Mrežno programiranje u Qt 5 biblioteci

Omanović, Aida

Master's thesis / Diplomski rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:501700

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-02-05**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Aida Omanović

MREŽNO PROGRAMIRANJE U QT 5 BIBLIOTECI

Diplomski rad

Voditelj rada:
prof. dr. sc. Mladen Jurak

Zagreb, lipanj, 2022.

Ovaj diplomski rad obranjen je dana ____________________ pred ispitnim povjerenstvom u sastavu:

1. ____________________________, predsjednik
2. ____________________________, član
3. ____________________________, član

Povjerenstvo je rad ocijenilo ocjenom ________________.

Potpisi članova povjerenstva:

1. ____________________________
2. ____________________________
3. ____________________________

Posveta obitelji i prijateljima.

Sadržaj

Sadržaj **iv**

Uvod **1**

1 Qt **3**
1.1 Povijest Qt-a . . . 3
1.2 Grafičko korisničko sučelje . . . 3
1.3 Qt Creator . . . 4
1.4 Alat za izgradnju *qmake* . . . 5
1.5 Meta objektni sustav . . . 5
1.6 Signali i utori . . . 7
1.7 Oblikovni obrasci . . . 9

2 Qt WebEngine **13**
2.1 Uvod . . . 13
2.2 Chromium . . . 13
2.3 Zamjena modula Qt WebKit . . . 14
2.4 Arhitektura . . . 16
2.5 Funkcionalnosti Qt WebEngine modula . . . 18
2.6 Aplikacija i njene funkcionalnosti . . . 20
2.7 Implementacija . . . 23

Bibliografija **37**

Uvod

Qt je softver za izradu grafičkih korisničkih sučelja ili aplikacija koje se pokreću na različitim platformama. Neke od platformi su Windows, Linux, macOS za stolna i prijenosna računala dok su za mobilne uređaje podržane Android, iOS, Universal Windows Platform i ostale. Uključuje široki raspon biblioteka i alata za razvoj softverskih aplikacija. Pisan je u programskom jeziku C++ i za razvoj koda se koristi C++. Korištenjem odgovarajućih modula, kod je moguće razvijati i u nekim drugim jezicima poput Pythona.

Tema ovog diplomskog rada je proučiti mrežno programiranje u Qt 5 biblioteci te demonstrirati naučeno kreiranjem web aplikacije koja koristi Qt WebEngine modul.

U prvom poglavlju opisujem Qt softver. Posebno, Qt Creator okolinu u kojem razvijamo Qt aplikacije i koja nam nudi grafičko sučelje za vizualno kreiranje aplikacija. Naglasak stavljam i na meta objektni sustav koji nadopunjuje Qt i ograničenja C++-a kao programskog jezika. U sljedećem potpoglavlju se opisuje jedan od glavnih mehanizama Qt-a, mehanizam signala i utora, koji je odgovoran za komunikaciju između objekata. Također, taj mehanizam prepoznajem kao primjer oblikovnog obrasca *Observer*. Osim njega, u zadnjem potpoglavlju sam se dotkanula i *Composite* oblikovnog obrasca koji se primjenjuje kod `QObject` klase, klase iz koje su izvedeni svi Qt Widget-i.

U drugom poglavlju se fokusiram na Qt WebEngine modul. Ovaj modul omogućuje prikazivanje internetskih stranica, lokalnog HTML sadržaja, prikaz PDF dokumenata, pristup *Chromium Devtools* i ostalo. Opisujem glavne dijelove tog modula i strukturu svakog od njih. Kroz Qt Widget aplikaciju *Fancy Browser* implementirala sam funkcionalnosti WebEngine modula. Izgled aplikacije i detalje implementacije objasnila sam u zadnjem potpoglavlju ovog rada.

Poglavlje 1

Qt

1.1 Povijest Qt-a

Razvoj Qt-a započeo je s dva norveška softverska inženjera, *Haavard Nordom* i *Eirik Chambe-Engom* 1991. godine, tri godine prije nego što je tvrtka osnovana kao *Quasar Technologies*, a zatim su promijenili ime u *Troll Tech*, a potom u *Trolltech*. U lipnju 2008. godine *Nokia* je kupila *Trolltech* te promijenila ime prvo u *Qt Software*, zatim u *Qt Development Frameworks*. Nakon *Nokie*, vlasništvo preuzima *Digia* 2014. godine.

Prve dvije verzije Qt-a imale su Qt/X11 za Unix i Qt/Windows za Windows dok je u svibnju 1995. godine prvi put postao javno dostupan. Veliku promjenu je označila verzija Qt 5 objavljena 2012. godine. Uvela je nove module, a posebno ubrzala i olakšala razvoj korisničkih sučelja. Također, s tom verzijom se pojavljuju QML i JavaScript. Trenutno najnovija verzija ovog softvera je 6.3.0 objavljena u travnju ove godine. Qt je dostupan u dvije verzije: komercijalnoj i verzija otvorenog koda koja se ne naplaćuje.

Danas je *Qt Company* u vlasništvu *Qt Group* koja posluje u Kini, Finskoj, Njemačkoj, Japanu, Koreji, Norveškoj, Rusiji, SAD-u, Francuskoj, UK-u i Indiji, a broji 550 zaposlenih diljem svijeta. Sjedište *Qt Group* je u Espoou u Finskoj.

1.2 Grafičko korisničko sučelje

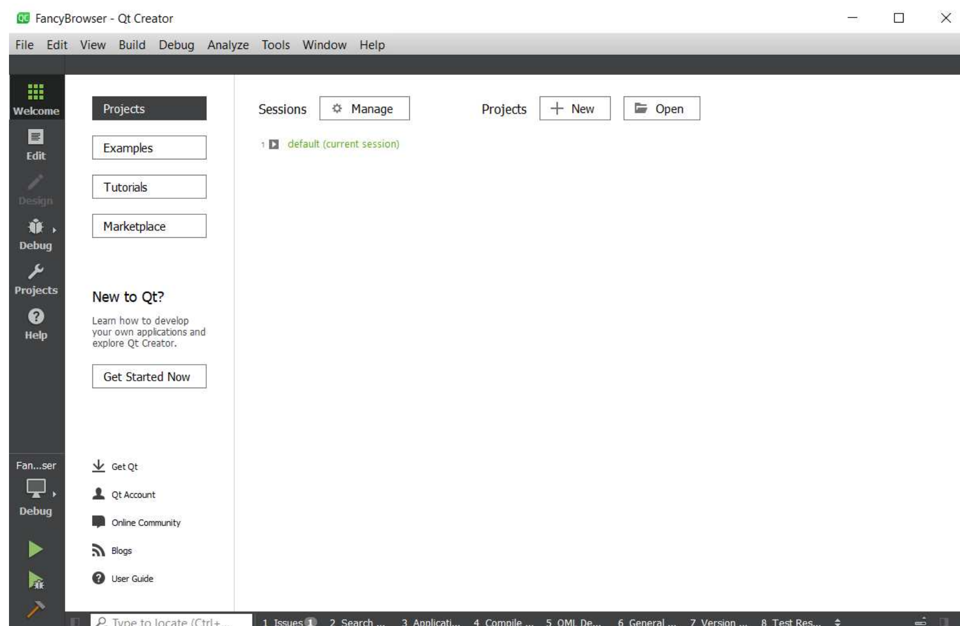
Grafičko korisničko sučelje (eng.*graphical user interface*, GUI) je sučelje kroz koje korisnik komunicira s elektroničkim uređajem odnosno računalom koristeći ikone, izbornike, gumbe i ostale grafičke elemente.

Grafička korisnička sučelja u Qt-u možemo programirati na dva načina. Prvi je kreiranjem Qt Widgets aplikacije, a drugi kreiranjem Qt Quick aplikacije. U Qt Widgets aplikaciji koristimo standardno imperativno programiranje dok aplikacija bazirana na Qt Quick-u implementira sučelje na deklarativan način. Qt Quick aplikacija je specifična po

tome što se sučelje opisuje u posebnom jeziku koji se naziva QML (eng. *Qt Modeling Language*), a logika u C++-u ili Pythonu. Za aplikacije namijenjene stolnim i prijenosnim računalima pomoću Qt Widgets-a možete brzo izgraditi jednostavno korisničko sučelje dok se Qt Quick preporučuje za mobilne aplikacije ili za softvere kompatibilne za više sustava (desktop i mobilni operativni sustavi).

1.3 Qt Creator

Qt Creator je integrirano razvojno okruženje (eng. *integrated development environment*, IDE) za kreiranje Qt aplikacija. Dostupan je za operacijske sustave Linux, macOS, FreeBSD i Windows za stolna i prijenosna računala dok je za mobilne uređaje podržan na sustavima Android, iOS, Blackbery, Maemo i MeeGo. Qt Creator podržava `.pro` datoteke, *cmake* i *Autotools*. Ima podršku za C++, QML i ECMAscript, alate za navigaciju kodom, podršku za refaktoriranje koda, *debugger* opcije, podršku za sustave verzioniranja programskog koda (npr. Git, Subversion, Perforce) i mnoge druge. Posebno, sadrži integrirano grafičko sučelje Qt Designer i Qt Quick Designer. Qt Designer omogućava korisnicima da vizualno dizajniraju i izgrade svoje aplikacije pomoću Qt widget-a dok sučelje Qt Quick Designer omogućuje izgradnju aplikacija pomoću deklarativnog programskog jezika QML.



Slika 1.1: Qt Creator

1.4 Alat za izgradnju *qmake*

U svakom C++ projektu je važno pojednostaviti izgradnju projekta odnosno da u jednom koraku možemo izvršiti naredbu kojom ćemo dobiti željeni izlaz. Također, kad god se neka datoteka projekta promijeni, ne želimo ponovno prolaziti kroz probleme izgradnje cijelog projekta već nam je cilj obnoviti samo promijenjene datoteke. Upravo su ovo karakteristike kojima se bavi alat *qmake* i *make*.

