UNIVERSITÀ DEGLI STUDI DI PADOVA

“InkMark”

PROGETTO DI PROGRAMMAZIONE AD OGGETTI

A cura di

Giovanni Jiayi Hu. Matricola: 1122347

Anno Accademico 2016-2017

Presentazione

Inkmark

Inkmark è un'applicazione desktop che permette la gestione di bookmarks/segnalibri di diverso genere e in comune a più utenti. Un caso d'uso dell'applicazione potrebbe difatti essere una collezione di links a risorse utili per il corso di Programmazione ad oggetti, con la possibilità per gli studenti di aggiungerne e per il docente di gestire l'intera collezione e gli utenti stessi. L'applicazione si ispira parzialmente a Dropmark per concetti riguardanti i bookmarks, ma la GUI e il sistema di utenti sono totalmente custom.

L'applicazione permette di aggiungere i bookmarks passando l'URL della risorsa, un titolo breve ed una descrizione. Ciascun link può essere un bookmark semplice oppure essere un bookmark di genere Articolo o di genere Video, qualora rimandino a pubblicazioni, blog posts o video su piattaforme quali Youtube, Vimeo e Twitch.

Il progetto è stato sviluppato seguendo i principi del pattern **MVC** (Model-View-Controller) e le best-practises della programmazione OOP. La parte logica è composta da classi con suffisso *-Model*, che implementano interfacce con suffisso *-Interface*. Le stesse interfacce sono utilizzate dalle views, che quindi nulla sanno dell'implementazione del *Model* o possono solo utilizzare le Interfacce che dispongono solo di metodi const. La comunicazione *Model<=>View* avviene quindi attraverso *Controllers*, ognuno dei quali dispone del *Model* applicativo e la *View* sotto il proprio controllo.

Model

La parte logica è costistuita dalle classi Model che definisco i dati di interesse per l'applicazione. Le principali classi sono:

- ApplicationModel
- BookmarkModel
    - ArticleModel
    - VideoModel
- UserModel
    - AdminModel
    - GuestModel

ApplicationModel

L'ApplicationModel è un **singleton**, ovvero esiste un'unica sua istanza in tutta l'app, e contiene i dati di interesse per l'applicazione. Difatti possiede come campi dati la lista dei bookmarks mostrati e la lista degli utenti registrati all'applicazione. Si occupa inoltre di gestire l'autenticazione, leggere/salvare i dati sul file e qualsiasi altra operazione sulla lista di bookmarks ed utenti, opportunamente delegando l'implementazione alle classi di quest'ultime. Tutte quest'ultime operazioni sono implementate come *SLOTs* e la loro esecuzione con successo provoca l'emissione di un relativo *SIGNAL*. Questo sistema slots-signal verrà ampiamente utilizzato dai controllers.

Poiché la classe è singleton non sono stati implementati i costruttori di copia e l'operatore di assegnazione, in quanto non ci potranno mai essere due istanze di ApplicationModel in contemporanea.

Nonostante ApplicationModel gestisca i dati dell'intera app essa delega la maggior parte delle operazioni alle classi dei suoi campi dati e al crescere dell'applicazione può essere suddivisa in sotto-parti, ad esempio una classe per la lista di bookmarks ed una per la lista di utenti. Tuttavia mantiene il vantaggio di rappresentare un'intuitiva organizzazione dei dati e delle loro reciproche relazioni. Ogni Controller è difatti collegato all'ApplicationModel e rimanere in ascolto di eventi riguardanti gli utenti, i bookmarks o l'autenticazione.

BookmarkModel

BookmarkModel è la classe Model che rappresenta un bookmark generico ed implementa l'interfaccia **BookmarkInterface**. Le interfacce sono classi che **contengono unicamente metodi virtuali puri ed in questo caso tutti i metodi sono anche const**. Difatti le interfacce sono ideali per l'utilizzo nelle *Views*, in quanto rende quest'ultime completamente indipendenti/ignare del *Model* ed impedisce addirittura qualsiasi modifica diretta al *Model* da parte loro. Grazie all'uso esclusivo delle interfacce nei controllers e nelle views si ottiene un **completo polimorfismo ed information hiding**.

