **ISAPI**
 - Integrated with Microsoft IIS server
 - Completed with asynchronous API
 - Supported for Windows

2.3 Hello World Application

Lets go through simple Hello World application (listing 2.1) just to get in touch with WebToolkit library environment.

```

1 #include <Wt/WBreak.h>
2 #include <Wt/WContainerWidget.h>
3 #include <Wt/WLineEdit.h>
4 #include <Wt/WPushButton.h>
5 #include <Wt/WText.h>
6
7 class HelloApplication : public Wt::WApplication
8 {
9 public:
10     HelloApplication(const Wt::WEnvironment& env);
11
12 private:
13     Wt::WLineEdit *nameEdit_;
14     Wt::WText *greeting_;
15 };
16
17 HelloApplication::HelloApplication(const Wt::WEnvironment& env)
18     : Wt::WApplication(env)
19 {
20     setTitle("Hello world");
21
22     root()->addWidget(std::make_unique<Wt::WText>("Your name, please? "));
23     nameEdit_ = root()->addWidget(std::make_unique<Wt::WLineEdit>());
24
25     Wt::WPushButton *button =
26         root()->addWidget(std::make_unique<Wt::WPushButton>("Greet me. "));
27
28     root()->addWidget(std::make_unique<Wt::WBreak>());
29     greeting_ = root()->addWidget(std::make_unique<Wt::WText>());
30
31     auto greet = [this]{
32         greeting_->setText("Hello there, " + nameEdit_->text());
33     };
34     button->clicked().connect(greet);
35 }
36
37 int main(int argc, char **argv)
38 {
39     return Wt::WRun(argc, argv, [](const Wt::WEnvironment& env) {
40         return std::make_unique<HelloApplication>(env);
41     });
42 }

```

Listing 2.1: A complete "Hello world" application [5].

Now when it comes to building, it could be done locally by setting up a localhost HTTP server. Having IDE set-up it is done automatically. Otherwise while operating on UNIX-like systems it would require a few commands presented on listing 2.2.

```
1 $ g++ -std=c++14 -o hello hello.cc -lwthttp -lwt
2 $ ./hello --docroot . --http-address 0.0.0.0 --http-port 9090
```

Listing 2.2: Building "Hello world" application.

The final result is shown in figure 2.5

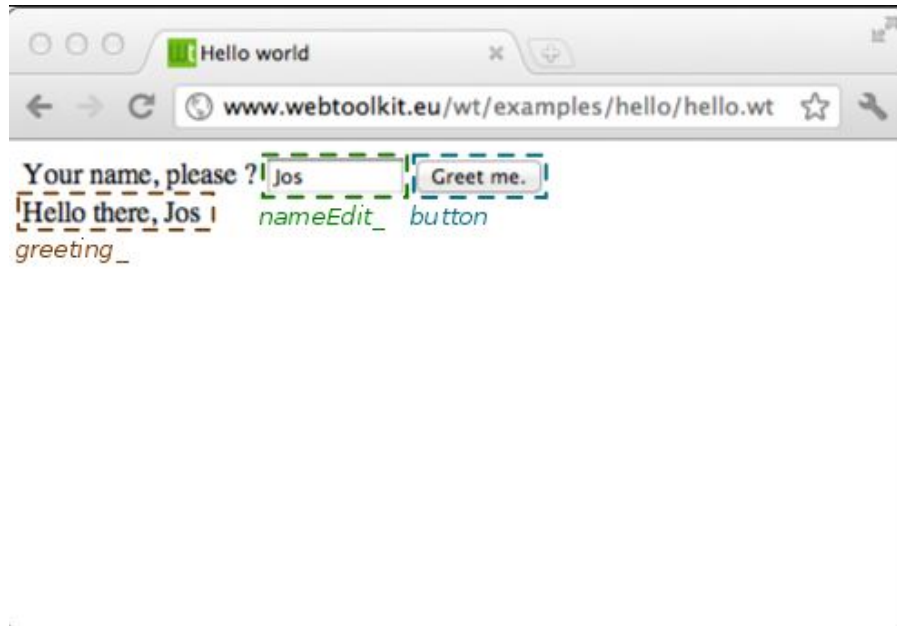


Figure 2.5: Hello Application on Web Toolkit.

The above code presents basic concepts of the library workflow.

class HelloApplication A class that represents application and extends `Wt::WApplication`.

Contains ctor that takes `Wt::WEnvironment` and private fields that represents input field and output greeting string wrapped in `Wt::WText`.

LINE 20: Sets title of appiaction which will be visible in web browser's card.

LINE 22-29: This part of the code handles adding widgets to the root which represents a container for the widgets. This is an implementation of SPA concept where we create widgets and plug them in or out, for instance by `addWidget()` function.

LINE 31: Lambda expression that returns `WText` string which is called upon clicking the button.

LINE 34: Calling lambda expression from line 31 when button is clicked.

MAIN: Creating and running application.

2.4 Widgets gallery

Web Toolkit also offers a widgets gallery, which is nothing else but a bootstrap where most common widgets are already implemented and ready to be used. Among them there are layouts, forms widgets, navigation widgets, trees widgets, tables widgets, graphics widgets, charts widgets and media widgets. Basically whilst going through development process one should keep in mind this feature as it may save a lot of time and a lot of coding. The layout of widgets gallery is presented in figure 2.6

The screenshot shows the Web Toolkit Widget Gallery interface. At the top, there is a navigation bar with tabs: 'Wt Widget Gallery', 'Layout', 'Forms' (selected), 'Navigation', 'Trees & Tables', 'Graphics & Charts', and 'Media'. Below the navigation bar, the title 'Forms — widgets and support classes for capturing user information' is displayed. On the left side, there is a sidebar with a list of widget categories: 'Introduction', 'Line/Text editor', 'Check boxes', 'Radio buttons', 'Combo box', 'Selection box', 'Autocomplete', 'Date & Time entry' (highlighted in blue), 'In-place edit', 'Slider', 'Progress bar', 'File upload', 'Push button', 'Validation', and 'Integration example'. The main content area is titled 'Date entry' and contains the following text: 'Wt provides to kinds of widgets to enter a date, namely `WCalendar` and `WDatePicker`.' Below this, it says 'Date entry using a calendar' and 'The `WCalendar` widget provides navigation by month and year, and indicates the current day.' An 'Example' section shows a screenshot of the `WCalendar` widget displaying the month of November 2020, with the 5th of the month highlighted. Below the example, there is a 'source' section with the following C++ code:

```
source
#include <Wt/WCalendar.h>
#include <Wt/WContainerWidget.h>
#include <Wt/WText.h>

auto container = Wt::cpp14::make_unique<Wt::WContainerWidget>();
Wt::WCalendar *c1 = container->addNew<Wt::WCalendar>();
```

Figure 2.6: Web Toolkit Widget Gallery [6].

Chapter 3

DevOps layer

3.1 Introduction

DevOps stands for development and operations and it is a modern approach for software engineering. Its core is to integrate developer teams with other sections involved in the process. According to this definition, DevOps engineer is someone who is fluent with development cycle and it is also someone who can improve the process. Naturally there are a lot of cutting-edge tools which are significantly helpful in DevOps engineering.

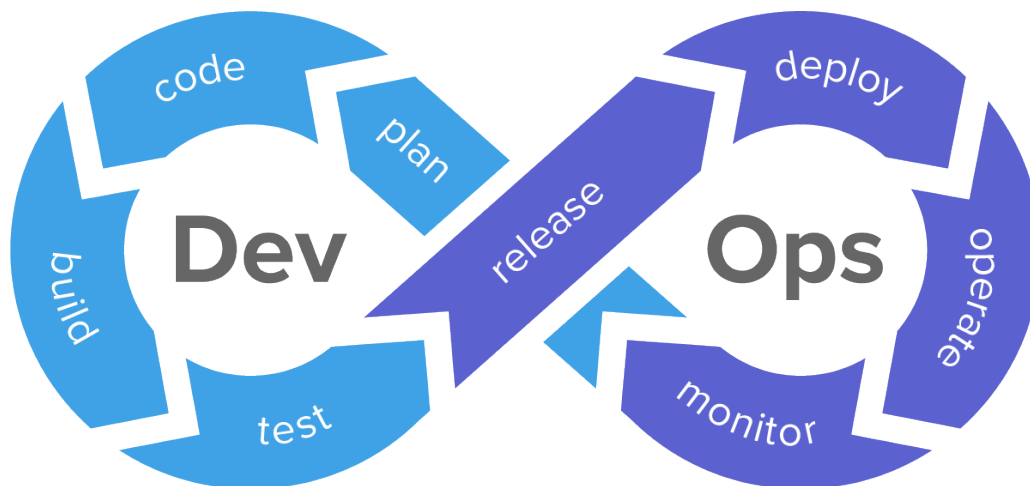


Figure 3.1: DevOps process. [25].

Figure 3.1 shows basic elements of devOps approach which are

- building the product that is being developed,
- coding,
- planning the whole process,
- monitoring the development,
- operating,
- deploying for instance to cloud,
- releasing to client,

- and testing throughout the whole time.

Creating web banking application was also about carrying out the development process based on DevOps approach.

3.2 Distributed version-control system

First tool that was used in this project was version-control system called **Git**. It is a tool that allows developers to track changes in code of the project, as well as making them. Every change introduces a new version of the code which is stored in history. The biggest advantage of this feature is capability of restoring previous version of the code which presents tremendous enhancement to reponsing to unwanted changes.

What is more, Git provides branching feature, which simplifies developing code simultaneously by many developers at the same time. By invoking simple command line functions developer can communicate his local repository with remote repository. This workflow is presented on figure 3.2. Many version-control systems have graphic interfaces too.