Naime, *qmake* je alat za izgradnju sustava isporučen s Qt bibliotekama koji pojednostavljuje postupak izgradnje na različitim platformama. Za razliku od *CMake*, *qmake* je dio Qt-a od samih početaka. Iz tog razloga Qt Creator ima najbolju podršku za *qmake*. On dolazi instaliran i konfiguriran već prilikom instalacije Qt-a. *Qmake* automatizira generiranje odgovarajućih *Makefile* datoteka na temelju informacija u `.pro` datoteci koju kreira programer. Projektna datoteka `.pro` se obično sastoji od popisa izvornih datoteka i datoteka zaglavlja, od općenitih informacija o konfiguraciji te popisa dodatnih biblioteka potrebnih za aplikaciju. Pomoću naredbi unutar jedne projektne datoteke, možemo prilagoditi izgradnju aplikacije za više različitih platformi. Operatore, funkcije i ostale naredbe koje možemo koristiti u projektnoj datoteci možete vidjeti na [4]. Alat *qmake* možete koristiti za bilo koji softver, pisan u Qt-u ili ne.

Makefile datoteke su obične tekstualne datoteke koje sadrže informacije o kompajliranju i povezivanju datoteki projekta. Kod velikih projekata, *Makefile* datoteke nam pomažu da projekt predstavimo na sustavan i učinkovit način. Isto tako, omogućuje kompajliranje samo datoteka koje su promijenjene. Stoga ne trebamo generirati cijeli projekt kada su samo neki dijelovi projekta izmijenjeni.

Make je alat koji se koristi za izgradnju izvršnih programa iz izvornog koda. *Make* zahtijeva datoteku *Makefile* jer iz nje čita instrukcije koje treba izvršiti.

1.5 Meta objektni sustav

Qt je pisan u programskom jeziku C++ koji je između ostaloga i statički jezik. On kao takav, nema mogućnost refleksije. Naime, refleksija je proces tijekom kojeg računalni program može pratiti i mijenjati vlastitu strukturu i ponašanje (konkretno vrijednosti, metapodatke, funkcije) tijekom njegovog izvršavanja. Kod C++-a ovo se odvija statički, odnosno prilikom kompilacije samog programa. Mehanizmi jezika su takvi da se informacije o tipovima, manipulacije i dedukcije istih dešavaju prilikom prevođenja programa. Zbog navedenih nedostataka jezika C++, postoji meta objektni sustav. Ovaj sustav je odgovoran za mehanizam signala i utora, informacije o tipu objekata tijekom izvršavanja programa (eng. RTTI, *run time type information*) i Qt *property* sustav.

U ovom sustavu, jedna `QMetaObject` instanca se kreira za svaku `QObject` potklasu koja se koristi u aplikaciji. Ova instanca sadrži sve metapodatke za tu potklasu. Meta

objektni sistem se temelji na sljedeće tri stvari:

1. `QObject` klasa je bazna klasa svih objekata u Qt-u i time pruža mogućnost klasama da iskoriste prednosti meta objektnog sustava.

2. `Q_OBJECT` makro omogućuje funkcionalnosti ovog sustava. Koristi se unutar privatnog dijela definicije klase te je obavezan za sve klase koje koriste mehanizam signala i utora.

3. *Meta-Object Compailer* (*moc*) čita izvornu C++ datoteku te pronalazi klase u kojima se koristi makronaredba `Q_OBJECT`. Na temelju toga za svaku od tih klasa proizvodi drugu izvornu C++ datoteku koja sadrži meta objektni kod. Ova generirana izvorna datoteka je ili uključena u izvornu datoteku klase ili, što je češće, kompajlirana i povezana s implementacijom klase.

Osim što omogućuje mehanizam signala i utora za komunikaciju između objekata, što je glavni razlog uvođenja meta objektnog sustava, ovaj sistem nudi i dodatne funkcionalnosti poput:

- Metoda `QObject::metaObject()` vraća pokazivač na metaobjekt ovog objekta koji sadrži informacije o imenu klase, metodama, svojstvima, konstruktorima i mnoge druge.

- Metoda `QMetaObject::className()` vraća ime klase kao string tijekom vremena izvršavanja programa.

- Metoda `QObject::inherits(const char *className)` vraća `true` ako je objekt instanca klase koja nasljeđuje klasu imena *className* ili ako je objekt instanca neke potklase od `QObject`-a i nasljeđuje klasu imena *className*, inače vraća `false`.

- `QObject::tr()` prevodi stringove za internacionalizaciju. (Internacionalizacija je proces kojim se softver dizajnira na način da se može prilagoditi različitim jezicima ili regionalnim razlikama.)

- `QObject::setProperty()` i `QObject::property()` dinamički postavljaju i dohvaćaju svojstva po imenu.

- `QMetaObject::newInstance()` kreira novu instancu ove klase.

Također je moguće izvršiti dinamičko pretvaranje koristeći `qobject_cast()` na `QObject` klasama. Funkcija `qobject_cast()` ponaša se identično kao standardna C++ funkcija `dynamic_cast()`, s prednostima da ne zahtijeva podršku za RTTI.

1.6 Signali i utori

Kroz grafičko korisničko sučelje često želimo da prilikom promjene jednog objekta, automatski se promijeni ili ažurira drugi objekt. U nekim programima se ova funkcionalnost postiže pomoću funkcija povratnog poziva (eng. *callback function*) dok se u Qt-u koriste signali i utori (eng. *signals and slots*). Ovaj mehanizam je jedan od najbitnijih karakteristika Qt-a. Zahvaljujući meta objektnom sustavu, mehanizam signala i utora možemo koristiti za komunikaciju između objekata unutar aplikacije.

Kada se dogodi neki određeni događaj, emitira se signal. Qt widget-i imaju već unaprijed predefinirane signale, ali bez obzira na to mi možemo dodati i svoje signale. Signali i utori moraju primati isti broj argumenata, ali mogu uzimati bilo koji broj argumenata proizvoljnog tipa. Bitno je samo da su tipovi argumenata kompatibilni. Povezivanje signala i utora vrši se metodom `QObject::connect()`.

Klasa koja emitira signal ne brine o utorima niti zna koji utori primaju taj signal. Utori se mogu koristiti za primanje signala, ali to su ujedno i obične funkcije članice. Baš kao što objekt ne zna prima li neki utor njegove signale, tako niti utor ne zna ima li spojene signale. Ovo osigurava da se pomoću Qt-a mogu kreirati uistinu neovisne komponente. Na jedan utor se može spojiti proizvoljan broj signala, a signal se također može spojiti na bilo koji broj utora. Čak se može spojiti jedan signal na drugi signal. U tom slučaju, kada god se emitira prvi signal, pokrenut će se emitiranje drugog signala. Sve klase koje nasljeđuju klasu `QObject` ili neku potklasu te klase, mogu sadržavati signale i utore.

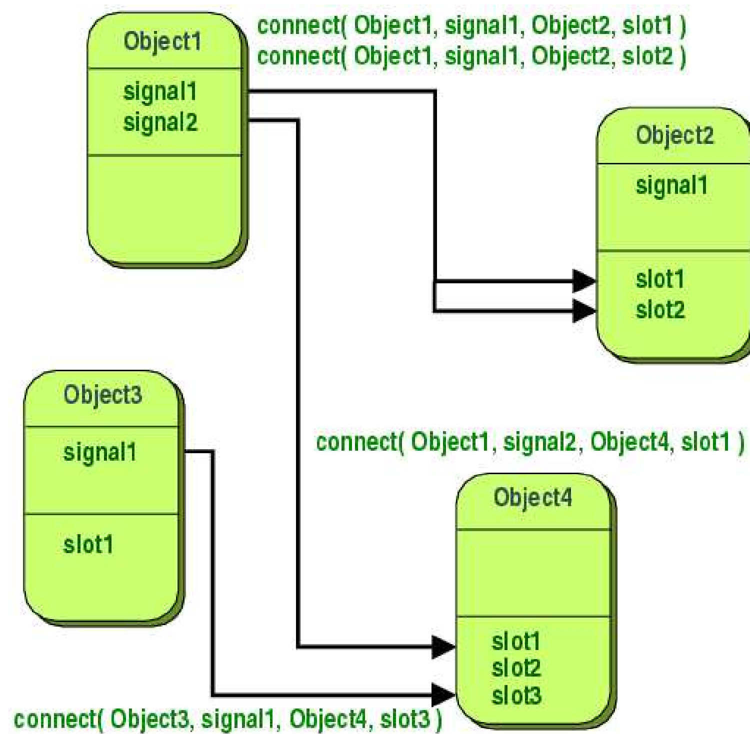
Pogledajmo sliku 1.2. Ovdje vidimo da je metodom `connect` spojen signal *signal1* objekta *Object1* na utore *slot1* i *slot2* nekog drugog objekta *Object2* odnosno isti signal je povezan s dva utora. Također, spojen je i signal *signal2* objekta *Object1* na utor *slot1* objekta *Object4* iz čega vidimo da objekt može imati više signala koji su spojeni na utore različitih objekata.

Signali

Signale emitira objekt kada se njegovo stanje promijeni na neki način. Signali su zapravo funkcije javnog pristupa i mogu se emitirati s bilo kojeg mjesta, ali preporučuje se da se emitiraju iz klase koja definira sam signal ili iz njegove potklase. Kada se signal emitira, utori koji su povezani s njim se izvršavaju odmah, baš kao i normalan poziv funkcije. Nakon što se pozovu i izvrše sve funkcije koje su povezane s tim signalom, nastavlja se izvršavanje koda nakon naredbe `emit` koja je emitirala taj signal. Ako je nekoliko utora spojeno na jedan signal, utori će se izvršavati jedan za drugim, redoslijedom kojim su spojeni, kada se signal emitira. Signali se automatski generiraju pomoću *moc*-a i ne smiju biti implementirani u `.cpp` datoteci.