VideoInterface eredita da *BookmarkInterface* ed è implementata da **VideoModel**. VideoModel rappresenta un bookmark ad un video e possiede in più i campi dati "*platform*" e "*duration*" per rispettivamente la piattaforma (Youtube, Vimeo o Twitch) e la durata del video.

Analogamente **ArticleInterface** eredita pure da *BookmarkInterface* ed è implementata da **ArticleModel**. ArticleModel rappresenta invece un bookmark ad un articolo o blog posts ed è caratterizzato dai campi dati "*publication*" e "*minRead*" per rispettivamente la data di pubblicazione e il tempo di lettura estimato.

UserModel

Tutte le classi degli utenti ereditano dall'interfaccia **UserInterface**, seguendo un ragionamento simile a quello per i bookmarks.

Troviamo in questo caso la classe **UserModel**, che rappresenta un utente registrato con privilegi base. Egli difatti può visualizzare tutti i bookmarks e anche aggiungerne di propri, ma NON può modificare o cancellare segnalibri che non siano stati creati da lui. Inoltre non può accedere all'area admin, ove è possibile gestire gli utenti dell'applicazione e che è dunque riservata solo agli admin.

Ogni UserModel ha inoltre un **id univoco**, che viene incrementato ogni volta che ne viene creata un'istanza. Difatti costruttore di copia e operatore di assegnazione sono stati ridefiniti per mantenere consistente questa caratteristica.

Un'ulteriore classe di utenti sono gli **AdminModel**, che ereditano da UserModel ma hanno maggiori privilegi. Essi possono infatti accedere sia all'area utenti che admin e hanno i privilegi per qualsiasi tipo di operazione di lettura, scrittura, creazione e modifica.

Infine vi sono i **GuestModel**, che ereditano da UserInterface (e non da UserModel). Essi non sono utenti a tutti gli effetti, ma bensì permettono ad una persona di accedere ai bookmarks senza doversi registrare. Tuttavia possono solamente visualizzare e cercare bookmarks, non possono crearne di nuovi o modificare quelli esistenti.

Persistenza dati

All'avvio dell'applicazione vengono letti i dati dal file *"model.json"* e caricati in ApplicationModel tramite "*readFromJSON"*, che internamente usa l'omonimo metodo virtuale delle classi UserModel e BookmarkModel.
Alla chiusura dell'applicazione viene invece intercettato l'evento e avviene il salvataggio in *"model.json"* tramite "*writeToJSON"* dell'ApplicationModel, prima di terminare definitivamente l'applicazione.

Controllers

I **Controllers** sono le classi che si occupano di far ponte nella **comunicazione Model<=>Views** mantenendo quest'ultime completamente ignare l'una dell'altra. Anzi sia il Model che le Views sono ignare persino dei Controllers.
Ciascun controller possiede come campi dati "*model"* e "*view"*, ove model è l'istanza di ApplicationModel unica per l'app e view è un'istanza della view di cui si occupa il Controller. Il compito del Controller è connettere, tramite *QObject::connect*, SIGNALs del Model a SLOTs della View e viceversa connettere SIGNALs della View a SLOTs del Model.
Esempio: *bookmark_list_controller* connette il signal *addedBookmark* del Model allo slot *addBookmarkView* della View affinché venga mostrato nella GUI il nuovo bookmark qualora venga creato.

Questo sistema permette non solo completa indipendenza tra Model e View ma anche un'implementazione meno programmatica delle operazioni. Nell'esempio precedente difatti la view non sa se tale nuovo bookmark è il risultato di una propria precedente azione nel form di creazione o se sia stato creato per altri motivi. Non fa differenza quale sia la causa, si occupa solamente di mostrare il nuovo bookmarks quando è presente uno nuovo. Questo pattern sfrutta e rispetta la **filosofia ad eventi del sistema a SIGNAL e SLOT di Qt**.