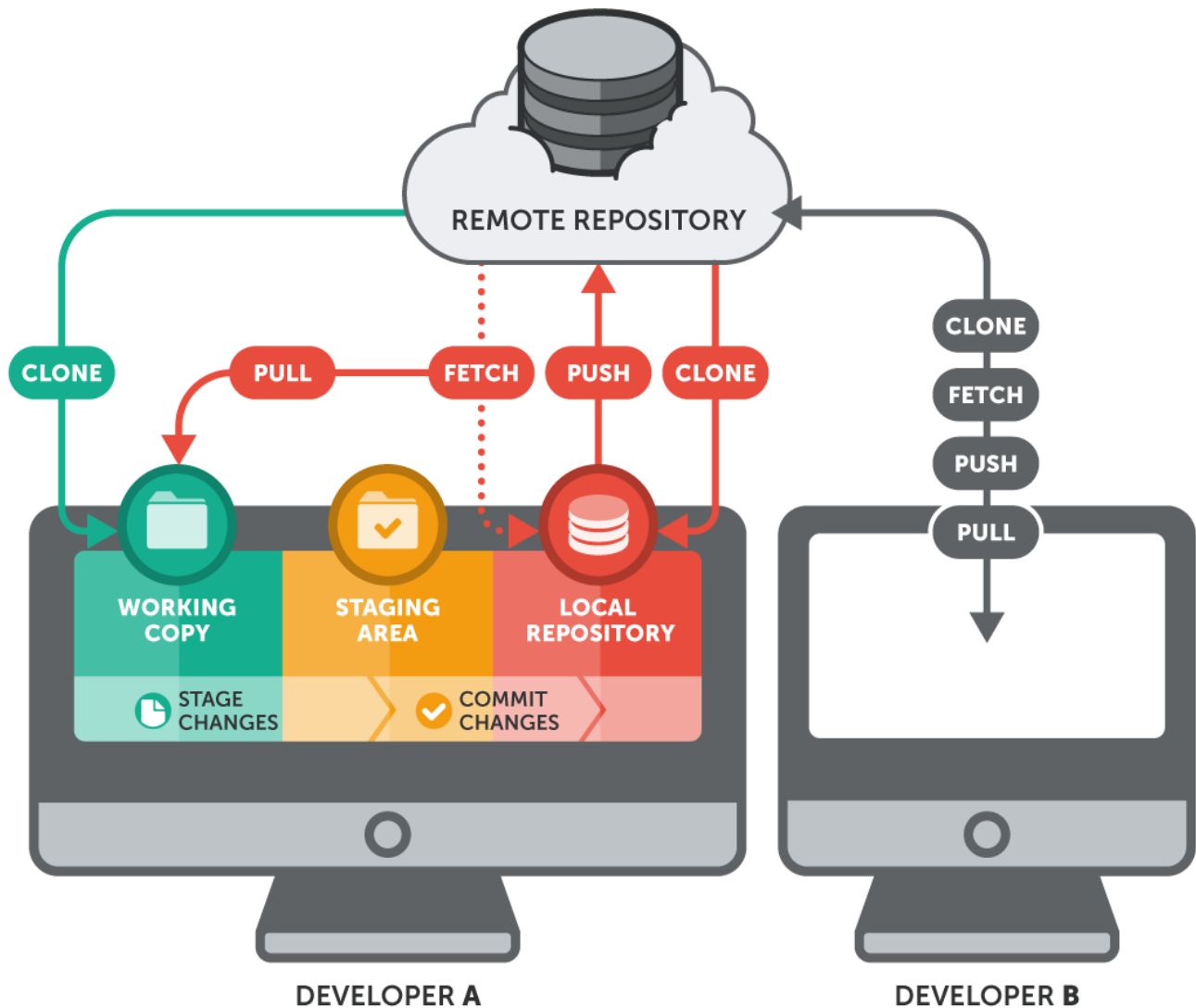


Figure 3.2: Distributed version-control system workflow [26].

3.3 GitHub

Web banking application was being developed using GitHub, which is a GIT version-control system hosting dedicated for programming projects - it's layout is presented on figure 3.3. A remote repository was stored there, which contained:

- applications code,
- CMake building scripts,
- YAML docker images for automation server,
- web toolkit library,
- static resources like images,
- unit and regression tests,
- this theesis,
- and most importantly - information about all changes introduced to the project.

Creating this application required over 150 commits, 18 branches and more than 20 pull requests. More statistics regarding percentage of programming languages used in project are presented on figure 3.4.

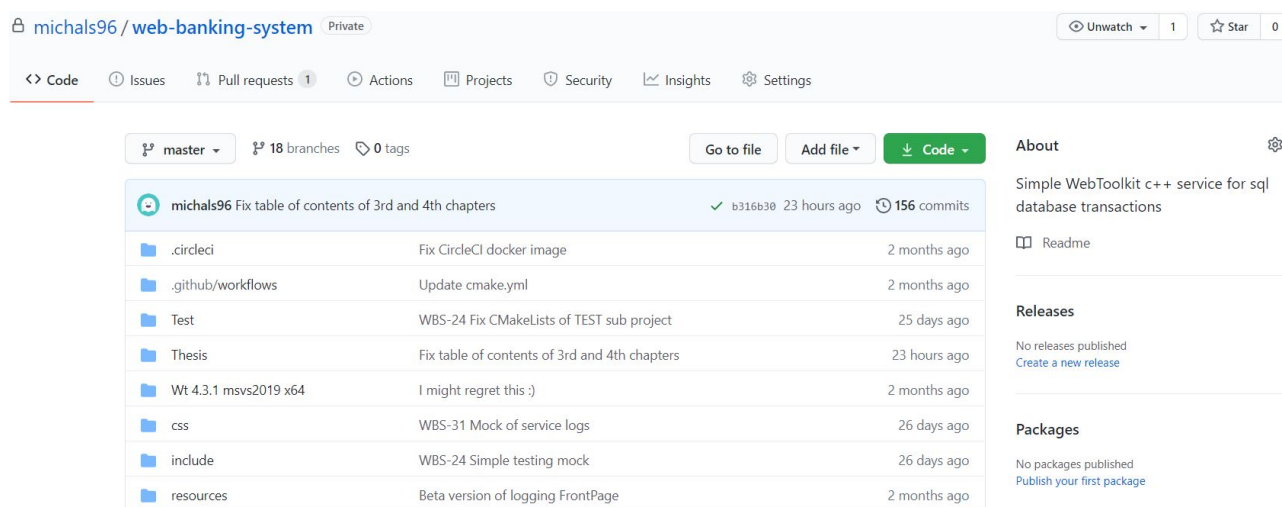


Figure 3.3: Web banking application github repository [27].

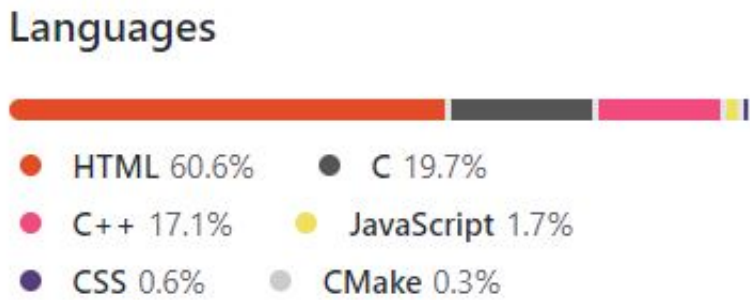


Figure 3.4: Languages statistics on Web banking application repository. The majority of is HTML code, because of the static resources of WebToolkit library, which handle HTML [27].

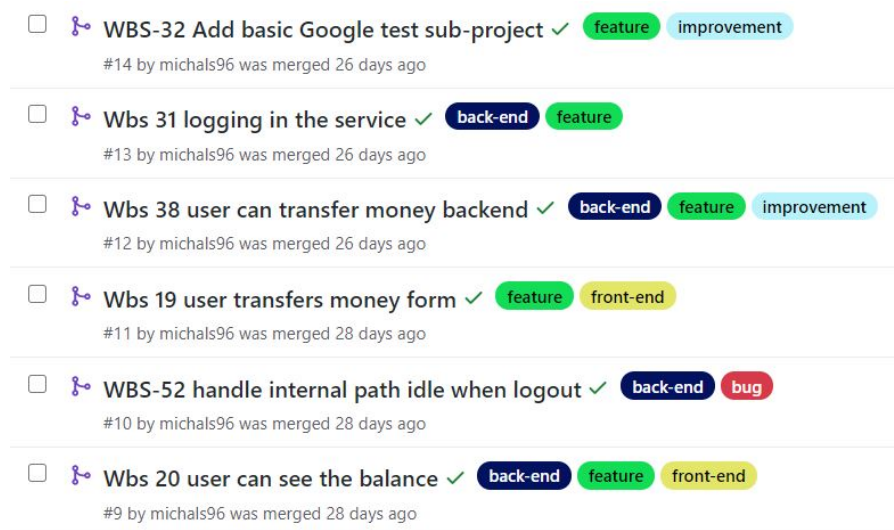


Figure 3.5: A small batch of a few pull requests done during the implementation [27].

A pull request itself is a github feature which allows programmers to do a review of a code that has been developed during a certain amount of time and is now to be ready to be merged to the main branch of the repository, which is usually the branch that is frequently being built and released to the client. This process performs a quality gate to the code and also increases readability of the changes introduced to the product. Figure 3.5 shows a few pull requests that were put out during the development. Each one of the pull requests has its unique name, id and can be also labeled by fancy labels (like feature, bug, improvement, back-end and others) which can be modified by repository owners. Before merging, the product is being built with the upcoming changes, which is another quality gate. If the build is successful and the reviewers approved the code - a programmer is allowed to merge the code and close the pull request. Another interesting thing is that every pull request or commit may be pre-tagged with a **WBS-number**, which stands for "WebBankingApplication-number of the story" and is thoroughly explained in the next section of this chapter.

3.4 Proprietary issue tracking

Another tool that came in really handy during development is proprietary issue tracking.

It is a system which reflects changes introduced to the project and ultimately allows to track and manage the progress.

3.4.1 JIRA Software

An issue tracking system which was used in the project is JIRA Software. With its graphic user interface it can provide a projects board which reflects Agile framework chosen in a development process (presented on figure 3.6) and later on it could impact on the way of presenting issues work flow in a different way. Agile framework is a set of organization patterns and work flow patterns that are followed in order to improve to development process. In the project we could group issues by those that are meant to be done, those that are being currently developed and those that are already done.