Utori

Utor se poziva kada se emitira signal povezan s njim. Utori su obične C++ funkcije i mogu se normalno pozivati. Ono što ih izdvaja od drugih funkcija je to da se na njih mogu spojiti signali. Budući da su utori normalne funkcije članice, oni slijede normalna pravila C++ kada se pozivaju izravno. Međutim, kao utori, njih može pozvati bilo koja klasa, bez obzira na njenu razinu pristupa, putem veze signal-utor. To znači da signal emitiran iz instance proizvoljne klase može uzrokovati pozivanje privatnog utora u instanci neke druge klase. U usporedbi s funkcijama povratnog poziva, signali i utori su nešto sporiji zbog povećane fleksibilnosti koju pružaju, iako je razlika neprimjetna za stvarne aplikacije.



Slika 1.2: Signali i utori

1.7 Oblikovni obrasci

Oblikovni obrasci (eng. *design patterns*) naziv su za rješenja problema koji se često javljaju u konstrukciji programskog koda. Mogu ubrzati proces razvoja programske potpore (eng. *software*) kroz korištenje ispitanih i u razvoju dokazanih razvojnih paradigmi. Svaki obrazac predstavlja rješenje za jedan oblikovni problem. Podijeljeni su u tri glavne grupe:

- **Obrasci stvaranja** (eng. *Creational Patterns*)

  Obrasci stvaranja primjenjuju se prilikom kreiranja instanci kada je uobičajen način stvaranja objekata problematičan ili otežava dizajn.

  Obrasci: *Abstract Factory*, *Builder*, *Factory Method*, *Object Pool*, *Prototype*, *Singleton*

- **Strukturni obrasci** (eng. *Structural Patterns*)

  Strukturni obrasci modeliraju međusobne odnose među objektima i načine slaganja objekata kako bi se dobila nova funkcionalnost.

  Obrasci: *Adapter*, *Bridge, Composite*, *Decorator*, *Facade*, *Flyweight*, *Proxy*

- **Obrasci ponašanja** (eng. *Behavioral Patterns*)

  Obrascima ponašanja definira se način komunikacije i sinkronizacije među objektima unutar sustava.

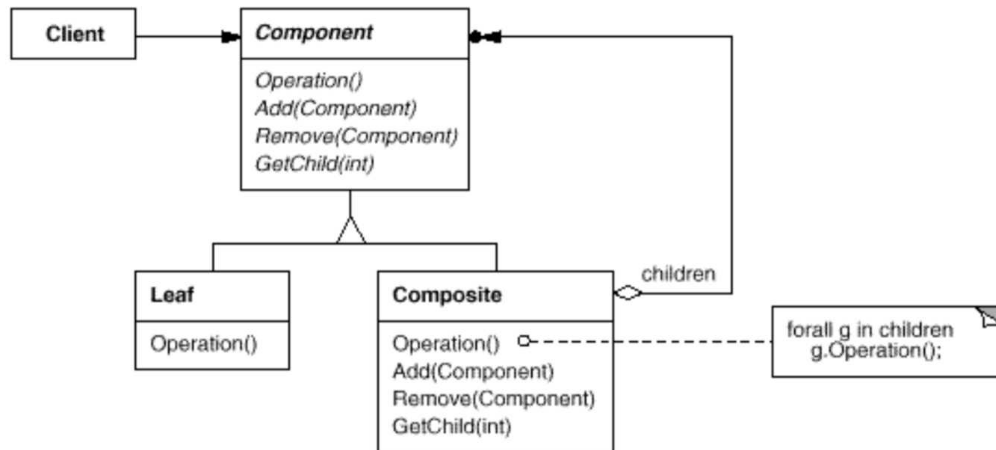
  Obrasci: *Chain of responsibility*, *Command*, *Interpreter*, *Iterator*, *Mediator*, *Memento*, *Observer*, *State*, *Strategy*, *Template method*, *Visitor*.

Istaknut ćemo dva oblikovna obrasca koji se primjenjuju u Qt-u.

Composite

Problem koji promatramo zahtijeva strukturiranje složenog objekta kao stablo koje predstavlja hijerarhiju sastavnih dijelova. Cilj je da klijent može jednako tretirati i dio i cjelinu. Rješenje se bazira na tome da ista apstraktna klasa (*Component*) predstavlja i složeni (*Composite*) i elementarni objekt (*Leaf*).

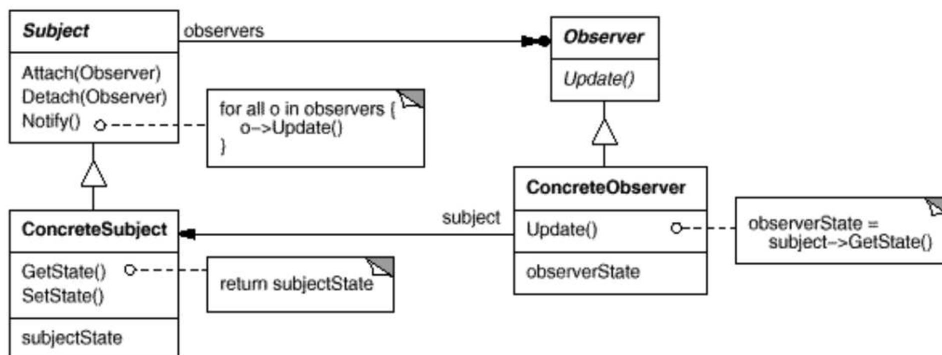
Klasa iz koje su izvedeni svi Qt Widget-i je `QObject`. Svaki objekt klase izvedene iz `QObject` može imati najviše jednog roditelja i proizvoljan broj djece. Odnos cjeline i dijela cjeline iz *Composite* obrasca, u Qt-u promatramo kao odnos roditelja i djece. `QObject` objekti koji imaju djecu smatramo *Composite* komponentama (drže referencu na djecu i implementiraju operacije vezane za djecu) dok kao *Leaf* komponente smatramo objekte bez djece.



Slika 1.3: Struktura *Composite* obrasca

Observer

Različiti objekti mogu ovisiti o trenutnom stanju nekog objekta i stoga moraju biti obaviješteni o promjeni stanja tog objekta u trenutku kad se ona dogodi. U ovom obrascu svaki Promatrač se registrira kod Subjekta i drži referencu na njega. Subjekt dozvoljava da se kod njega registrira proizvoljno velik broj promatrača. Subjekt signalizira svakom Promatraču kada je došlo do promjene njegovog stanja. Promatrač ima javnu metodu `update()` koju Subjekt može pozvati kada želi signalizirati promjenu Promatraču.



Slika 1.4: Struktura *Observer* obrasca

Mehanizam signala i utora koji smo objasnili u poglavlju 1.6 se može promatrati kao generalizacija oblikovnog obrasca *Observer*. Objekt koji emitira signal je Subjekt dok je

Promatrač zapravo utor. Za razliku od *Observer* obrasca, ovdje Subjekt ne mora znati ništa o promatračima i Promatrač ne drži referencu na Subjekt.

Poglavlje 2

Qt WebEngine

2.1 Uvod

Od verzije Qt 5.6, Qt WebKit modul je zamijenjen modulom Qt WebEngine. Ovaj modul nam omogućava da ugradimo internetski preglednik (eng. *web browser*) u našu Qt aplikaciju te da pomoću njega prikažemo web-stranice ili lokalni HTML sadržaj. Također, podržava i funkcionalnosti poput pristupa *Chromium DevTools*, HTML5 *drag and drop* načina, prikaz internetskog sadržaja na cijelom zaslonu (eng. *fullscreen mode*), pregled PDF dokumenata i mnoge druge. Pomoću raznih C++ klasa i QML tipova omogućuje renderiranje HTML, XHML, SVG dokumenata stiliziranih CSS-om (*Cascading Style Sheets*). Qt WebEngine modul se bazira na Chromiumu odnosno koristi kod iz Chromium projekta. Nije utemeljen na svim funkcionalnostima Chromiuma, stvari poput binarnih datoteki, pomoćnih usluga koje komuniciraju s Googleovim platformama su izostavljene.

2.2 Chromium

Google je razvio Chromium, projekt otvorenog koda (eng. *open source*) na kojem se temelji Google Chrome preglednik. S obzirom da je otvoren, svi mogu pristupiti izvornom kodu Chromiuma. Njegova svrha je pružiti izvorni kod Google Chrome preglednika čiji kod nije dostupan za kopiranje i korištenje. Kao *web* preglednik je manje stabilan od Chromea, međutim nudi vrlo sličan korisnički doživljaj kao preglednik Chrome. Chrome se temelji na Chromiumu, ali ima i dodatne značajke koje Chromium ne posjeduje. Neke od njih su: Adobe Flash Player, PDF preglednik, automatsko ažuriranje. Uz Chrome, postoji još preglednika koji se temelje na Chromiumu i dodaju nove fukcionalnosti, a neki od njih su: Opera, Yande, Vivaldi, i drugi. Postoji i vizualna razlika u logotpiovima, Chrome-ov logotip je šarene boje dok je Chromiumov plave boje. [14]

Slika 2.1: Chromium i Chrome

2.3 Zamjena modula Qt WebKit

Qt WebKit je Qt modul utemeljen na WebKitu gdje je WebKit softver odgovoran za prikazivanje internetskih stranica (eng. *web browser engine*) dizajniran od strane Applea. Modul Qt WebKit se smatra zastarijelim (engl. *deprecated*) od verzije Qt5.5. Naime, to označava da će u budućnosti biti uklonjen iz standardnih Qt modula. Kao alternativa, uveden je modul Qt WebEngine baziran na Chromiumu. Neki od razloga zašto je uveden Qt WebEngine su sljedeći:

- Chromium je usmjeren na više platformi, a preglednik je dostupan na svim glavnim desktop platformama i na Androidu dok za WebKit to ne vrijedi.
- Korištenje Chromiuma pojednostavljuje upravljanje integracijom OS-a (eng. *operating system*).
- Chromium se razvija uz strogu kontrolu kvalitete što omogućuje lakše testiranje i pružanje stabilnijeg i kvalitetnijeg WebEngine modula.
- Za razliku od WebKita, Chromium pruža bolju i učinkovitiju integraciju s widgetima i Qt Quick grafom scene (eng. *scene graph*).