Le classi controllers sono le seguenti, ve n'è una per ogni view che abbia bisogno di comunicare col Model:

- **ApplicationController**: si occupa di gestire l'autenticazione e di istanziare i controllers *AdminApplicationController* e *UserApplicationController*
- **AdminApplicationController**: controller dell'area admin. Si occupa di istanziare i sotto-controllers delle views nell'area admin.
    - UsersListController: controller per la view che mostra la lista di utenti e permette la modifica e cancellazione di questi
- **UserApplicationController**: analogo a AdminApplicationController ma per l'area utenti
    - AddBookmarkController: controller per la view di creazione bookmarks

    - BookmarksListController: controller per la view di modifica e cancellazione bookmarks
    - SearchBookmarkController: : controller per la view di ricerca bookmarks
- **AuthController**: si occupa di gestire i form di login e registrazione

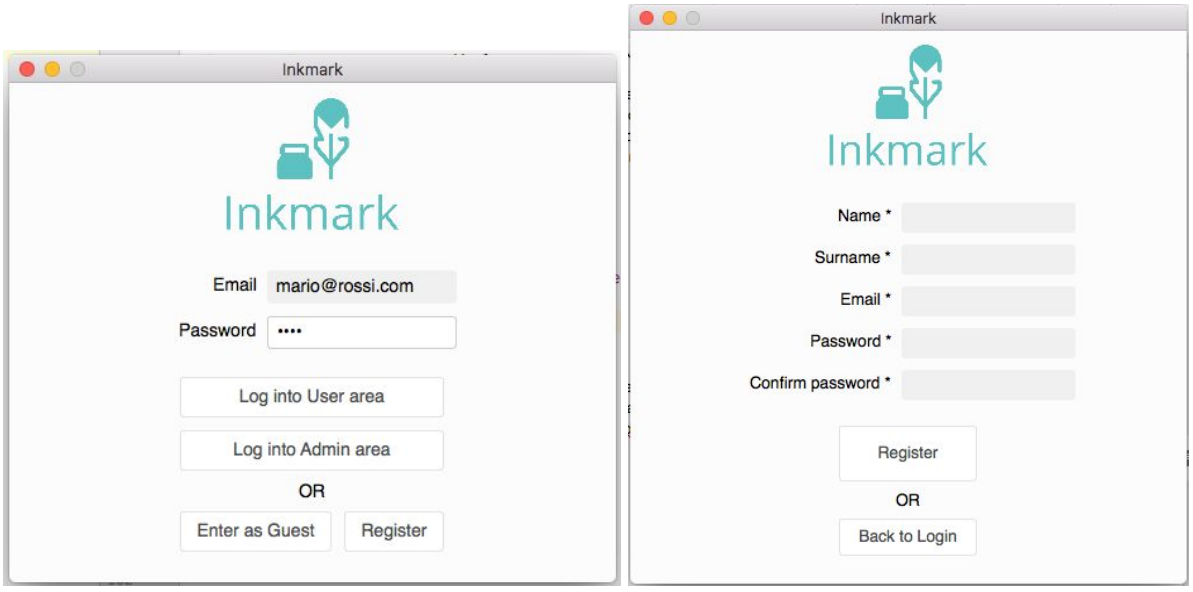
Views (GUI)

La GUI è stata sviluppata utilizzando i Widgets forniti da Qt ed in maniera totalmente custom, ovvero senza il supporto di QtDesigner, al fine di avere un codice più leggibile e minimal per le esigenze dell'applicazione.

La GUI è divisa in tre aree: admin, utente e login/registrazione.

Ognuna delle 3 fa uso di alcuni widgets contenuti nella cartella "*widgets*" quali ad esempio *ButtonWidget*. Sono classi widget utilizzate dalle tre aree, ereditano da quelle di Qt ed implementano uno stile grafico custom per l'applicazione Inkmark. In tal modo lo stile grafico per bottoni e campi di testo è uniforme in tutta l'applicazione, oltre che risparmiare molto codice duplicato.