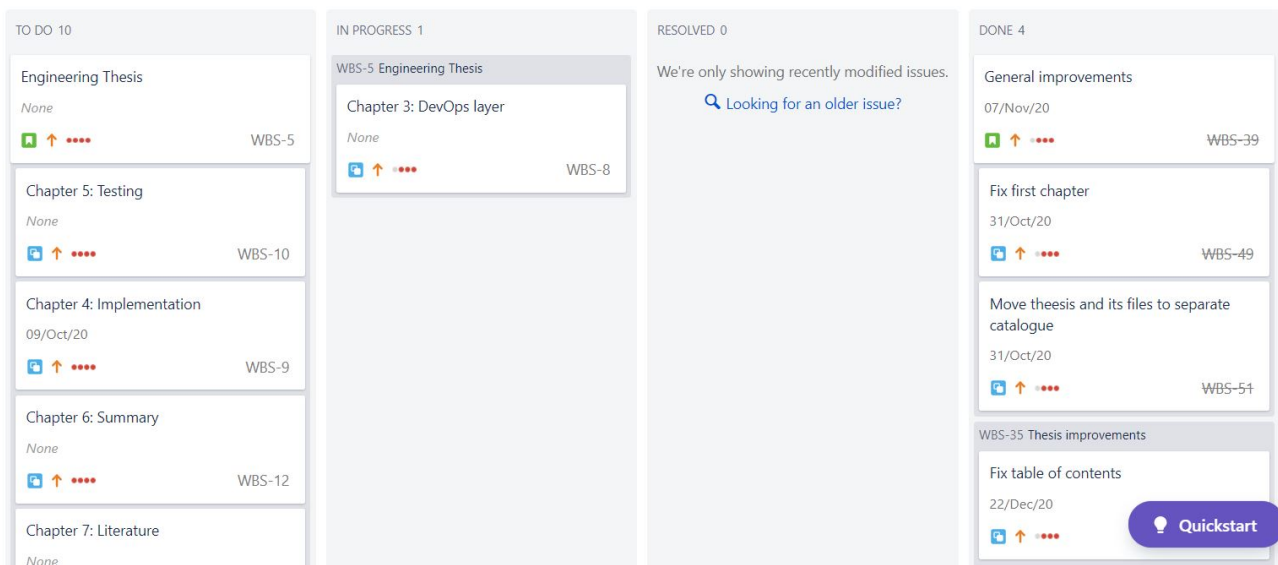


Figure 3.6: JIRA Kanban board used during projects development [28].

Kanban board is a board that reflects another work flow, where issues are reported and resolved as development goes, without working in a frequent time rate like in Scrum where developers could work for example in two week sprint methodology. Nevertheless, before starting programming Web banking application, the whole development process needed to be thought through and planned well. After planning, issues were created and added to the board. With creating every part of the project there was an issue that reflected actual state of the task that is being worked on. Example issue management in JIRA is presented on figures 3.7 and 3.8. Swift GUI of JIRA Software allows to easily do drag-and-drop, starting with issues in "To Do", ending up with issues in "Done" table.

In this project issues were group by

- engineering thesis issues (mostly chapters),
- theses improvements,
- application improvements,

- general improvements,
- and code of the application.

JIRA Software allows developers to connect project issue tracking with its repository and during this development it was no different - github repository was connectet with JIRA. What it does is it detects branches and pull requests by id in commits (mentioned in previous section) that are binded to the given story or issue. It improves managing the project work flow by allowing to track the code that was merged to accomplish particular issues. All a developer needs to do is to remember to include repository acronym in commit message as a tag. In this case it was "WBS". There is a possibility to set on a repository that commits without project tag cannot be push onto remote repository, which is guarding the quality of branching.

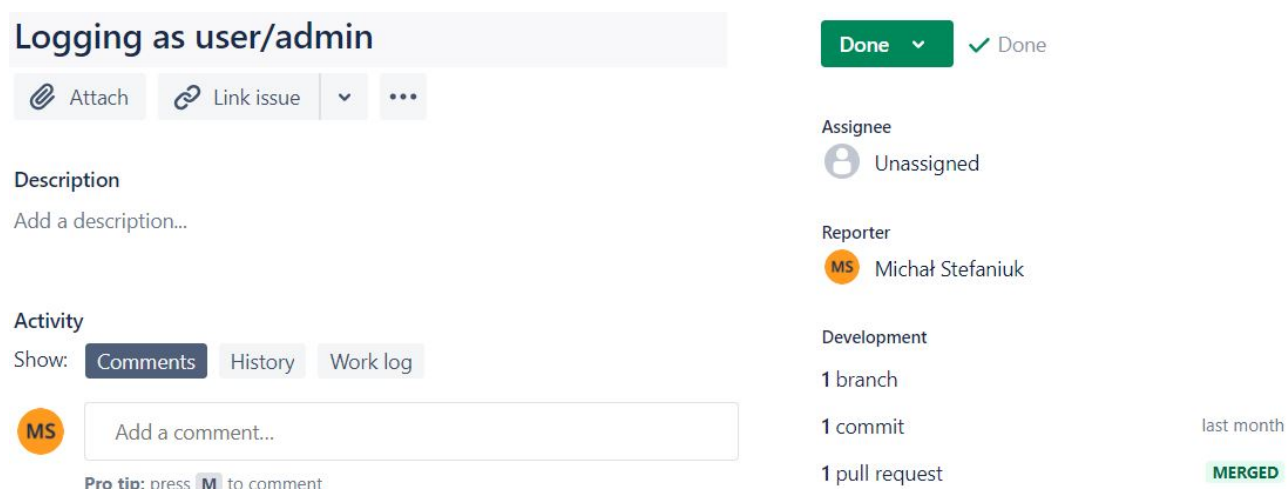


Figure 3.7: Adding project tags to commit messages on a repository that is connected with jira board detects the code created to resolve particular story ("Development" panel on the right) [28].

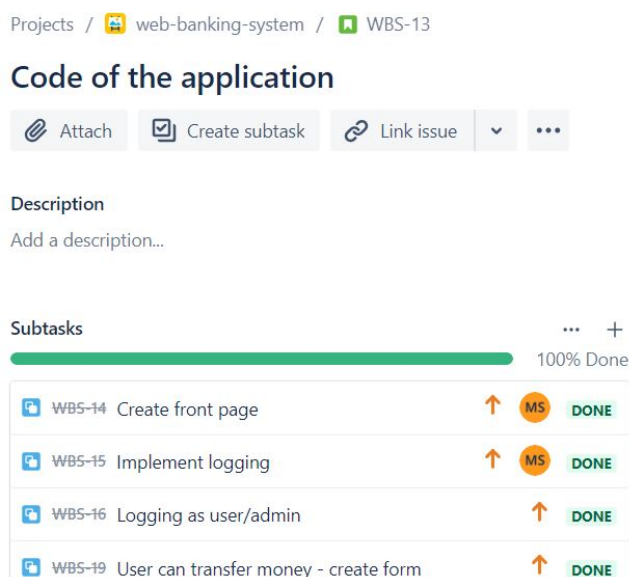


Figure 3.8: Example story with subtasks [28].

Apart from project board JIRA provides many other features like for example simple reports (figure 3.9), that are usually presented on graphs that are available by a single click. Estimating due dates and development duration is also a very important part of tracking issues. However in this project only due dates were estimated as Kanban framework does not expect estimating the duration for each issue. All in all, the project was planned for 5 stories, with around 38 subtasks. Those which were not chosen to be developed and were not completed were archived in backlog section of the board.

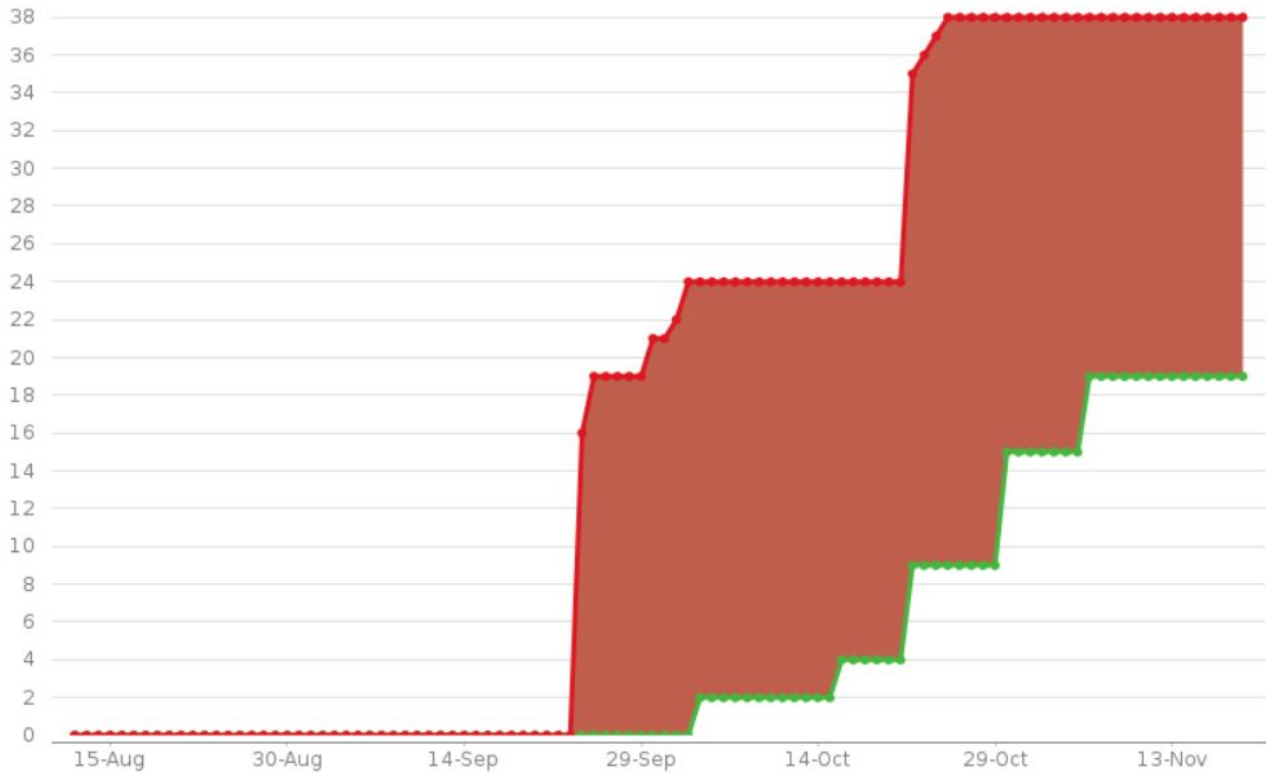


Figure 3.9: This chart shows the number of issues created vs. the number of issues resolved in the last 100 days [28].