Neke od promjena koje su došle s novim Qt WebEngine modulom su:

- Klase Qt WebEngine modula koje su ekvivalentne Qt WebKit klasama, modificirane su s prefiksom „*QWebEngine*“ umjesto prethodnog „*QWeb*“ (npr. `QWebHistory` klasa je preimenovana u `QWebEngineHistory`).
- U *qmake* projektnoj datoteci sada umjesto

```
QT += webkitwidgets
```

koristimo

```
QT += webenginewidgets
```

- Qt WebKit modul smo u izvornim datotekama uključivali pomoću naredbi:

```
#include <QtWebKit/QtWebKit>
#include <QtWebKitWidgets/QtWebKitWidgets>
```

  dok je sada za uključenje Qt WebEngine modula potrebno dodati:

```
#include <QtWebEngineWidgets/QtWebEngineWidgets>
```

- `QWebFrame` je sada dio `QWebEnginePage` klase.

  Naime, HTML okviri se mogu koristiti za raspodjelu *web* stranica u nekoliko prozora gdje se sadržaj može pojedinačno prikazati. U svaki takav okvir se može učitati HTML dokument.

  U Qt WebKit modulu, `QWebFrame` predstavlja okvir unutar *web* stranice. Svaki objekt `QWebPage` sadrži barem jedan okvir, glavni okvir, dobiven pomoću metode `QWebPage::mainFrame()`. Dodatni okviri će se stvoriti za HTML `<frame>` element, koji definira izgled i sadržaj jednog okvira, ili za `<iframe>` element, pomoću kojeg možemo umetnuti okvir unutar bloka teksta.

  U Qt WebEngine modulu, manipulacija okvirima je dio klase `QWebEnginePage`. Svi podređeni okviri sada se smatraju dijelom sadržaja i dostupni su samo putem *JavaScript* koda. Metode klase `QWebFrame`, kao što je *load()*, sada su dostupne izravno preko `QWebEnginePage` objekta.

- Qt WebEngine nema interakciju s `QNetworkAccessManager` klasom za razliku od Qt WebKita.

  Signali i metode `QNetworkAccessManager` klase koje su još uvijek podržane premještene su u klasu `QWebEnginePage`.

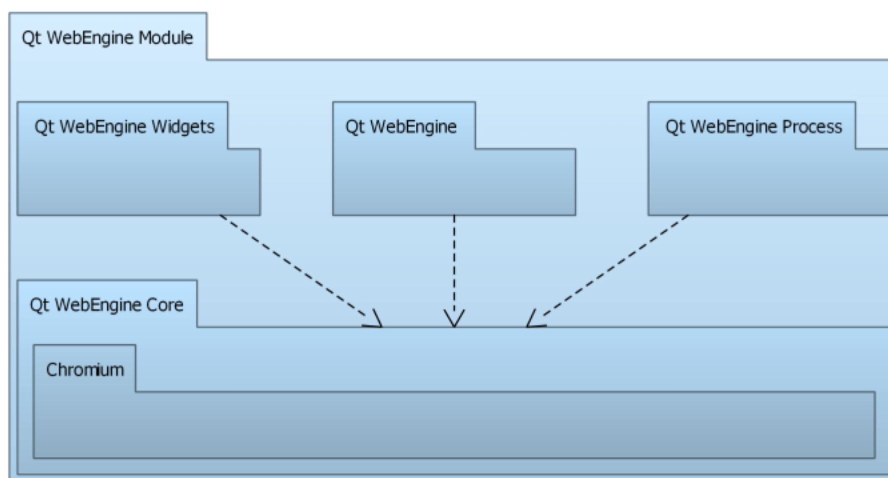
Ostale promjene te Qt WebKit klase i metode koje više neće biti dostupne unutar Qt WebEngine modula možete pogledati na [3].

2.4 Arhitektura

Funkcionalnosti Qt WebEngine modula su raspodijeljene u dijelove:

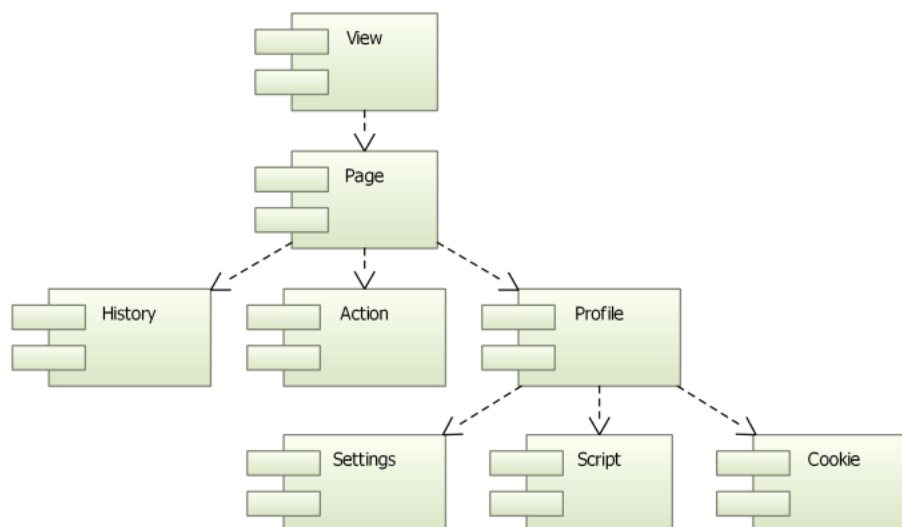
- **Qt WebEngine Widgets modul** za kreiranje *web* aplikacija utemeljenih na *widget*-ima
- **Qt WebEngine modul** za kreiranje *web* aplikacija na temelju Qt Quick-a
- **Qt WebEngine Core modul** za interakciju sa *Chromium*-om.
- **Qt WebEngine Process modul** za renderiranje web stranica i izvršavanje JavaScript-a.

Navedene dijelove možemo jasnije vidjeti na slici 2.2. U nastavku ćemo pojasniti svaki modul malo detaljnije.



Slika 2.2: Struktura Qt WebEngine modula

Qt WebEngine Widgets modul



Slika 2.3: Qt WebEngine Widgets modul

U Qt WebEngine Widgets modulu glavna *widget* komponenta je *web engine view* koja se koristi za učitavanje *web* sadržaja.

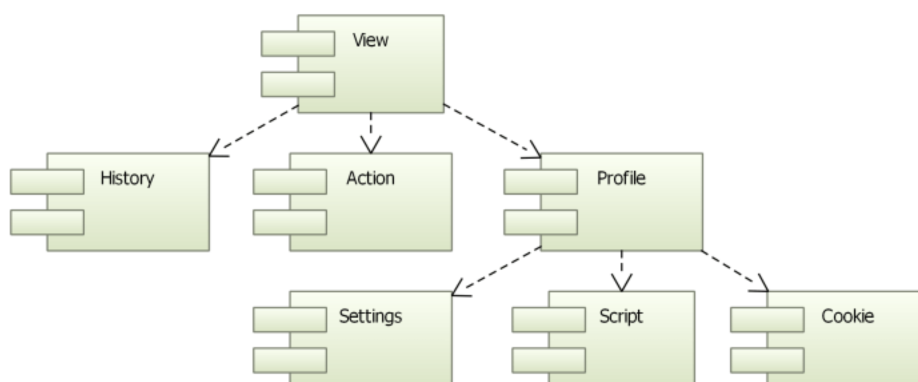
Unutar *view*-a imamo komponentu *web engine page* koja pak sadrži glavni okvir odgovoran za *web* sadržaj, povijest pregledavanja poveznica i radnji. Komponente *view* i *page* su poprilično slične s obzirom da pružaju zajedničke funkcije.

Na slici 2.3 možemo primjetiti kako sve stranice (*pages*) pripadaju *web engine profile*-u koji sadrži dijeljene postavke (eng. *settings*), skripte (eng. *scripts*) i kolačiće (eng. *cookies*). Profili se mogu koristiti za izolaciju stranica međusobno. Jedan primjer toga je privatni način pretraživanja. Naime, ovdje se otvara privremena sesija koja je izolirana od glavne sesije i korisničkih podataka.

Qt WebEngine Widgets modul koristi *Qt Quick scene graph* za sastavljanje elemenata *web* stranice u jedan prikaz. To znači da korisničko sučelje (eng. *UI, user interface*) zahtijeva OpenGL ES 2.0 ili OpenGL 2.0 za svoje renderiranje.

Qt Web Engine modul

Ovaj modul sadrži sve iste elemente kao i Qt WebEngine Widgets modul, osim jedne razlike, nema zasebne komponente *web engine page*. Ona dolazi integrirana u prikaz odnosno *web engine view* što vidimo na 2.4.



Slika 2.4: Qt WebEngine modul

Qt Web Engine Core modul

Qt WebEngine Core modul utemljen je na *Chromium Project*-u. *Chromium Project* uključuje Chromium i Chromium OS, besplatne softverske projekte otvorenog koda (eng. *open source*) za preglednik Google Chrome i Google Chrome OS, redom. Važno je naglasiti da je Qt WebEngine baziran na Chromium-u, ali ne sadrži niti koristi bilo koje usluge koje bi mogle biti dio preglednika Chrome.

Verzija *Chromiuma* može se pročitati tijekom izvođenja pomoću metode `qWebEngineChromiumVersion()` i `qWebEngineChromiumSecurityPatchVersion()`. Također, verzije možete pronaći u datoteci `CHROMIUM_VERSION`.