Area login/registrazione



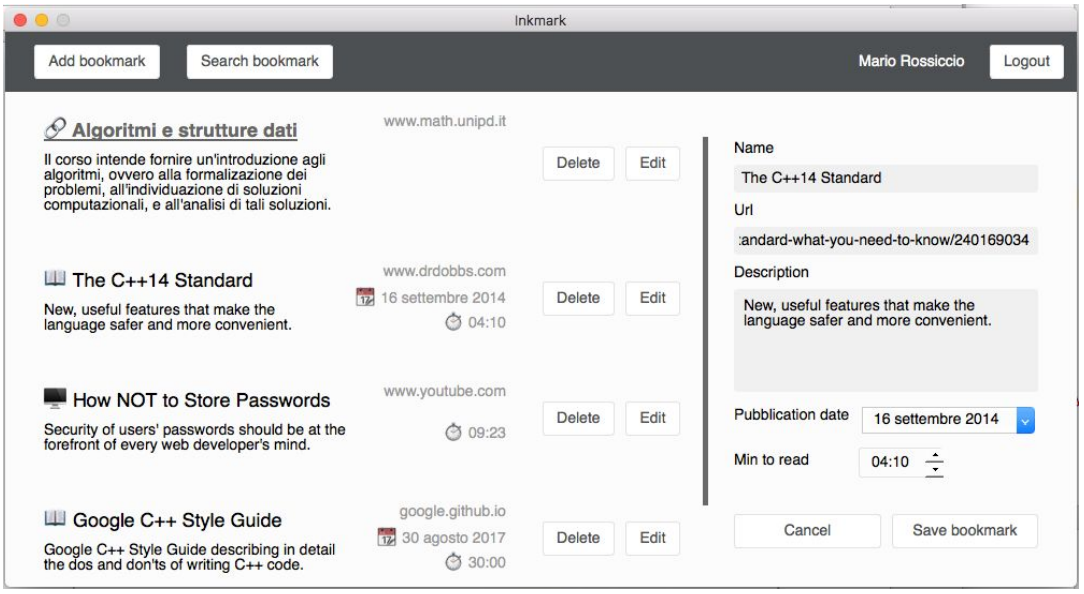
All'avvio dell'applicazione viene mostrata l'area di autenticazione. E' possibile da essa fare login all'area utente o all'area admin. Nel secondo caso sono richiesti i privilegi da admin.

E' anche possibile accedere all'area utente come Guest, ovvero ospite, senza doversi quindi registrare. In tal caso si potrà visualizzare e cercare tra i bookmarks, ma né crearne, modificarne o cancellarne.

Infine è possibile anche registrarsi fornendo nome, cognome, email e password. La registrazione porta ad accedere automaticamente come utente normale UserModel. Non è possibile registrarsi come admin. Per diventare admin è necessario che sia un altro admin a settare tale ruolo nell'area admin. Inizialmente vi è per l'app un solo admin.

In caso di dati di login errati, di campi mancanti o errati per la registrazione viene mostrato un box di errore come illustrato nella sezione *Gestione errori utenti* di seguito nel documento.