3.5 Automation server and regression testing

Software development should not be done without assuring that the quality of the product is there. Among many quality gates like code review or unit testing there is also another one equally important - regression testing. It is being run on an automation server, which usually is another software that helps to automate the building process by configuring tasks, so called "jobs". It can also be used as a documentation generator. It is a common practice to run regression testing as well, which are nothing but a functional or non-functional test scenarios that ensure that the code that is about to be merged won't have a negative impact on the product. In a nutshell - regression testing will also verify if the software that is being developed works at all the time, considering that the regression tests might be run not only when a programmer tries to modify the code, but also when the code is merged. As a matter of fact, there are many testing approaches when it comes to regression. For instance - the tests that are causing the highest traffic on the automation server might be run over night or during the weekend when developers are not working. It is a very powerful tool which can have a major impact on the quality of the development.

During implementing web banking application, an automation server was set up, which was connected to the repository where the code was stored. This server was also set up on a GitHub, as a part of the repository. Anyone who owns a repository in this service can set such server by himself. All it is required is to go into actions section of repository settings and choosing a specific workflow that we would like to set up. Everything can be done in a few minutes and it does not require any specific knowledge. It is provided by GitHub for free with certain limitations - 20,000 minutes a month of testing is available. Anything more than this requires subscribing to the paid edition.

With each pull request, automation server took changes that were about to be merged. Then the server has been building specific branch and if the build was successful, it went to another job which was running the regression test. If every job has passed successfully, then the developer was allowed to merged the code - that is assuming that the code review was already done and all the test passed. This workflow is presented on figure 3.10

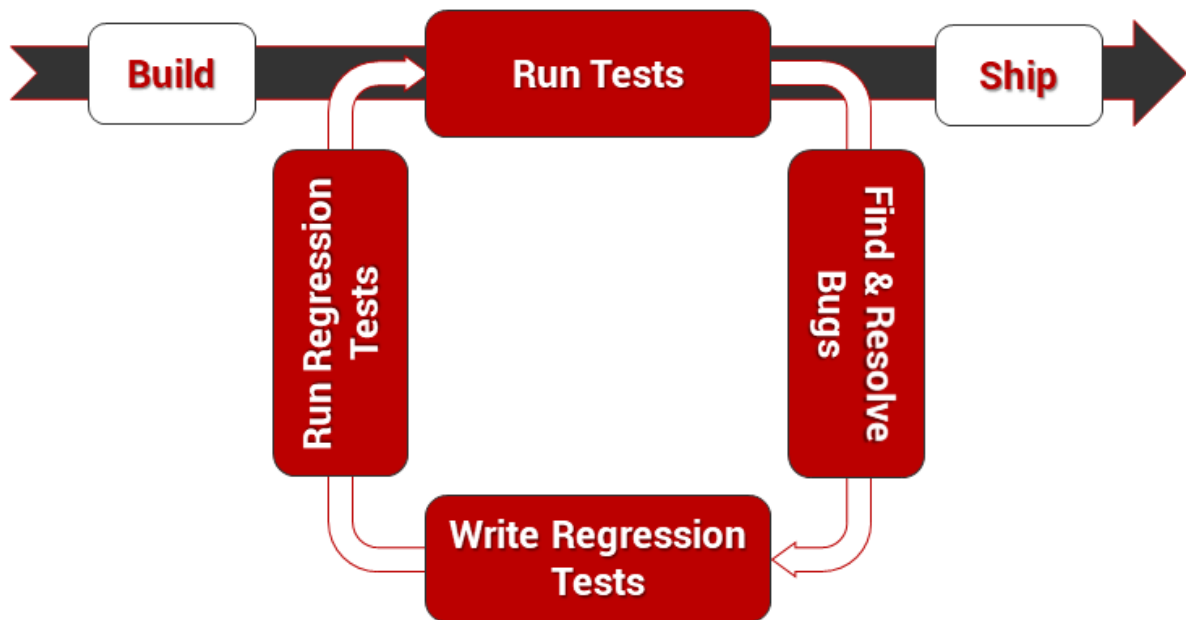


Figure 3.10: Regression testing [29].

3.5.1 GitHub CMake

The automation server that was used in a project is GitHub CMake. It is basically provided by github and it's free. All it takes is to define a CMake file that describes building dependencies for the project. Automation server will fetch it and build the application on it's remote machine with Linux's Ubuntu distribution. From this point, when the application is remotely build, regression testing might be started.

```

1  #ifndef _WIN32
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(int argc, char** argv) {
6      if (argc < 2) {
7          printf("usage: test <argument>\n");
8          printf("If <argument> is 0, print SUCCESS. Otherwise print FAIL.\n");
9          exit(1);
10     }
11
12     if (atoi(argv[1]) == 0) {
13         printf("SUCCESS\n");
14         exit(0);
15     }
16     else {
17         printf("FAIL\n");
18         exit(1);
19     }
20 }
21 #endif

```

Listing 3.1: Example ctest file used in project. It is quite self-explanatory, if first input argument equals 0, the scenario prints 'SUCCESS' and exits with 0 status. Otherwise it prints 'FAIL' and exit with 1 status.

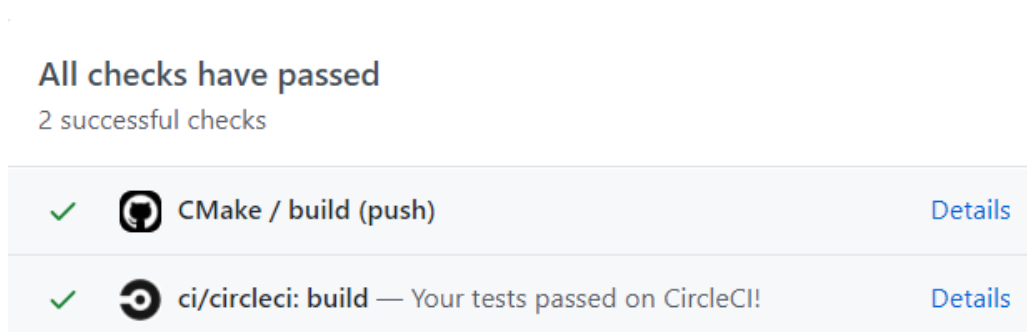


Figure 3.11: When a pull request is created, automation server triggers and verifies incoming changes by building the project and running testes.

To be able to add a test case scenario, first it must be implemented. In web banking application ctest was used, which is just a part of CMake and as an executable it enables testing, adds tests, runs them and reports results. Listing 3.1 presents example ctest used in project.

Having test implemented in place, a scenarios must be defined in CMake file (listing 3.2), that will report the results.

```

1  SET(CTEST_SRC {CTEST_SRC} ctest.cpp)
2  add_executable(test {CTEST_SRC})
3

```



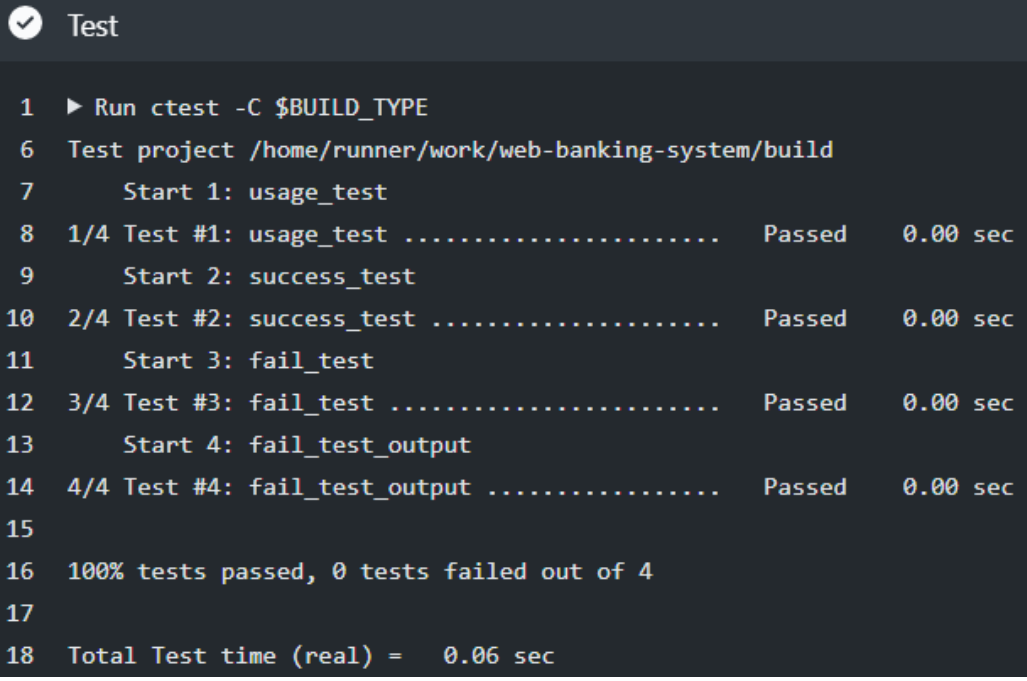
```

4 add_test(usage_test test)
5 set_tests_properties(usage_test PROPERTIES WILL_FAIL TRUE)
6
7 add_test(success_test test 0)
8 set_tests_properties(success_test PROPERTIES PASS_REGULAR_EXPRESSION "SUCCESS")
9
10 add_test(fail_test test 1)
11 set_tests_properties(fail_test PROPERTIES WILL_FAIL TRUE)
12
13 add_test(fail_test_output test 1)
14 set_tests_properties(fail_test_output PROPERTIES PASS_REGULAR_EXPRESSION "FAIL")

```

Listing 3.2: Example CMakeLists file used to define scenarios. In first two lines source code file and executable are set. Then by calling add test a scenario is added with input arguments. In the next lines, set test properties will define expected behaviour.

Including above CMakeLists file in the project will trigger running four test case scenarios with each regression run. The results will occur on an automation server and on github pull requests - figures 3.12 and 3.11.



```

1  ▶ Run ctest -C $BUILD_TYPE
6  Test project /home/runner/work/web-banking-system/build
7      Start 1: usage_test
8  1/4 Test #1: usage_test ..... Passed    0.00 sec
9      Start 2: success_test
10 2/4 Test #2: success_test ..... Passed    0.00 sec
11      Start 3: fail_test
12 3/4 Test #3: fail_test ..... Passed    0.00 sec
13      Start 4: fail_test_output
14 4/4 Test #4: fail_test_output ..... Passed    0.00 sec
15
16 100% tests passed, 0 tests failed out of 4
17
18 Total Test time (real) =  0.06 sec

```

Figure 3.12: Regression job responsible for running tests and verifying the results.