Qt WebEngine Process modul

Qt WebEngine Process je zaseban modul koji se koristi za renderiranje *web* stranica i izvršavanje JavaScript-a. Time se ublažavaju sigurnosni problemi i izolira potencijalni pad programa.

2.5 Funkcionalnosti Qt WebEngine modula

Neke od funkcionalnosti koje nam nudi Qt WebEngine modul su:

- *Audio and Video Codecs*
- *Chromium DevTools*
- *Client Certificates*

- *Drag and Drop*
- *Fullscreen*
- *PDF File Viewing*
- *Print to PDF*
- *Spellchecker*
- *Touch*
- *View Source*
- *Web Notifications*

U nastavku ću detaljnije opisati *Chromium DevTools*, *PDF File Viewing*, *Print to PDF* i *Spellchecker*, a ostale fuknciоnalnosti i njihove specifikacije možete pogledati na [8].

Chromium DevTools

Chromium DevTools nam pruža mogućnost da pregledamo razmještaj (eng. *layout*) elemenata *web* stranice te nam pomaže u otklanjanju grešaka (eng. *debugging*).

Ovu funkcionalnost možemo testirati pokretanjem Qt WebEngine aplikacije sa sljedećom opcijom u komandnoj liniji

```
--remote-debugging-port=[your-port]
```

ili postavljanjem varijable okruženja `QTWEBENGINE_REMOTE_DEBUGGING`. Tada otvorimo neki od preglednika, koji se bazira na Chromiumu, i iskoristimo ga za povezivanje na `http://localhost:[your-port]`. (Naravno, umjesto `[your-port]` upišemo naš port.)

Isto tako, stranica *Chromium DevTools* se može prikazati unutar aplikacije. Kako bi to postavili, možemo pozvati metodu `QWebEnginePage::setInspectedPage()` ili metodu `QWebEnginePage::setDevToolsPage()` na stranicu koju želimo pregledati (eng. *inspect*). Tada će se implicitno učitati *DevTools*.

Odgovarajuća QML svojstva su redom, `WebEngineView.devToolsView` i `WebEngineView.inspectedView`.

PDF File Viewing i *Print to PDF*

Qt WebEngine podržava pregled PDF (eng. *Portable Document Format*) dokumenata. Koristi Chromiumov API i dodatak za PDF preglednik kako bi mogao prikazati PDF dokumente. Korištenjem enumeracije `QWebEngineSettings::PluginsEnabled` ili za QML aplikaciju `WebEngineSettings::pluginsEnabled`, možemo omogućiti učitavanje dodataka. Dok pomoću enumeracije `QWebEngineSettings::PdfViewerEnabled` ili svojstva `WebEngineSettings::pdfViewerEnabled` uključujemo ili isključujemo ovu funkcionalnost. Inicijalno je ova funkcionalnost uključena.

Također, Qt WebEngine podržava ispis *web* stranice u PDF datoteku. Metode koje se koriste pri tome su `QWebEnginePage::printToPdf()` za widget aplikaciju te za QML aplikaciju `WebEngineView.printToPdf`.

Spellchecker

Prilikom ispunjavanja HTML obrazaca, Qt WebEngine nam pruža podršku za provjeru pravopisa. To omogućuje korisnicima da prije slanja forme, provjere ispravnost podataka. U slučaju da je riječ pogrešno napisana, bit će crveno pocrtana te će korisnik moći kliknuti na nju i time dobiti izbornik koji prikazuje do četiri prijedloga. Odabirom jednog zamijenit će se pogrešno napisana riječ.

Za provjeru pravopisa su potrebni riječnici. Qt WebEngine podržava riječnike iz projekta *Hunspell*, ali ih je potrebno prevesti u poseban binarni format.

Po inicijalnim postavkama, provjera pravopisa je onemogućena. Može se omogućiti korištenjem metode `QWebEngineProfile::setSpellCheckEnabled()` za widget aplikaciju ili analogno korištenjem svojstva `WebEngineProfile.spellCheckEnabled` u Qt Quick aplikacijama.

2.6 Aplikacija i njene funkcionalnosti

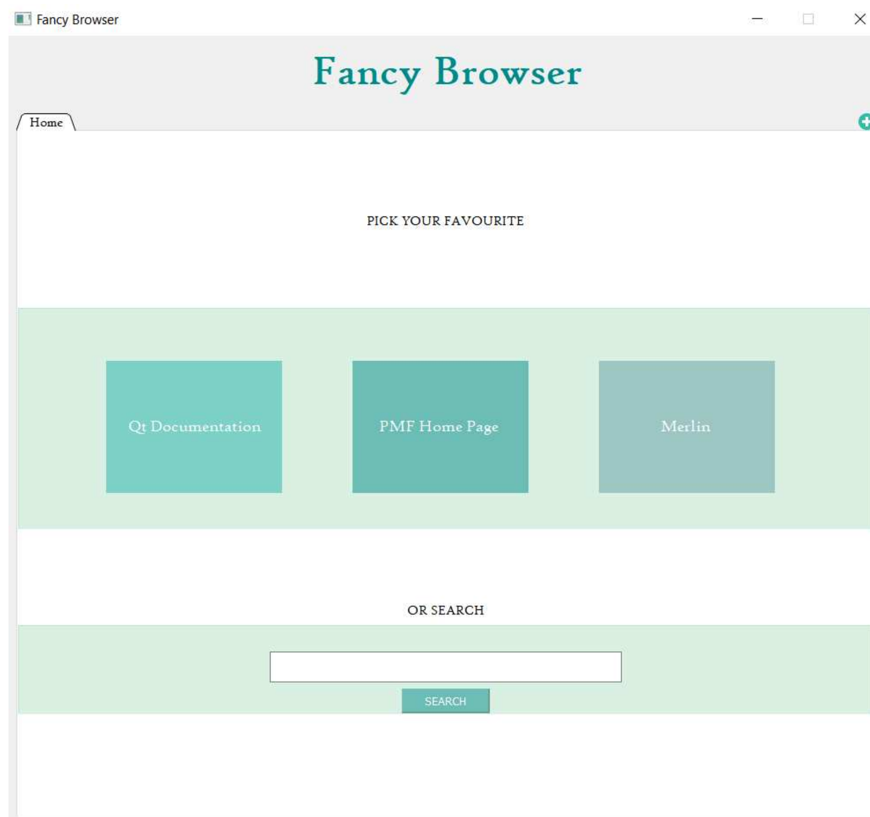
U Qt Creatoru izradila sam *web* aplikaciju koristeći Qt WebEngine modul. Aplikacija implementira internetski preglednik.

Funkcionalnosti koje su podržane i nalaze se na početnom zaslonu aplikacije su sljedeće:

- Klikom na jedan od tri pravokutnika (*Qt Documentation*, *PMF Home Page*, *Merlin*) otvara nam se izabrana *web* stranica u novoj kartici.

- U glavno polje odnosno linijski tekstualni editor možemo upisati pojam ili internetsku adresu po želji te klikom na *Search* gumb ili na tipku *Enter* u novoj kartici se otvara tražena internetska stranica.

- U gornjem desnom kutu, imamo gumb s ikonicom znaka plus. Prelaskom mišem preko njega, pojavljuje se *tooltip* s tekstom *Add new tab*. Klikom na gumb, otvara nam se nova kartica s praznim linijskim editorom koji predstavlja tražilicu.



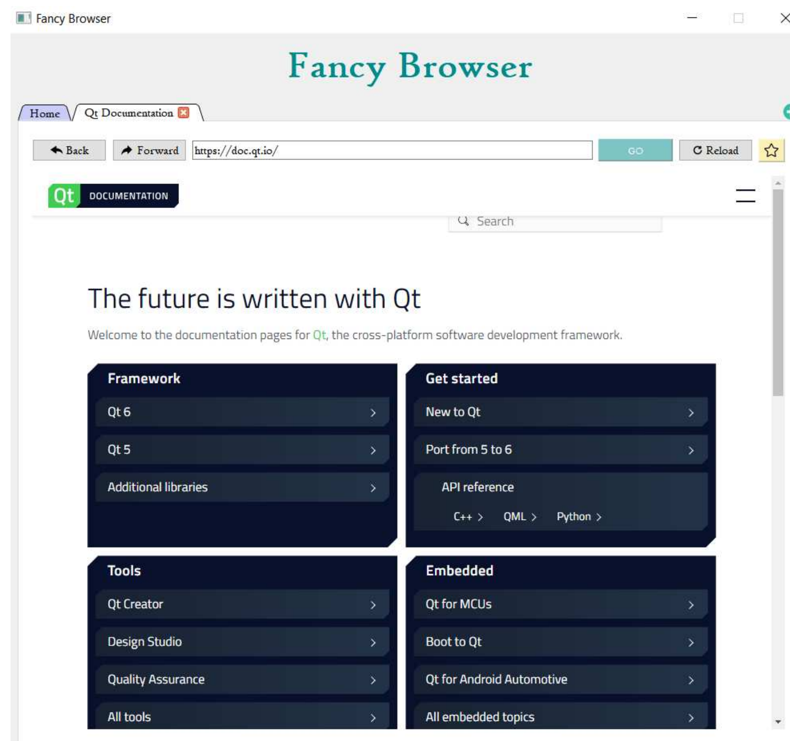
Slika 2.5: Početni zaslon aplikacije

Kada nam se otvori internetska stranica u novoj kartici, aplikacija nam nudi nove funkcionalnosti. Novi izgled aplikacije možemo vidjeti na slici 2.6, a nove funkcionalnosti su:

- U adresnoj traci, u kojoj vidimo URL trenutno otvorene internetske stranice, možemo mijenjati internetsku adresu ili upisati neki novi pojam te klikom na gumb *GO* ili klikom na tipku *Enter* ju i prikazati.

- Svaki put kada pretražujemo novi pojam, automatski se ažurira i naziv otvorene kartice.