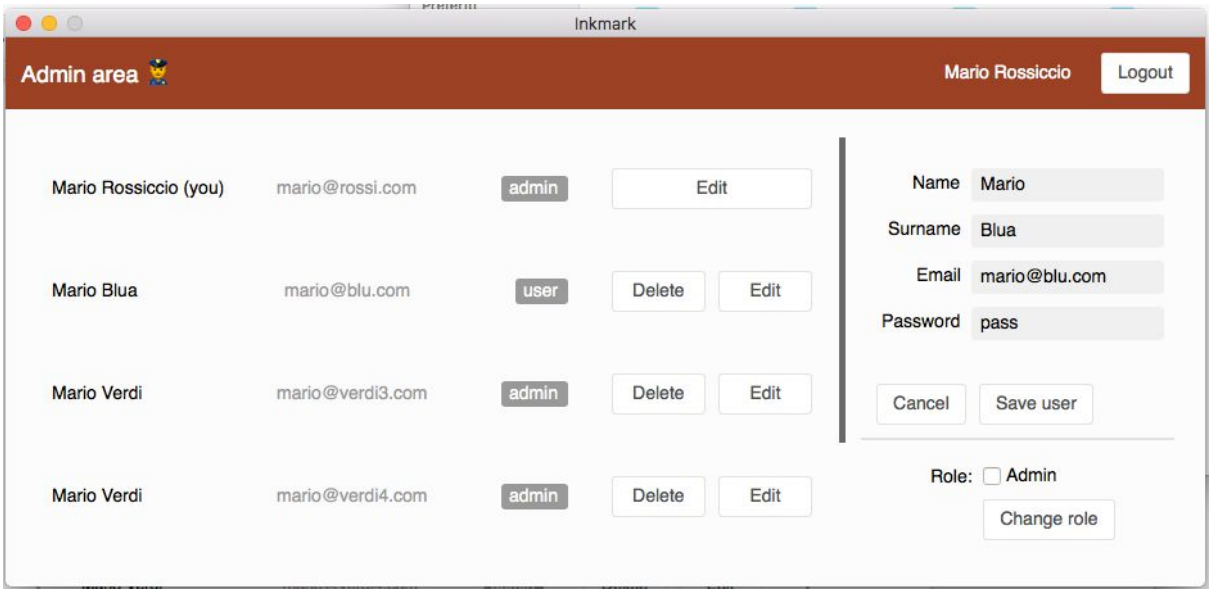
Area utente



L'area utente permette di visualizzare i bookmarks dell'applicazione. In cima vi è il menu che mostra il nome e cognome dell'utente loggato, oltre al bottone per fare Logout.
Sulla sinistra del menù invece vi sono i bottoni per aggiungere o cercare bookmark. Ciascuno di essi apre un pannello laterale a sinistra, simile a quello mostrato nell'immagine, sebbene questi compaia a destra e sia per l'editing di un bookmark.

Ogni bookmark mostra il titolo clickabile, che apre nel browser la pagina del link. Altri dati mostrati sono la descrizione, il dominio del link e altre informazioni che dipendono dalla tipologia del bookmark, quali data di pubblicazione per gli articoli e tempo di durata per i video.
Accanto a ciascun bookmark vi sono anche i bottoni per cancellarlo o editarlo. Nel caso dell'immagine l'utente loggato è admin e ha dunque tutti i privilegi. Un utente normale invece vedrebbe tali bottoni **solo** per i link aggiunti da egli stesso.