Regression Docker file

Docker is a software that realizes the concept of virtualization at OS level. By composing a docker image, a developer is able to create precise programming environment in a container, in which it possible to run any desired application. It is a lighter way of virtualization, rather than setting up a virtual machine. Figure 3.13 presents an example of container which was used in the project.

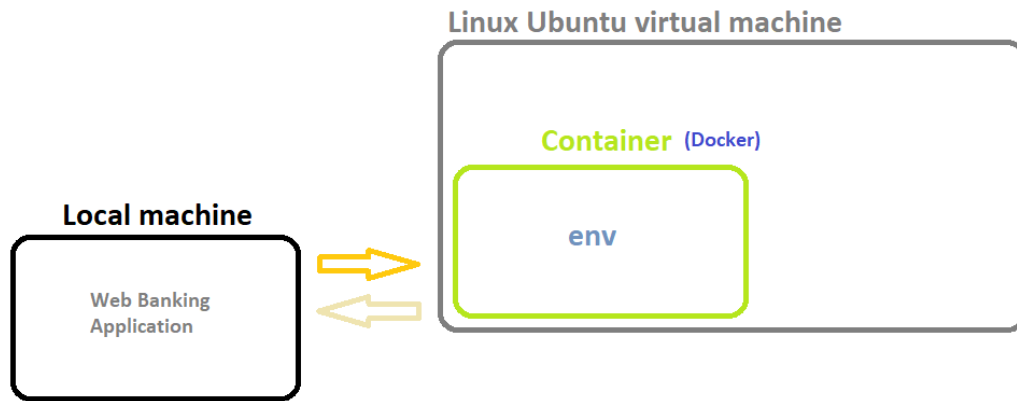


Figure 3.13: Github CMake provides Linux virtual machine for the developers, where Docker container with a specific environment is set up.

To set up the machine responsible for automation one must place docker file all necessary dependencies. The jobs that has been set up are (figure 3.14 and listing 3.3):

Set up job Loads Ubuntu 18.04.5 LTS operating system, prepares workflow directory and all required actions.

Run actions/checkout@v2 Fetches the repository, gets git version and authorizes user.

Get Boost libraries Downloads and installs Boost.

Create build environment Makes build directory for output files.

Configure CMake Identifies compilers.

Build Builds the project.

Test Runs regressions tests.

Post Run actions/checkout@v2 Post job cleanup.

Complete job Cleaning up orphan processes.

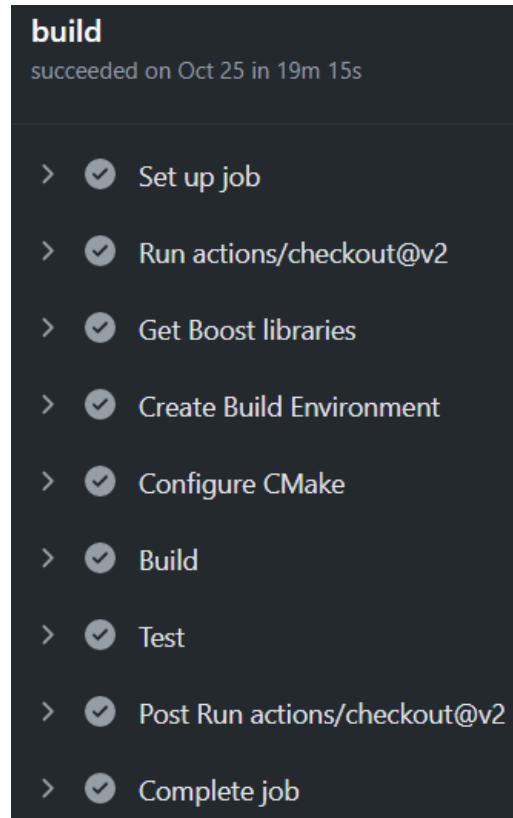


Figure 3.14: Github CMake jobs.

```
1 name: CMake
2
3 on: [push]
4
5 env:
6   BUILD_TYPE: Release
7
8 jobs:
9   build:
10     runs-on: ubuntu-latest
11
12     steps:
13     - uses: actions/checkout@v2
14
15     - name: Get Boost libraries
16       run: sudo apt-get install libboost-all-dev
17
18     - name: Create Build Environment
19       run: cmake -E make_directory ${runner.workspace}/build
20
21     - name: Configure CMake
22       shell: bash
23       working-directory: ${runner.workspace}/build
24       run: cmake $GITHUB_WORKSPACE -DCMAKE_BUILD_TYPE=$BUILD_TYPE
25
26     - name: Build
27       working-directory: ${runner.workspace}/build
28       shell: bash
29       run: cmake --build . --config $BUILD_TYPE
30
31     - name: Test
32       working-directory: ${runner.workspace}/build
33       shell: bash
34       run: ctest -C $BUILD_TYPE
```

Listing 3.3: cmake.yaml docker image used in project to set up automation server.

Chapter 4

Implementation

The Web Banking Application contains four major fields (presented at figure 4.1) which will be described in further sections. Among those, there are:

- core class,
- widgets,
- session,
- user entity,
- and database.

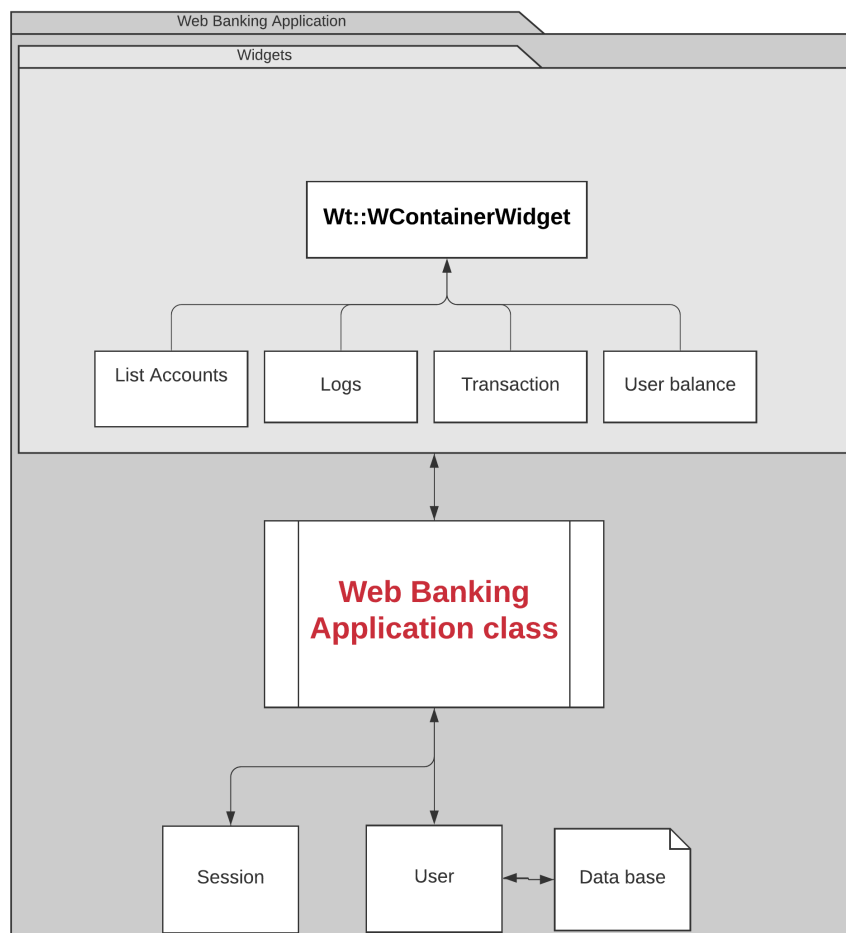


Figure 4.1: UML implementation diagram of the application.

4.1 Web Banking core class

The class represented in listing 4.1 is the main class that holds all necessary service functionalities and business logic.

```

1  class WebBankingApplication : public Wt::WContainerWidget
2  {
3  public:
4      WebBankingApplication();
5      void handleInternalPath(const std::string& internalPath);
6
7  private:
8      // Acts like a container for widgets
9      WStackedWidget      *mainStack;
10     // Widgets responsible for user and admin features
11     UserBalanceWidget    *userBalance;
12     ListAccountsWidget   *accounts;
13     TransactionsWidget    *transactions;
14     LogsWidget           *logs;
15     // Widgets that show box with links
16     // for admin and user
17     WContainerWidget      *userLinks;
18     WContainerWidget      *adminLinks;
19
20     // WAchor is a link which
21     // changes URL upon clicking
22     WAnchor* balanceAnchor;
23     WAnchor* transactionAnchor;
24     WAnchor* listUsersAnchor;
25     WAnchor* serviceLogsAnchor;
26
27     Session session;
28
29     // This function verifies if user is logged in
30     // and who is logged in. Regarding this information
31     // it renders admin menu, user menu or login panel.
32     void onAuthEvent();
33
34     // Those functions are responsible for
35     // plugging individual widgets
36     void showUserBalance();
37     void showAllAccounts();
38     void showTransactionPanel();
39     void showServiceLogs();
40 };

```

Listing 4.1: WebBankingApplication.h file. Most members are private because they are supposed only to be used in this class, apart from `handleInternalPath()` which could be used in other widgets that would want to impact the url path. The biggest disadvantage of current implementation is the class became really swell and in future it should be separated into two classes holding user and admin features individually. However the current solution is really simple, which is its biggest advantage. If a programmer would like to add a new page representing new feature, he should first create a new widget where rules of rendering content would

be described. Then he should have add an object of this widget in this class and create a function which would clear current widget container and plug the new widget. This function would be called in handle internal path and it would be triggered upon URL change, which as we now would happen by clicking new WAnchor object.