- Klikom na gumb *Back*, vraćamo se na prethodno pretraživanu stranicu.
- Klikom na gumb *Forward*, otvara se sljedeća posjećena stranica.
- *Reload* gumb omogućuje osvježavanje internetske stranice.
- U slučaju da želimo spremiti URL stranice kako bi je kasnije mogli posjetiti, potrebno je kliknuti na žuti gumb s ikonicom zvjezdice i ona će se spremiti u listu omiljenih (eng. *Favourites*). Prelaskom mišem preko gumba, pojavljuje se *tooltip* s tekstom *Add to favourites*. Ako je lista prazna, neće biti prikazana na aplikaciji. U slučaju kada imamo barem jednu spremljenu omiljenu stranicu, lista će se prikazati kao na slici 2.7.
- Također, prilikom klika na žuti gumb s ikonicom zvjezdice, otvara nam se dijalog koji nam omogućuje da upišemo naziv pod kojim ćemo spremiti našu omiljenu stranicu. U slučaju da ne upišemo ništa, stranica će se spremiti u listu kao URL adresa.
- Klikom na neku internetsku adresu iz *Favourites* liste, u novoj kartici će se otvoriti odabrana stranica.
- Desnim klikom na karticu, otvara nam se izbornik s tri opcije. Možemo duplicirati trenutno otvorenu karticu, dodati karticu u listu omiljenih kartica i otvoriti potpuno novu karticu.
- Zatvaranje svih novootvorenih kartica je moguće klikom na crveni gumb za zatvaranje dok je isti izostavljen kod kartice *Home* kako bi nam ona uvijek bila dostupna u aplikaciji.
- Moguće je mijenjati redoslijed kartica klikom na neku karticu te pozicioniranjem je na željeno mjesto.

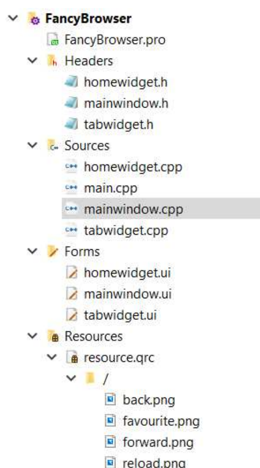


Slika 2.6: *Qt Documentation* stranica

Slika 2.7: Lista *Favourites*

2.7 Implementacija

Aplikacija *Fancy Browser* sastoji se od datoteka prikazanih na slici 2.8. U nastavku ćemo detaljno pojasniti najbitnije dijelove implementacije tih klasa.



Slika 2.8: Struktura datoteki aplikacije

Projektna datoteka

Qt sustav za izgradnju sastoji se od jedne datoteke s ekstenzijom `.pro`. U toj datoteci se nalaze sve bitne informacije vezane za projekt.

```
FancyBrowser.pro
1  greaterThan(QT_MAJOR_VERSION, 4): QT += core gui widgets webengine webenginewidgets
2
3  CONFIG += c++11
4
5  SOURCES += \
6      homewidget.cpp \
7      main.cpp \
8      mainwindow.cpp \
9      tabwidget.cpp
10
11 HEADERS += \
12     homewidget.h \
13     mainwindow.h \
14     tabwidget.h
15
16 FORMS += \
17     homewidget.ui \
18     mainwindow.ui \
19     tabwidget.ui
20
21 # Default rules for deployment.
22 qnx: target.path = /tmp/$${TARGET}/bin
23 else: unix:!android: target.path = /opt/$${TARGET}/bin
24 !isEmpty(target.path): INSTALLS += target
25
26 RESOURCES += \
27     resource.qrc
```

Slika 2.9: Projektna datoteka

U projektnu datoteku sam dodala sljedeću liniju:

```
QT += core gui widgets webengine webenginewidgets
```

kojom dodajemo module koje ćemo koristiti u aplikaciji. Inače ako ne uključimo niti jedan modul direktno, tada se uključuje samo Qt Core modul.

Linija

```
SOURCES += homewidget.cpp main.cpp mainwindow.cpp
           tabwidget.cpp
```

dodaje sve izvorne datoteke koje će sudjelovati u izgradnji.

Analogno, naredbom `HEADERS +=` dodajemo datoteke zaglavlja koje sudjeluju u izgradnji, a naredbom `FORMS +=` uključujemo datoteke formata `.ui`.

Bitno je naglasiti i resursnu datoteku koja je dodana aplikaciji pomoću naredbe:

```
RESOURCES += resource.qrc
```

Resursna datoteka

Iz prikaza aplikacije na slici 2.6 možemo uočiti da se u aplikaciji pojavljuju i neki resursi kao na primjer ikonica za gumb *Reload*. Pri distribuciji programa ikona mora ići uz izvršni program koji mora znati na kojoj se lokaciji nalazi ikona. To je prilično nezgodno za kod koji treba biti neovisan o o platformi na kojoj se izvršava. Stoga, Qt nam nudi podsistem koji se naziva *Qt Resource System* i koji dozvoljava da se binarne datoteke, koje bismo distribuirali zajedno s izvršnim programom (ikona za gumb *Reload* u ovom primjeru), uključe u izvršni kod i time budu uvijek dostupne aplikaciji. Svi resursi aplikacije moraju biti deklarirani u `.qrc` datoteki (eng. *Resource Collection* datoteka). Radi se o XML datoteci koju će tijekom procesa izgradnje aplikacije obraditi *rcc* kompajler. On generira izvornu C++ ili Python datoteku na temelju podataka u resursnoj datoteci. U slučaju moje aplikacije, ona ima oblik kao na 2.10.

(a) XML (b) Resursi

Slika 2.10: Resursna datoteka

U izvornom kodu resurse dohvaćamo s prefiksom `:/` odnosno u slučaju postavljanja ikone za gumb *Reload* to izgleda kao na slici 2.11.

```
ui->reloadButton->setIcon(QIcon(":/reload.png"));
```

Slika 2.11: Metoda *setIcon()*

Glavni program

Qt aplikacije su programi vođeni događajima. Cijeli program se nalazi unutar jedne beskonačne petlje unutar koje se distribuiraju događaji generirani mišom, tipkovnicom, GUI elementima i ostalima.

Fancy Browser aplikacija je `Qt Widget` aplikacija. Ona započinje instanciranjem klase `QApplication` koja uzima parametre komandne linije. Za svaku GUI aplikaciju koja koristi Qt, postoji točno jedan `QApplication` objekt koji se mora kreirati prije nego se kreiraju bilo koji drugi objekti. On će inicijalizirati sustav prozora, a u destruktoru će izvršiti potrebnu destrukciju cijelog sustava. U `return` naredbi se poziva metoda *exec()* na `QApplication` objektu koja pokreće glavnu petlju događaja. Aplikacija završava kada se zatvori glavni prozor, a time se prekida i metoda *exec()*.

Slika 2.12: Main.cpp datoteka

MainWindow klasa

U `Main.cpp` datoteci koristimo objekt klase `MainWindow`. Detaljniji izgled datoteke zaglavlja te klase vidimo na 2.13.

Ovo je klasa koja je odgovorna za glavni prozor aplikacije. Vidimo da u privatnom dijelu sadrži pokazivače na objekte klasa `HomeWidget` i `QTabWidget`. `HomeWidget` klasa modelira izgled i funkcionalnosti početnog zaslona dok je `QTabWidget` Qt klasa koja pruža

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QtWebEngineWidgets/QtWebEngineWidgets>

#include "homewidget.h"
#include "tabwidget.h"

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

protected:
    void hideFavouritesList(bool);

private slots:
    void openTab(QString, QString);
    void closeTab(int);
    void addNewFavourite(QString, QString);
    void addNewFavourite();
    void on_favComboBox_activated(const QString &arg1);
    void showContextMenu(const QPoint &point);
    void duplicateTab();
    void addNewTab();
    void updateTab();

private:
    Ui::MainWindow *ui;
    HomeWidget *homeWidget;
    QTabWidget *tabWidget;
    QToolButton *cornerButton;
    QWebEngineView* webview;

};
#endif // MAINWINDOW_H
```

Slika 2.13: Datoteka mainwindow.h

traku kartica (eng. *tab bar*) gdje je svaka od njih povezana s nekim *widget*-om. Objektu tog tipa ćemo pomoću metode *addTab()* dodavati kartice odnosno u našem slučaju jedan objekt tipa `HomeWidget` i više objekata tipa `TabWidget`. Klasa `TabWidget` predstavlja svaku novu karticu u kojoj će biti prikazana internetska stranica. Također, uočimo da klasa `MainWindow` sadrži i makro `Q_OBJECT` koji je obavezan za klasu koja koristi mehanizam signala i utora.

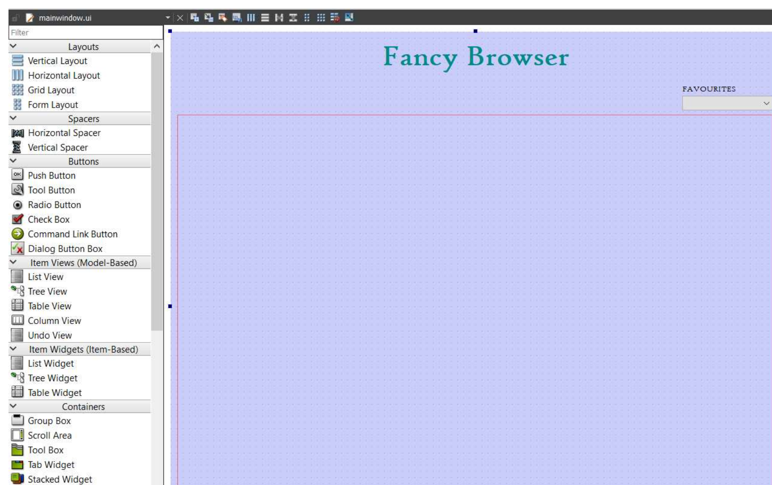
`HomeWidget` objekt dodajemo odmah napočetku u konstruktoru `MainWindow` klase dok se objekti tipa `TabWidget` dodaju u metodi *openTab()* te uklanjaju u metodi *closeTab()*. Na slici 2.13 vidimo da su obje spomenute metode definirane u dijelu privatnih utora (eng. *private slots*), a na slici 2.14 su dane implementacije tih metoda. Istaknula bih kako sam pomoću objekta tipa `QUrl` konstruirala URL koji se dalje prosljeđuje metodi *loadUrl()*. Ta metoda na objektu tipa `QWebEnginePage` učitava dani URL u stranicu. Također, na objektu tipa `NewTab` postavljam svojstva *url* i *title* što će biti bitno kasnije kod izbornika koji se otvara na desni klik.