Area admin

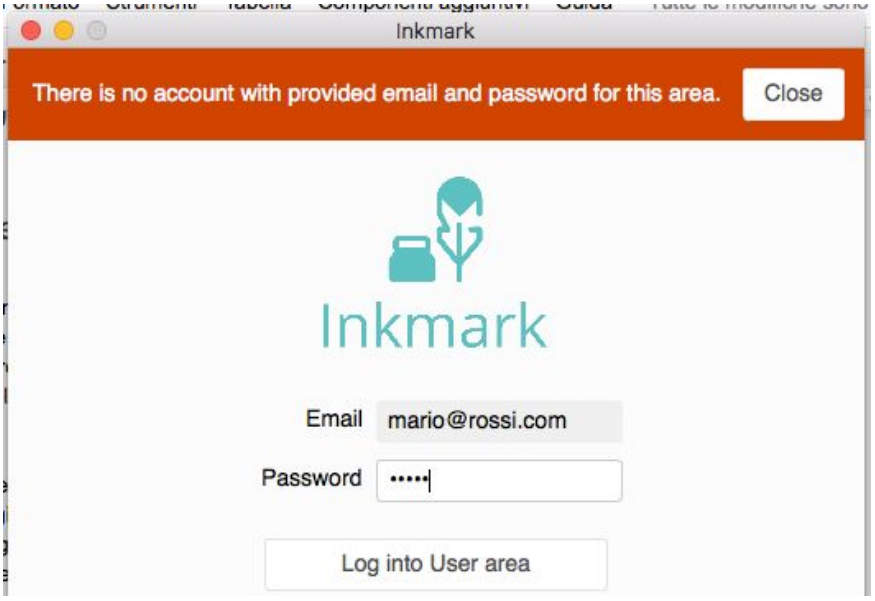


L'area admin è accessibile solo dagli admin e permette di gestire gli utenti registrati all'applicazione. La GUI è simile all'area utente ed è possibile visualizzare per ciascuno il nome, cognome, email e ruolo oltre ai bottoni per cancellare e modificare.

Non è possibile cancellare se stessi né cambiare il proprio ruolo, per evitare che un admin si auto-degradi di ruolo e non ne rimangano più. D'altra parte un admin può cambiare il ruolo di un altro utente, promuovendolo ad admin oppure cambiandolo a semplice utente.

Gestione errori utente

L'applicazione gestisce anche errori dell'utente, ovvero quel tipo di errore causato da un'azione non consentita e la cui possibile occorrenza è prevista dall'app. Nello specifico la classe ApplicationModel può emettere un SIGNAL "*hadUserError(const QString &)"* passando il testo dell'errore e questi viene mostrato nella GUI all'utente.



Gli errori utenti gestiti dall'app sono:

- Login di utente inesistente
- Registrazione con campi mancanti o con password diversa da quella di conferma
- Inserimento di bookmark con campi titolo e url assenti o con url invalido
- Ricerca di bookmarks senza risultati
- Modifica di ruolo del proprio account (non è possibile cambiare il ruolo a se stessi nemmeno da admin)

Ambiente di sviluppo

Informazioni sull'ambiente di sviluppo utilizzato qualora siano necessarie:

- Sistema operativo: macOS 10.12
- Versione Qt: 5.5.1
- Compilatore: Apple LLVM version 8.1.0 (clang-802.0.38) ma testato anche con G++

E' anche possibile accedere al repository Github del progetto per la totalità dei files originali: https://github.com/jiayihu/inkmark

Materiale consegnato

La cartella del progetto contiene i seguenti files:

- `src`: contiene i files sorgenti .cpp e .h del progetto e l'immagine logo

- `main.cpp:` file di partenza dell'applicazione
- `inkmark.pro`: Qt project file
- `inkmark.qrc`: file di configurazione Qt per le risorse dell'app, ovvero il logo
- `model.json`: file JSON di salvataggio dei dati
- `relazione.pdf`: Relazione del progetto

Compilazione ed esecuzione

I comandi per compilare l'applicazione sono:

```
qmake inkmark.pro
make
```

Non è necessario eseguire `` `qmake -project` `` in quanto esiste già il file .pro

Per l'accesso nell'applicazione sono richiesti i dati di login, sebbene sia possibile registrarsi come utenti semplici. Inizialmente vi sono già due utenti disponibili:

1. **Utente admin** tramite
    a. email: mario@rossi.com
    b. password: pass
2. **Utente comune** tramite
    a. email: mario@blu.com
    b. password: pass

Tempi di sviluppo

I tempi di sviluppo sono stati i seguenti:

- Progettazione modello e GUI: 6
- Codifica modello e GUI: 52 (incluso Debugging)
- Testing: 4
- **Totale**: 62

Qualche ora supplementare è stata spesa per migliorare graficamente la GUI. Ci sarebbero anche altre funzionalità interessanti quali la possibilità di segnare i link preferiti e di ordinarli in base alla popolarità. Sarebbe anche molto interessante aprire una QtWebView per mostrare le pagine dei bookmarks direttamente nell'app invece che passare al browser, ma non sarebbe stato possibile rimanere nel limite delle 60 ore e saper rispettare le deadlines è anch'essa una qualità importante in un progetto software.