This class derives from Wt::WContainerWidget class, which is built-in WebToolkit class that behaves like a container for widgets. In the application the logic is that widgets are created and they are supposed to reflect each page and feature. Depending on which content user wants to see, the application plugs in or plugs out specific widget which is responsible for rendering particular content of the WebBankingClass container. In the constructor of this class the application creates a Session, which is a link between processes and components. Then it creates Wt::WtStackedWidget object, which will hold widgets and plugs in logging page widget into it. At the end it connects with handleInternalPath signal and start processing the whole environment.

Constatly calling handleInternalPath()(listing 4.2) function will put on a guardian, which will react to any changes in URL internal path. The logic inside of this function is whenever a path is changed, different widget will be plugged in the container, which ultimately will cause in rendering different content for the application user.

```

1 void WebBankingApplication::handleInternalPath(const std::string& internalPath)
2 {
3     // Check if user is logged in as an user
4     if (session.login().loggedIn() && session.userName() != "admin")
5     {
6         if (internalPath == "/balance")
7         {
8             showUserBalance();
9         }
10        else if (internalPath == "/transaction")
11        {
12            showTransactionPanel();
13        }
14        else
15        {
16            WApplication::instance()->setInternalPath("/", true);
17        }
18    }
19    // Check if user is logged in as an admin
20    else if (session.login().loggedIn() && session.userName() == "admin")
21    {
22        if (internalPath == "/accounts")
23        {
24            showAllAccounts();
25        }
26        else if (internalPath == "/logs")
27        {
28            showServiceLogs();
29        }
30        else
31        {

```

```

32         WApplication::instance()->setInternalPath("/", true);
33     }
34 }
35 else if (session.login().loggedIn() == false)
36 {
37     WApplication::instance()->setInternalPath("", true);
38 }
39 }

```

Listing 4.2: `handleInternalPath()` function. It reacts to URL changes. Regarding the internal path, different content is being rendered. For instance if path is changed to `localhost:8080/balance` then this function checks if user is logged in. If so, `showUserBalance()` function is being called, which clears all current widgets and plugs in user balance widget, which will cause in displaying the balance of the currently logged in user. If the user is not logged, the path will be immediately changed to `localhost:8080` and the logging page will be rendered.

Web Banking Application

Login

This is simple web application that simulates banking service.

Written and created by **MICHAL STEFANIUK** as his engineering theesis.

If you would rather not register for some reason, use the **guest/guest** or **admin/admin** account.

User name
Enter your user name

Password
Enter your password

Remember me ☐
Keeps login for 2 weeks

[Lost password](#) | [Register](#)

Figure 4.2: This is the page that renders to user when he turns the application on. Default logging widget is plugged in.

An example widget that is being rendered upon turning the application on is presented at figure 4.2. Other members of this class are just widgets and methods responsible for plugging them-in, which is the topic of the next section.

4.2 Widgets

List accounts widget

This widget renders one of admin functionalities - figure 4.3. It is responsible for listing all users in the service with the details like their balances. It could simulate a very simple functionality that could be provided to the administrator by the real banking service.

Web Banking Application

All accounts in the bank

ID	NAME	BALANCE
1	user3	80054
2	user2	52179
3	admin	15000
4	guest	2718
5	user1	549

[List all accounts](#) [Service logs](#)

Figure 4.3: This is the page that renders to user when he logs as an admin and chooses to list all accounts.

The code of this class is simple. It derives from `WContainerWidget` and has only two methods. First one is constructor, which only initializes parent class and session. Second one is `update()` function (listing 4.3) which is responsible for composing the widget with table.

```

1 void ListAccountsWidget::update()
2 {
3     clear();
4
5     this->addWidget(cpp14::make_unique<WText>
6     ("<h2> All accounts in the bank </h2>"));
7
8     const std::vector<User> &top = session->getUsers();
9
10    WTable* table = this->addWidget(cpp14::make_unique<WTable>());
11
12    table->elementAt(0, 0)->addWidget(cpp14::make_unique<WText>("ID"));
13    table->elementAt(0, 1)->addWidget(cpp14::make_unique<WText>("Name"));
14    table->elementAt(0, 2)->addWidget(cpp14::make_unique<WText>("Balance"));
15    table->setHeaderCount(1);
16
17    for (auto& user : top) {
18        int row = table->rowCount();
19
20        table->elementAt(row, 0)->addWidget
21        (cpp14::make_unique<WText>(user.id));
22
23        table->elementAt(row, 1)->addWidget
24        (cpp14::make_unique<WText>(user.name));
25
26        table->elementAt(row, 2)->addWidget

```

```

27 |         (cpp14::make_unique<WText>(asString(user.balance)));
28 |     }
29 | }

```

Listing 4.3: ListAccountsWidget::update() function. Assuming that you use a compiler with C++14 support you can use `std::make unique` instead of `Wt::cpp14::make unique`. In the application `std::make unique` was used because it allows us to have a better control of the lifetime of the widget. For instance if we want to, we can retrieve a `std::unique ptr` with `WObject::removeChild()` function, which ultimately will unplug a single widget. The disadvantage of using this smart pointer is the fact that we are dealing with a raw pointer and as always it's ownership must be changed carefully, otherwise it might cause a memory leak.

At the beginning of `update()` function in this widget, a list of users is fetched from the database by calling `getUsers()` and it is being assigned to the `top` vector. Then a very basic table is being constructed in a `for` loop regarding the information about users in `top` vector. When the iteration is finished, the widget is composed and rendered. Plugging the widget to the front page was already done in line 10, by calling `addWidget()` function on this object.

Logs widget

This widget is responsible for rendering service logs (figure 4.4) from the transactions that has been conducted among certain accounts. It is a second and the last admin functionality. The most common reason for having logs in any application is the fact that they allow to track any malicious behaviors in the system. Their purpose is to save and store information about the traffic which is happening in the programs components. In the web banking application an admin can go through service logs which contain information about every transaction - as we could imagine a real banking application admin could do.

Web Banking Application

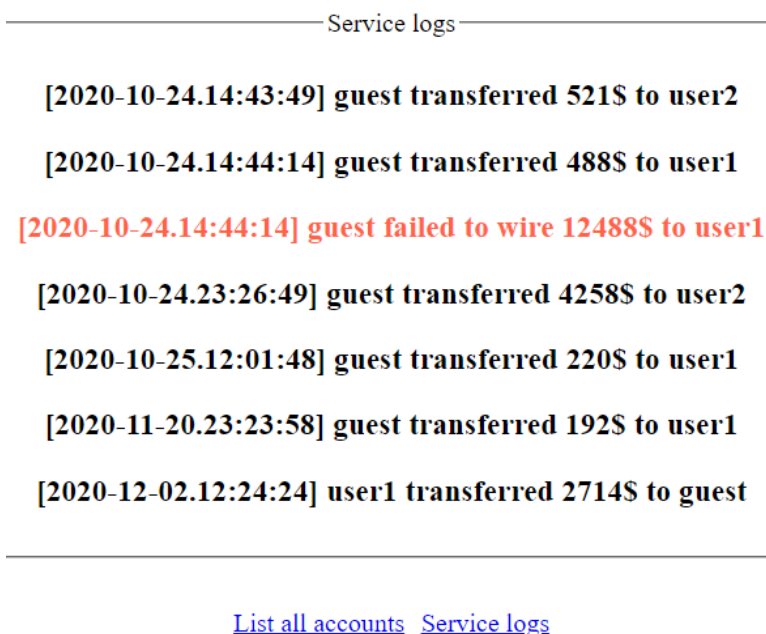


Figure 4.4: This is the page that renders to user when he logs as an admin and chooses to list all transactions.

The implementation is just the same as in every widget in this application. Derivation from `WContainerWidget`, constructor, update method for constructing the widget (listing 4.4) and a session member.

```

1 void LogsWidget::update()
2 {
3     clear();
4
5     auto groupBox = Wt::cpp14::make_unique<Wt::WGroupBox>("Service logs");
6     groupBox->addStyleClass("centered-example");
7
8     std::ifstream file("logs.txt");
9     if (file.is_open()) {
10         std::string line;
11         while (std::getline(file, line)) {
12             groupBox->addNew<Wt::WText>(line.c_str());
13         }
14
15         this->addWidget(std::move(groupBox));
16
17         file.close();
18     }
19 }

```

Listing 4.4: `LogsWidget::update()` function.

What is interesting about this widget is its construction in `update()` function. As in every widget, in the beginning all widgets from the page are removed by calling `clear()` function. Then, a group box is created, which is another type of table that is built in and provided by default Web Toolkit library. The reason why a group box was chosen was just to present another widget from the widget gallery, which is full of interesting widgets that are ready to use. Then, a file called `logs.txt` is being opened. It is a file that contains information about transactions - when a transaction is made, logs are appended to this file. If it doesn't exist, no logs are rendered. As the file is loaded, it is iterated line by line. With each iteration a line with transaction information is added to the `groupBox`, which ultimately constructs the whole widget. Then it is added to the page.

From the security point of view this is not the safest approach for storing and fetching any kind of service logs, especially as sensitive ones as the ones from the banking application, which contain personal data. Regardless the type of the application, the service logs are always of a great importance as usually they could reveal system vulnerabilities to the potential hacker. To increase security of this mechanism the logs could have been encrypted and decrypted upon loading. Better yet, they could have been stored as an object in a database, which could have been stored in a various places limiting an access to those logs, by for instance authorizing potential clients.


Transaction widget

This widget is the first one that is provided by the service to the user without admin permissions. It is a panel, that allows user to actually conduct a simple transaction which is wiring money - figure 4.5.

Web Banking Application

Transfer money menu

How much money do you want to transfer?



1646\$

user1 ▾ You selected user1.

100 % Resume Stop Reset

Transfer complete!

[See the balance](#) [Make a transaction](#)

Figure 4.5: This is the page that renders to user without admin permissions when chooses to make a transaction.

The wiring menu is rather simple. At the very top there is a slider where user can point out how much money he wants to wire. It is a nice show-how of the Web Toolkit library. By using built in slider widget it is possible to create an animation on the page without writing single javascript line of code. Then a user can choose to who he wants to wire the money. The last box is where a user can start a transaction and while it's being processed, he can stop it or reset the whole menu. When a transfer is completed, a notification appears.