```
void MainWindow::openTab(QString url, QString title)
{
    TabWidget *newTab = new TabWidget();

    QUrl *urlq = new  QUrl(url);
    if(!url.isEmpty())
        urlq->setScheme("https");

    newTab->loadUrl(*urlq);
    newTab->setProperty("url", url);
    newTab->setProperty("title", title);

    int index = tabWidget->addTab(newTab, title);
    tabWidget->setCurrentIndex(index);

    connect(newTab, SIGNAL(favouriteClickedSignal(QString, QString)),
            this, SLOT(addNewFavourite(QString, QString)));
    connect(newTab, SIGNAL(updateTabSignal()), this, SLOT(updateTab()) );
}

void MainWindow::closeTab(int index)
{
    tabWidget->removeTab(index);
}
```

Slika 2.14: Metode *openTab()* i *closeTab()* u mainwindow.cpp

Početni zaslon

U klasi `HomeWidget` implementiram početni zaslon aplikacije sa slike 2.5. On u okolini Qt Creator izgleda kao na 2.15. Pomoću objekta klase `QPainter` iscrtala sam tri pravokutnika u metodi *paintEvent()*. Klikom na površinu nekog od njih, korisniku se automatski otvara željena *web* stranica u novoj kartici. U pozadini se zapravo klikom na određeno područje, emitira signal *goButtonClickedSignal* s argumentom URL-a stranice.



Slika 2.15: Qt Creator

```
homewidget.h                    Ui
1  #ifndef HOMEWIDGET_H
2  #define HOMEWIDGET_H
3
4  #include <QWidget>
5  #include <QPainter>
6
7  namespace Ui {
8  class HomeWidget;
9  }
10
11 class HomeWidget : public QWidget
12 {
13     Q_OBJECT
14
15 public:
16     explicit HomeWidget(QWidget *parent = nullptr);
17     ~HomeWidget();
18
19 protected:
20     void paintEvent(QPaintEvent *event) override;
21     void mousePressEvent(QMouseEvent *event) override;
22     void calculateClickedSquare(int x, int y);
23     void search();
24
25 private slots:
26     void on_goButton_clicked();
27     void on_urlLineEdit_returnPressed();
28
29 signals:
30     void goButtonClickedSignal(QString, QString);
31
32 private:
33     Ui::HomeWidget *ui;
34 };
35
36 #endif // HOMEWIDGET_H
```

Slika 2.16: Datoteka homewidget.h

Izbornik

Jedna od funkcionalnosti koju sam spomenula je i otvaranje prilagođenog izbornika desnim klikom na karticu. (slika 2.17)

Kako bi to omogućili na objektu tipa `QTabWidget` treba postaviti *context menu policy* s metodom `setContextMenuPolicy(Qt::CustomContextMenu)`. Uz to, potrebno je i povezati odgovarajući signal *customContextMenuRequested* s utorom *showContextMenu* kojeg sami implementiramo. Na slici 2.18 vidimo da nam signal šalje i objekt tipa `QPoint`, a on sadrži koordinate klika miša. To nam je potrebno kako bi se izbornik prikazao upravo na tim koordinatama.

Pogledajmo sada kako izgleda implementacija metode odnosno utora koji kreira izbornik i u njega dodaje potrebne elemente. Izbornik kreiramo pomoću klase `QMenu`. Opcije koje su moguće izabrati unutar izbornika, dodavat ćemo pomoću metode *addAction* na objektu tipa `QMenu`. Vidimo na 2.19 da sam kreirala tri akcije za izbornik. Izbornik nam nudi opciju za dupliciranje kartice (*duplicateAction*), dodavanje trenutno otvorene stranice

Slika 2.17: Izbornik

```
connect(tabWidget, SIGNAL(customContextMenuRequested(const QPoint &)),
        this, SLOT(showContextMenu(const QPoint &)));
```

Slika 2.18: *Connect* metoda izbornika

u listu omiljenih stranica (*addToFavouritesAction*) i otvaranje nove prazne kartice (*addTabAction*). Pomoću klase `QAction` kreiramo navedene akcije te pomoću njenog signala registriramo kada je korisnik kliknuo na neku od opcija odnosno kada je aktivirao određenu akciju. Signal *triggered()* je povezan s utorima koji dalje implementiraju traženu fukcionalnost.

```
void MainWindow::showContextMenu(const QPoint &point)
{
    if (point.isNull())
        return;

    QMenu menu(this);

    QAction duplicateAction("Duplicate", this);
    connect(&duplicateAction, SIGNAL(triggered()), this, SLOT(duplicateTab()));

    QAction addToFavouritesAction("Add to favourites", this);
    connect(&addToFavouritesAction, SIGNAL(triggered()), this, SLOT(addNewFavourite()));

    QAction addTabAction("Add new tab", this);
    connect(&addTabAction, SIGNAL(triggered()), this, SLOT(addNewTab()));

    menu.addAction(&duplicateAction);
    menu.addAction(&addToFavouritesAction);
    menu.addAction(&addTabAction);

    menu.exec(tabWidget->mapToGlobal(point));
}
```

Slika 2.19: Utor *showContextMenu*

Primjer dupliciranja određene kartice ću pojasniti detaljnije u nastavku. Implementaciju utora *duplicateTab()* vidimo na slici 2.20. Naime, karticu iz objekta tipa `QTabWidget` možemo dobiti preko metode `QTabWidget::widget(int index)` koja za povratni tip ima `QWidget`. Kako sam prilikom svakog dodavanja nove kartice postavila svojstva *url* i *title* (slika 2.14), to nam sada omogućava da ta svojstva pročitamo i iz objekta tipa `QWidget`. Pomoću njih znamo naziv nove kartice te znamo koju sljedeću internetsku stranicu moramo otvoriti. Jedino što je još preostalo nakon dohvaćanja tih svojstva je prosljeđivanje istih već postojećoj metodi *openNewTab* koju sam pojasnila prethodno u ovom poglavlju.

```
void MainWindow::duplicateTab()
{
    int index = tabWidget->currentIndex();
    QWidget *widget = tabWidget->widget(index);

    openTab(widget->property("url").toString(),
            widget->property("title").toString());
}
```

Slika 2.20: Utor *duplicate()*

Mehanizam signala i utora

U prethodnom potpoglavlju 2.7 sam spomenula primjer mehanizma signala i utora koji je jedan od glavnih značajki Qt-a. Ovdje ga detaljnije objašnjavamo na konkretnom primjeru otvaranja i zatvaranja kartica.

U primjeru 2.21 radi se o dva poziva funkcije *connect()* unutar konstruktora klase `MainWindow`.

```
connect(homeWidget, SIGNAL(goButtonClickedSignal(QString, QString)), this, SLOT(openTab(QString, QString)));
connect(tabWidget, SIGNAL(tabCloseRequested(int)), this, SLOT(closeTab(int)));
```

Slika 2.21: Metoda *connect()*

Naime, prvim pozivom te metode želimo postići da svaki put kada korisnik klikne na gumb *GO* na početnom zaslonu (slika 2.5), otvori nam se tražena internetska stranica u novoj kartici. Zapravo, kada objekt klase `HomeWidget` emitira signal *goButtonClickedSignal*, definirani utor odnosno metoda *openTab* u klasi `MainWindow` će reagirati na taj signal. Nisam koristila već definirane Qt signale i utore već sam u ovom primjeru implementirale svoje.

Na slici 2.22 vidimo da je pod oznakom **signals** definiran novi signal u klasi `HomeWidget`. Kada korisnik na početnom zaslonu klikne gumb *GO* ili tipku *Enter* za otvaranje stranice

u novoj kartici, na slici 2.23 možemo primjetiti kako će se u oba slučaja pozvati metoda *search()* koja će emitirati signal *goButtonClickedSignal* s URL-om stranice koju korisnik želi otvoriti i tekstom iz *LineEdit* polja.

Utor *openTab* definiran pod oznakom **private slots** u klasi `MainWindow` (2.13), osluškuje emitiranje signala *goButtonClickedSignal*. Kada se pozove utor, objektu klase `QTabWidget` dodajemo novu karticu implementiranu klasom `TabWidget` i pozivamo metodu *setCurrentIndex()* koja prebacuje fokus na novu dodanu karticu.