Although TransactionsWidget implementation is rather classic as it is for every widget in this application, there are a few additions to this particular widget. There is member `log()` function that is being called upon every transaction and it appends to `logs.txt` file information about the transfer that has just been made. The update function has a lot of boilerplate code which is responsible for the built in widgets (like the slider), so instead focusing on the whole `update()` function let's just take a look about the most important part which is the lambda responsible for handling the wiring itself - listing 4.5.

```

1 intervalTimer->timeout().connect([=] {
2     bar->setValue(bar->value() + 1);
3     if (bar->value() == 10) {
4         session->addToBalance(transactionUsers[cb->currentIndex()].name,
5         session->userName(), slider->value());
6
7         log(transactionUsers[cb->currentIndex()].name,
8         session->userName(), slider->value(), true);
9
10        stopButton->clicked().emit(Wt::WMouseEvent());
11        startButton->disable();
12
13        this->addWidget(cpp14::make_unique<WBreak>());

```

```

14         this->addWidget( std::move( cpp14::make_unique<WText>(" Transfer
15         complete!") ) ) );
16     }
17 });

```

Listing 4.5: Part of TransactionWidget::update() function.

In the beginning of this lambda function, we connect a timeout(), which is a signal emitted after the timer times out, to the intervalTimer which generates those signals when a start() function is being called. So basically whenever user presses "Start" button to wire money, this lambda is being called. First we increment the value of the slider by one just to get the animation going. It's just a visual effect, but it also gives user a chance to stop the transaction. When the bar slider has reached an end, addToBalance() function is called, which finds source and target user and updates their balances by adding or subtracting money. Then a log() function is called - at this point we know what it's responsible for. The rest of the logic is just disabling start button, clicking stop button and printing "Transfer complete!" notification.

User balance widget

This is the second and the last widget provided by the service to the user without admin permissions. It is a very basic one, which prints the information about current user's balance and it is presented in figure 4.6.

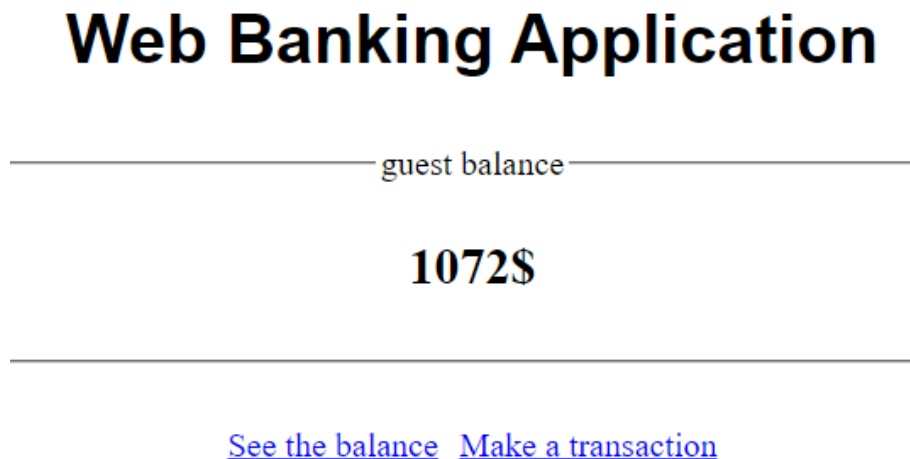


Figure 4.6: This is the page that renders to user without admin permissions when he chooses to see the balance.

The update() function for this widget is very transparent (listing 4.6). What it does is it just fetches users from the database and the user that is just logged. Then it compares logged user to the one in a database and if it finds the user, it fetches his balance. Lastly, the balance is added to the simple groupBox, which is later added to the widget container and that is how user balance widget is being composed.

```

1 void UserBalanceWidget::update()
2 {
3     clear();

```

```

4
5     int userBalance = 0;
6
7     std::string userName = session_ -> userName();
8     std::vector<User> top = session_ -> topUsers();
9
10    for (auto& user : top) {
11        if (user.name == userName) {
12            userBalance = user.balance;
13        }
14    }
15
16    auto groupBox = Wt::cpp14::make_unique<Wt::WGroupBox>(userName + " balance");
17    groupBox->addStyleClass("centered-example");
18
19    std::string userBalanceStr = std::to_string(userBalance);
20    groupBox->addNew<Wt::WText>("<h2>" + userBalanceStr + "</h2>");
21    this->addWidget(std::move(groupBox));
22 }

```

Listing 4.6: UserBalanceWidget::update() function.

4.3 Session

Knowing that HTTP is a stateless protocol we have to point out one very important thing. WWW server will process each transaction independently, without looking for any relations to other transactions committed by the user. This means that it is almost impossible for an application to track all systems of authorization that require to track every user activity within the application. For example - if we log in the service at the logging page, how does the service know at the next page that we are still logged in? The answer is it doesn't. To be more precise - it doesn't until it's provided with lasting entity carrying information, that could be passed between components. The answer to this challenge is session. It is a class that is equipped with business logic that allows it to carry valuable information from the service point of view and may be passed between components (implementation in listing 4.7). It can be simulated when we use dynamic language like C++. In many languages it is a common practice to implement session, like for instance when coding a server-side of the application in PHP.

```

1 typedef Auth::Dbo::UserDatabase<AuthInfo> UserDatabase;
2
3 class Session
4 {
5 public:
6     static void configureAuth();
7
8     Session();
9     ~Session();
10
11     Auth::AbstractUserDatabase& users();
12     Wt::Auth::Login& login();
13     std::vector<User> topUsers(int limit);
14     std::string userName() const;

```

```

15     int findId();
16     void addToBalance(std::string tgtName, std::string srcName, int amount);
17
18     static const Auth::AuthService& auth();
19     static const Auth::AbstractPasswordService& passwordAuth();
20     static const std::vector<const Auth::OAuthService*>& oAuth();
21
22 private:
23     mutable Dbo::Session session_;
24     Wt::Auth::Login login_;
25     std::unique_ptr<UserDatabase> users_;
26     Dbo::ptr<User> user() const;
27 };

```

Listing 4.7: Session.h file.

Most of the methods are just getters to the private fields which are mostly WebToolkit built-in type variables. They are already provided with many valuable methods like for instance querying the database objects directly. The most important part of session logic are topUsers() and addToBalance() methods and those are the ones that we will focus on.

When it comes to topUsers() method (listing 4.8), it is used in every method that tries to modify the state of the database or in method that wants to fetch information about the users in the service.

```

1  std::vector<User> Session::topUsers(int limit)
2  {
3      dbo::Transaction transaction(session_);
4
5      Users top = session_.find<User>().orderBy("balance desc").limit(limit);
6
7      std::vector<User> result;
8
9      for (Users::const_iterator i = top.begin(); i != top.end(); ++i)
10     {
11         dbo::ptr<User> user = *i;
12         result.push_back(*user);
13
14         result.back().name = user->name;
15     }
16
17     transaction.commit();
18
19     return result;
20 }

```

Listing 4.8: Session::topUsers() method.

In the very beginning a transaction object is created using current session. Then all users are being assigned to top variable. This is the place in code where idea behind session concept clarifies. To fetch all users from database, find() method is being called on current session with chain of orderBy and limit methods. This will trigger a native query on a sqlite application database, which is actually visible in an application admin console. Second part of this method

just iterates through User object pushing back each user to the vector which is being returned as a result. Of course, before returning this vector, a transaction must be committed - just like in figure 4.7.

```

Dbo.backend.Sqlite3: begin transaction
Dbo.backend.Sqlite3: select "id", "version", "name", "balance" from "user" order by balance desc limit ?
Dbo.backend.Sqlite3: commit transaction

```

Figure 4.7: This is a really classic model for a database transaction. First a transaction is being made, then it's being created and when it's done, it is committed.

The second equally important method is `addToBalance()` (listing 4.9). As mentioned before, it is quite self-explanatory. Its job is to find two users, the one that wants to wire some money and the one that the money will be transferred to. Next thing that this function does, is it updates balances of both subjects of this transaction. The whole logic is completed by committing such transaction, which obviously needs to be opened at the very beginning.

```

1 void Session::addToBalance(std::string tgtName, std::string srcName, int amount)
2 {
3     dbo::Transaction transaction(session_);
4
5     dbo::ptr<User> targetUser =
6         session_.find<User>().where("name = ?").bind(tgtName);
7
8     targetUser.modify()->balance += amount;
9
10    dbo::ptr<User> srcUser =
11        session_.find<User>().where("name = ?").bind(srcName);
12
13    srcUser.modify()->balance -= amount;
14
15    transaction.commit();
16 }

```

Listing 4.9: `Session::addToBalance()` method.

4.4 User entity

User class (listing 4.10) is probably the most important class in Web Banking Application. It is a class that defines a structure of an user object that is being mapped to a database object. Later on, all operations rely on this content like for instance logging to the service or making transactions which is a core of the banking application.

```

1 class User;
2 typedef Wt::Auth::Dbo::AuthInfo<User> AuthInfo;
3 typedef dbo::collection< dbo::ptr<User> > Users;
4
5 class User
6 {

```



```

7 public:
8     User();
9     std::string name;
10    int balance;
11    dbo::collection<dbo::ptr<AuthInfo>> authInfos;
12
13    template<class Action>
14    void persist(Action& a)
15    {
16        dbo::field(a, name, "name");
17        dbo::field(a, balance, "balance");
18        dbo::hasMany(a, authInfos, dbo::ManyToOne, "user");
19    }
20 };
21
22 DBO_EXTERN_TEMPLATES(User);

```

Listing 4.10: User class.

What is really interesting about this class is the way Web Toolkit transforms this class into the database object. To achieve this, it uses ORM operation. It stands for Object-Relational Mapping and it is a way of mapping an objective architecture of a single system to the database that has relational character. Such implementation is a common approach for applications that are implemented in an objective language using objective pattern. In this class there is only one `persist()` method defined. In this member function we define how Web Toolkit should map class member variables to database fields, so whenever an User object is created in code, Web Toolkit already knows how to put it into the database, if necessary. Other than that there are no methods whatsoever and the constructor is default. The class itself is rather really simple, there are three fields. Name, balance and authInfos for authentication upon logging. It is a built-in functionality provided by Web Toolkit. Let's jump into an example code (listing 4.11) which will get us closer to the idea behind using this class with object relational mapping.

```

1     auto sqlite3 = cpp14::make_unique<Dbo::backend::Sqlite3>
2     (WApplication::instance()->appRoot() + "WebBankingUserDatabase.db");
3
4     sqlite3->setProperty("show-queries", "true");
5
6     session_.setConnection(std::move(sqlite3));
7     session_.mapClass<User>("user");
8     session_.mapClass<AuthInfo>("auth_info");
9     session_.mapClass<AuthInfo::AuthIdentityType>("auth_identity");
10    session_.mapClass<AuthInfo::AuthTokenType>("auth_token");
11
12    dbo::Transaction transaction(session_);
13    try {
14        session_.createTables();
15
16        // Creating and inserting Auth Users
17        Auth::User guestUser = users_->registerNew();
18        guestUser.addIdentity(Auth::Identity::LoginName, "guest");
19        myPasswordService.updatePassword(guestUser, "guest");
20
21        /*

```

```

22     Creating other auth users
23     ...
24     */
25
26     // Creating reflection of Auth Users used in transactions
27     std::unique_ptr<User> userGuest{ new User() };
28     userGuest->name = "guest";
29     userGuest->balance = 10000;
30     dbo::ptr<User> userPtr = session_.add(std::move(userGuest));
31
32     /*
33     Creating other reflection users
34     ...
35     */
36
37     log("info") << "Database created";
38 }
39 catch (...) {
40     log("info") << "Using existing database";
41 }
42
43 transaction.commit();

```

Listing 4.11: Example usage of object-relational mapping on User class. This logic happens in constructor of Session class, which is called upon turning the application on.

At the very beginning sqlite database object is created. Then all necessary properties are set, including connecting current session with this database and setting mapping properties to user and authentication model. Then, as usually, a transaction is created and later on in a try block it is executed. At the beginning of try block createTables() function is being called on session object. If WebBankingUserDatabase.db already exists it will throw an exception which is then caught. This will cause in using already existing database. Otherwise try block will go further where new users are created and added to the database. Each user has to have two identities, first one for authentication purposes with login and password. Second one for business logic, which is a "reflection" of authentication user.

4.5 Database

The database that was chosen for this application was sqlite3 data base. It is implemented in C language and it supports basic SQL language queries. The reason behind choosing sqlite is because it is natively supported by Web Toolkit and it's very light and simple. In the data base used in project we have three tables.

Authentication identity

This table (figure 4.8) contains one of two reflections of users in the service. Those are actually the ones used for authentication during logging into the service. The most valuable information that this table carries is identity of the user auth info id which later on creates one-to-one relation with Authentication information table.

Table: **auth_identity**

	id	version	auth_info_id	provider	identity
	Filter	Filter	Filter	Filter	Filter
1	1	0	1	loginname	guest
2	2	0	2	loginname	admin
3	3	0	3	loginname	user1
4	4	0	4	loginname	user2
5	5	0	5	loginname	user3

Figure 4.8: Authentication identity table.

Authentication information

This table (figure 4.9) in relation with Authentication identity creates a whole piece of record that is used to authenticate any user. It stores valuable and fragile information like password. Because of this fact, the password is encrypted with bcrypt method. In the database there is a password salt and password hash column - with those information it is possible to decrypt each password, but for the sake of security such information should not be stored in a plain text. There is also a table named "failed login attempts" which will increment with every failed login attempt an user makes.





Table: **auth_info**

	id	version	user_id	password_hash	password_method	password_salt	status	failed_login_attempts	last_login_attempt
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	29	NULL	\$2y\$07\$WjKwMDbSYknOTILORzOxc...	bcrypt	bs28WTjjPVsPOT3y	1	0	2020-12-06T15:42:42.653
2	2	44	6	\$2y\$07\$WTjYTCsbk7uOUDmLVHxJe...	bcrypt	aYgiQ.vopAah5rs.	1	0	2020-12-05T20:38:11.299
3	3	3	NULL	\$2y\$07\$TBasJlb0LivwRijkORjuaujyio...	bcrypt	T7..wv6LrNIfA9ps	1	0	2020-12-02T11:24:04.011
4	4	2	NULL	\$2y\$07\$cRDTWjLHXjS3RjHFSjSzQec...	bcrypt	y1UbSifU9NRGRU5J	1	0	2020-11-15T01:03:26.971
5	5	1	NULL	\$2y\$07\$Kj/BY1TDaUq1RxjpZxf/...	bcrypt	2PCkuEqk7O9ko8AD	1	0	2020-10-24T11:21:35.515

Figure 4.9: Authentication information table.

User

User table is the second one containing information about the users in the table, but this one is actually the one that is object-relational mapped from the code to the database object. Records from this table are also the ones used for transactions and other Web Banking Application features different then logging and authentication. It stores information about users name, balance and version, which actually shows an amount of changes that has been made to each record. For example if a guest's version is 17, it means that it was involved in 17 transactions (figure 4.10).

Table:  user   

	id	version	name	balance
	Filter	Filter	Filter	Filter
1	1	17	guest	1072
2	2	0	admin	15000
3	3	7	user1	2195
4	4	11	user2	52179
5	5	1	user3	80054

Figure 4.10: User table.

Chapter 5

Summary

The topic of this thesis was to create a web application that would simulate a banking service, which would allow to use software engineering tools during development. The finished application is a program that can be run by anyone and it realizes all the initial assumptions that were made at the beginning. During the implementation there were three tools established to enhance the development - GitHub repository, GitHub regression server and JIRA issue tracking tool. The GitHub version control system was extremely helpful especially when something needed to be debugged and fixed. Tracking changes in the project may have a huge impact on the development. JIRA software also helped a lot in tracking the actual state of the process of not only implementing the application, but also in writing this thesis. The regression testing came out to be more like a proof of concept, rather than an important quality gate. Mostly it's because regression testing makes a lot of sense when the application is rather big, developed frequently and in a long term perspective. The code should also be merged almost on a daily basis by a team of developers. This application instead was developed by one person for this thesis only and it's rather small, so setting up whole regression with proper building scripts was really time consuming. As a matter fact it might have been more time consuming than what it would really take to fix a potential bugs that could came up without the regression. For a project of this size the most efficient way to assure the quality of the code would be unit testing and a proper code review.

The biggest obstacle that was overcome during this development was lack of knowledge about Web Toolkit coding standards. This library introduces specific type of logic that may occur a bit unorthodox to many C++ developers. Another thing which came out to be troubling is rather small community behind Web Toolkit. Programming is many things, but one is for sure - there is no point in solving problems that were already solved (at least other than educational). While implementing this application almost every problem that came up had to be solved individually with no support from the programming forums whatsoever. This was actually really time consuming.

Had I have a chance to implement the application all over again, I would probably focus a little bit more on the front-end layer, which definitely could have been improved. Instead I wouldn't set up the regression testing. Sure, it was a nice addition that showed the way of doing this, but for such a small project it was unnecessary. Also, my most important conclusion is I wouldn't choose C++ for developing web application again. It is very interesting and educational process, but other than that it is way more complicated than it should be. What makes it unnecessarily difficult are for instance pointers, whereas javascript does not require to think about possible memory leaks. It usually takes a lot of lines of code to come up with a really simple functionality that could have been just a few lines in javascript. The bottom

line is, one could have a lot of fun while using C++ for web development just for fun, but in a professional project this could be a nightmare.

Conclusions

The final application is quite flexible. From this point it could have been enhanced with many user features like for instance taking loans, banning users, setting transaction limits, creating subaccounts or even investing money. All it would take is making more widgets corresponding to each functionality. A front end layer could have been improved too with many fancy sliders, forms or widgets. When it comes to user interface - sky is the limit.

It is a fact - one can actually implement a web application using C++ and although it is possible, it is not the most efficient way to go for when choosing technology stack. The standard technologies used for web development are javascript and html with css and it's because they are natively supported and easily interpreted by most web browsers. During the development it came clear that C++ library like Web Toolkit can only act as a wrapper for such technologies and it causes a lot of tricky riddles when it comes to debugging such code. Nevertheless, as a developer gets familiar with Web Toolkit concepts, the development rapidly speeds up and indeed, it could be used for creating simple single page web applications.

Bibliography

- [1] Introduction to CMake, <http://derekmolloy.ie/hello-world-introductions-to-cmake>.
- [2] Single Page Applications, <https://www.excellentwebworld.com/what-is-a-single-page-application>.
- [3] WT Logo and Features, <https://www.webtoolkit.eu/wt/features>.
- [4] Introduction to Wt::Auth, <https://www.webtoolkit.eu/wt/doc/tutorial/auth.html>.
- [5] Wt Hello World Application, <https://www.webtoolkit.eu/wt/doc/tutorial/wt.html>.
- [6] Wt Widgets Gallery, <https://www.webtoolkit.eu/widgets/forms/>.
- [7] Facebook, <https://www.facebook.com>.
- [8] airbnb, <https://www.airbnb.com>.
- [9] Twitter, <https://www.twitter.com>.
- [10] PayPal, <https://www.paypal.com>.
- [11] gmail, <https://www.gmail.com>.
- [12] Netflix, <https://www.netflix.com>.
- [13] JavaScript, <https://www.javascript.com/>.
- [14] TypeScript, <https://www.typescriptlang.org/>.
- [15] HTML, <https://www.w3schools.com/html/>.
- [16] CSS, <https://www.w3schools.com/css/>.
- [17] Java, <https://www.java.com/pl/>.
- [18] Python, <https://www.python.org/>.
- [19] PHP, <https://www.php.net/>.
- [20] HTML5, <https://developer.mozilla.org/pl/docs/HTML/HTML5>.
- [21] HTML4, <https://www.w3.org/TR/html5-diff/>.
- [22] C++11 lambda expressions, <https://en.cppreference.com/w/cpp/language/lambda>.
- [23] OpenGL, <https://www.opengl.org/>.
- [24] HTML5 SVG, <https://css-tricks.com/using-svg/>.

- [25] devOps logo, <https://www.netsparker.com/devops-security-tools/>.
- [26] Git workflow, <https://panizkomputerem.pl/czym-jest-git/>.
- [27] Applications github repository, <https://github.com/michals96/web-banking-system>.
- [28] JIRA board, <https://www.atlassian.com/pl/software/jira>.
- [29] Regression testing, <https://www.maveryx.com/automated-regression/?cn-reloaded=1>.

Additional materials

- LaTeX wikibooks
<https://en.wikibooks.org/wiki/LaTeX>