```
signals:
    void goButtonClickedSignal(QString, QString);
```

Slika 2.22: Signal u homewidget.h datoteci

```
void HomeWidget::on_goButton_clicked()
{
    search();
}

void HomeWidget::on_urlLineEdit_returnPressed()
{
    search();
}

void HomeWidget::search()
{
    emit goButtonClickedSignal("https://www.google.com/search?q=" + ui->urlLineEdit->text(),
                               ui->urlLineEdit->text());
}
```

Slika 2.23: Emitiranje signala u homewidget.cpp datoteci

U drugom primjeru, radi se o povezivanju Qt predefiniranog signala *tabCloseRequested* iz klase `TabWidget` i novodefiniranog utora *closeTab* klase `MainWindow` koji metodom *removeTab()* uklanja karticu određenog indeksa.

Primjetimo, kako u oba primjera poziva metode *connect()* i signali i utori primaju isti broj argumenata istog tipa.

TabWidget klasa

WebEngine

Ideja klase `TabWidget` je da implementira potrebne funkcionalnosti za karticu koja prikazuje *web* stranicu. Njen izgled vidimo na slici 2.6. U privatnom dijelu ove klase se nalazi pokazivač na objekt klase `QWebEngineView`. Klasa `QWebEngineView` je glavni *widget* WebEngine modula zaslužan za prikazivanje internetskih stranica. U konstruktoru klase `TabWidget` inicijaliziramo i dodajemo `QWebEngineView` *layout*-u. (slika 2.24)

```
webview = new QWebEngineView;
ui->verticalLayout->addWidget(webview);
```

Slika 2.24: Datoteka tabwidget.cpp

Internetska stranica se učitava pomoću metode *load()* kojoj proslijedimo URL stranice. Na klik gumba *GO* ili tipke *Enter*, pozvat će se metoda *search()* čiju implementaciju vidimo na slici 2.25 koja poziva spomenutu metodu *load()*.

Isto tako, ovdje provjeravamo da li je korisnik u tražilicu upisao URL stranice ili pojam te sukladno tome, učitavamo *web* stranicu. Metoda *updateTabProperties()* ažurira svojstva *url* i *title* kartice i emitira signal za ažuriranje naziva kartice prilikom svakog novog pretraživanja. Objekt tipa `MainWIndow` sadrži utor koji osluškuje taj signal i vodi računa o tome da kartica ima najnoviji naziv odnosno mijenja naslov svakim pojavljivanjem signala.

```
void TabWidget::on_urlLineEdit_returnPressed()
{
    search();
}

void TabWidget::on_goButton_clicked()
{
    search();
}

void TabWidget::search()
{
    QString urlLineEditText = ui->urlLineEdit->text();

    if(urlLineEditText.left(4) == "http"){
        webview->page()->load(urlLineEditText);
        updateTabProperties(urlLineEditText, urlLineEditText);
    }
    else{
        webview->page()->load("https://www.google.com/search?q=" + urlLineEditText);
        updateTabProperties("https://www.google.com/search?q=" + urlLineEditText,
                            urlLineEditText);
   }
}
```

Slika 2.25: Metoda *search()*

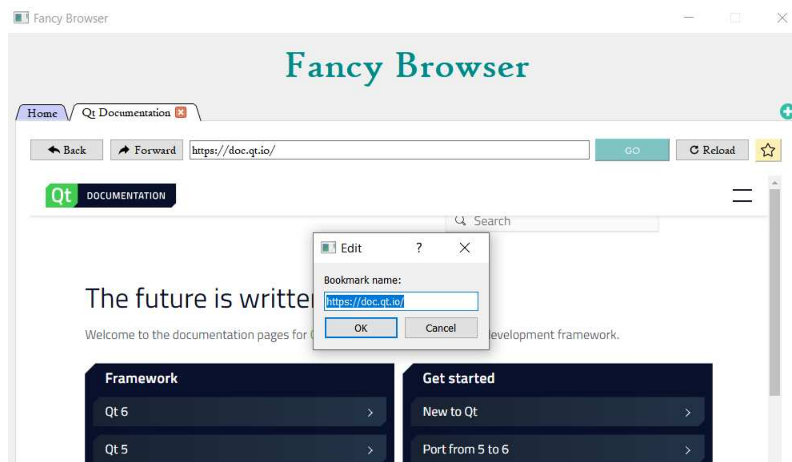
Na sljedećoj slici 2.26 vidimo kako su implementirane funkcije u pozadini gumbova *Back*, *Forward* i *Reload*. Na klik tih gumbova, na `webview` objektu pozivaju se metode *back()*, *forward()* i *reload()* redom, koje nam omogućuju navigiranje kroz posjećene stranice te osvježavanje istih.

```
void TabWidget::on_backButton_clicked()
{
    webview->back();
}

void TabWidget::on_forwardButton_clicked()
{
    webview->forward();
}

void TabWidget::on_reloadButton_clicked()
{
    webview->reload();
}
```

Slika 2.26: Metode *back()*, *forward()*, *reload()*

Lista *Favourites*

Također, u klasi `TabWidget` sam implementirala funkcionalnosti gumba koji dodaje omiljene stranice u listu *Favourites*. Naime, klikom na žuti gumb otvara nam se dijalog kao na slici 2.27. On nam nudi da upišemo ime pod kojim ćemo spremiti našu stranicu u listu omiljenih stranica. Kao zadani naziv, nudi se URL adresa stranice te ako ne upišemo ništa, u listu *Favourites* naša stranica će se spremiti pod tim nazivom.



Slika 2.27: Dijalog

Za implementaciju ove funkcionalnosti koristila sam Qt klasu `QInputDialog` (slika 2.28). Metoda *getText()* zapravo otvara dijalog te ako je korisnik kliknuo OK vraća tekst iz polja za unos, inače vraća *null* `QString`. U ovom slučaju, ako je korisnik kliknuo OK, emitira se signal *favouriteClickedSignal* koji je povezan s utorom za dodavanje novih stranica u listu *Favourites*.

```
void TabWidget::on_favouriteButton_clicked()
{
    bool ok;
    QString text = QInputDialog::getText(this, tr("Edit"),
                                         tr("Bookmark name:"), QLineEdit::Normal,
                                         ui->urlLineEdit->text(), &ok);
    if (ok && !text.isEmpty()){
        emit favouriteClickedSignal(text);
    }
}
```

Slika 2.28: Utor *on_favouriteButton_clicked()*

Bibliografija

[1] The Qt Company, *Introducing the Qt WebEngine*, https://www.qt.io/blog/2013/09/12/introducing-the-qt-webengine.

[2] ______, *The Meta-Object System*, https://doc.qt.io/archives/qt-4.8/metaobjects.html.

[3] ______, *Porting from Qt WebKit to Qt WebEngine*, https://doc.qt.io/qt-5/qtwebenginewidgets-qtwebkitportingguide.html#qwebframe-has-been-merged-into-qwebenginepage.

[4] ______, *qmake Language*, https://doc.qt.io/qt-6/qmake-language.html.

[5] ______, *Qt Documentation*, https://doc.qt.io/.

[6] ______, *Qt Group*, https://www.qt.io/company.

[7] ______, *Qt History*, https://wiki.qt.io/Qt_History.

[8] ______, *Qt WebEngine Features*, https://doc.qt.io/qt-5/qtwebengine-features.html#web-notifications.

[9] ______, *Qt Webengine*, https://doc.qt.io/qt-5/qtwebengine-overview.html.

[10] ______, *Qt Widgets vs Qt Quick*, https://resources.qt.io/videos/qt-widgets-or-qt-quick-2.

[11] Lee Zhi Eng, *Qt5 C++ GUI Programming Cookbook Second Edition*, Packt Publishing, 2019.

[12] Alan Ezust i Paul Ezust, *An Introduction to Design Patterns in C++ with Qt 4*, Prentice Hall, 2006.

[13] Flylib.com, *Makefile, qmake, and Project Files*, https://flylib.com/books/en/2.385.1/makefile_qmake_and_project_files.html.

[14] Google, *The Chromium projects*, https://www.chromium.org/chromium-projects/.

[15] Mladen Jurak, *Alat cmake*, https://web.math.pmf.unizg.hr/nastava/napcpp/vjezbe-cmake.html.

[16] ———, *Oblikovni obrasci*, https://web.math.pmf.unizg.hr/nastava/opepp/old/Slides/Predavanja/html-noslides/slides-6-1.html.

[17] Qt Wiki, *Qt Creator*, https://wiki.qt.io/Qt_Creator.

[18] Wikipedia, *Grafičko korisničko sučelje*, https://hr.wikipedia.org/wiki/Grafi%C4%8Dko_korisni%C4%8Dko_su%C4%8Delje.

Sažetak

Qt je skup biblioteka i alata namijenjenih za izradu aplikacija. Neke njegove karakteristike poput meta objektnog sustava, mehanizma signala i utora opisne su u prvom poglavlju. Sastoji se od raznih modula kao što su Qt Core, Qt Quick, Qt Widgets i drugih. U ovom radu fokusirala sam se prvenstveno na Qt WebEngine modul. Kroz izradu aplikacije *Fancy Browser* u Qt Creatoru koristila sam funkcionalnosti koje nude ovaj i drugi moduli Qt-a. Aplikacija realizira internetski preglednik kojim možemo pretraživati proizvoljne *web* stranice te najdraže od njih spremati u listu omiljenih. Detalje o izgledu i implementaciji aplikacije opisala sam u zadnjem poglavlju ovog rada.

Summary

Qt is a set of libraries and tools designed to build applications. Its main characteristics such as the meta-object system, signal and slots are described in the first chapter. It consists of various modules such as Qt Core, Qt Quick, Qt Widgets and others. I mainly focused on the module Qt WebEngine. Qt Widget application Fancy Browser was created in Qt Creator using the functionalities provided by this and other modules in Qt. The application implements web browser that embeds content from the World Wide Web and offers saving favourites web pages. Application layout and implementation details I described in the last chapter.

Životopis

Aida Omanović rođena je 17. siječnja 1997. godine u Zagrebu. Školovanje je započela godine 2003. u Osnovnoj školi Augusta Šenoe. Nakon završetka, 2011. upisuje IX. gimnaziju. Godine 2015. upisuje preddiplomski sveučilišni studij Matematika na Prirodoslovno-matematičkom fakultetu u Zagrebu. Uspješnim završetkom studija stječe akademski naziv Baccalaurea matematike (sveučilišna prvostupnica matematike) te dalje nastavlja s obrazovanjem na diplomskom studiju Računarstvo i matematika na istom fakultetu u Zagrebu. Tijekom posljednje godine studija radi u tvrtci *AG04 Innovative solutions* na poziciji softverskog injženjera.