Relazione progetto Programmazione a Oggetti

Carraro Riccardo, mat. 2042346
Toniolo Riccardo, mat. 2042332

Titolo: Bernie

Introduzione

In un mondo in cui la sicurezza digitale diventa sempre più rilevante e il numero di credenziali di accesso aumenta, possedere un sistema di gestione di credenziali e di memorizzazione sicura di queste, risulta sempre più necessario e utile. Il progetto, infatti, consiste in un applicativo di gestione e salvataggio dati (in particolar modo sensibili) oscurati da un'apposita logica di *encrypting* (Vigenère), protetto da un'unica password, che diventa anche la chiave di cifratura utilizzata. In questo modo infatti l'utente, mediante un'unica *Master-Password* è in grado di accedere a tutte le informazioni salvate e con la possibilità di poterle ricercare, visualizzare, modificare o rimuovere, grazie anche ad una interfaccia grafica appositamente strutturata e organizzata, al fine di garantire una quanto più semplice e intuitiva esperienza. L'applicativo permette il salvataggio di elementi quali account, carte di credito, portafogli di criptovalute, note e contatti.

Particolare attenzione va posta al container utilizzato per la gestione di tali oggetti: il progetto, infatti, implementa una versione rivisitata di un albero rosso-nero binario di ricerca, realizzato principalmente grazie alle conoscenze ottenute durante il corso di "Algoritmi e strutture dati", che permette di garantire un'alta efficienza in tutte le operazioni di visita, inserimento e eliminazione di elementi a patto di una più complessa implementazione rispetto ad altri sistemi di container.

L'idea di questo progetto nasce principalmente dalla comodità che un sistema del genere può offrire all'utente, e soprattutto dalla volontà di voler unire conoscenze e concetti interdisciplinari appresi in corsi di "Algoritmi e strutture dati" e "Cybersecurity" sulla gestione e sull'*encryption* dei dati.

Descrizione del modello

Il modello logico del progetto è caratterizzato dalle seguenti sezioni fondamentali, quali la gerarchia degli oggetti salvabili, il container utilizzato, il sistema di interfacciamento con il file system e l'interfaccia di gestione.
Il diagramma UML, omesso per motivi di dimensione, è reperibile al seguente link.
Mediante un accurato e quanto più completo sistema di commenti e documentazione all'interno del codice, è possibile dedurre il comportamento dei metodi delle varie classi implementate, le cui definizioni sono di fatto precedute da PRE e POST condizioni, che vanno ad esplicitare il comportamento e il risultato prodotto.

Gerarchia

Come previsto dalle richieste implementative, il progetto fa uso di una gerarchia composta da una classe astratta, SerializableObject, e cinque classi derivate concrete, rappresentative dei dati gestiti dall'applicativo. Questi oggetti, derivanti dalla classe base astratta SerializableObject, come suggerisce il nome stesso, implementano un metodo in grado di *serializzare* l'oggetto, fornendo in output una stringa composta da tutte le informazioni dell'oggetto stesso, in modo da permetterne il salvataggio su file (simile a CSV).
La classe SerializableObject, radice della gerarchia, dunque fornisce l'interfaccia base, comprendente i metodi essenziali, che ogni classe derivata dovrà successivamente implementare:

- ***virtual std::string serialize() const*** (virtuale puro) che restituisce una stringa contenente tutte le informazioni del file in formato simil CSV.
- ***virtual bool modify(****const SerializableObject**)*** (virtuale puro) che permette la modifica dell'oggetto mediante il passaggio di un puntatore ad un SerializableObject con il quale voglio modificare i dati dell'oggetto corrente.
- ***virtual void accept(****SerializableObjectsVisitor*, bool = false****) const*** (virtuale puro) che permette, sempre su base polimorfa, l'accettazione di un Visitor (secondo il *Visitor Pattern*) per la rappresentazione grafica dell'oggetto.

Decisa infatti la logica di salvataggio dei dati mediante un carattere separatore e un carattere di *escaping*, vengono inoltre definiti e implementati nella classe i metodi atti alla sanitizzazione e alla de-sanittizzazione delle stringhe generate dalla serialize() e della lettura da file, aggiungendo/rimuovendo eventuali escaping e garantendo che la stringa scritta/letta sia nel formato corretto:

- **static std::string sanitize(**const std::string&**)** che permette di sanitizzare la stringa generata dalla serialize() nel momento in cui i valori inseriti dall'utente nei campi degli oggetti da salvare, contengano esattamente i caratteri utilizzati per *escaping* o come sepratatore.
- ***static std::pair<bool, std::vector<std::string>> deSanitize(****const std::string &****)*** che permette, passata la stringa letta dal file in formato CSV, di restituire una flag di corretta lettura, e un vettore contenente le stringhe necessarie raffiguranti i campi dell'oggetto letto.

La gerarchia dunque, radicata in SerializableObject che possiede il campo *name* come identificativo degli oggetti, prevede le seguenti classi derivate concrete:

- Account, avente i campi relativi a email, username, password.
- CreditCard, avente i campi relativi a proprietario, numero, cvv e data di scadenza.
- Contact, avente i campi relativi a nome, cognome, data di nascita, numero di telefono e email.
- CryptoWallet, avente i campi relativi al nome della blockchain a cui fa riferimento e fino a 24 parole relative ai sistemi di sicurezza utilizzati dai moderni portafogli di criptovalute.
- Note, avente il campo relativo al testo.

Classe EncDec_File per la gestione del file system

Per l'interazione con il file system per il salvataggio e la gestione dei file per la persistenza dei dati è stata prevista una classe apposita. Essa racchiude dunque tutti i metodi necessari alla scrittura e lettura dei file, occupandosi inoltre, come suggerisce il nome stesso, della procedura di *encryption* e *decryption* delle informazioni. La classe dunque rappresenta l'interfacciamento dell'applicativo con il file system del dispositivo, permettendo una stesura di codice più lineare e pulita nel corpo del programma.

Interfacciamento con il modello

Al fine di rendere il codice il quanto più leggibile e mantenibile, la classe del container *RBBSTree* e la classe di accesso alle informazioni sul file *EncDec_File* sono state logicamente riunite in una classe *Vault* che rappresenta, all'interno del codice, il vero interfacciamento con il *core* dell'applicativo. Inoltre, mediante questa scelta è stato possibile ottenere considerevoli vantaggi implementativi, non solo garantendo una più rapida e migliore stesura del codice, ma anche segnando una netta e invalicabile separazione tra il modello logico e grafico, rendendo netta la loro distinzione a livello implementativo.

Polimorfismo

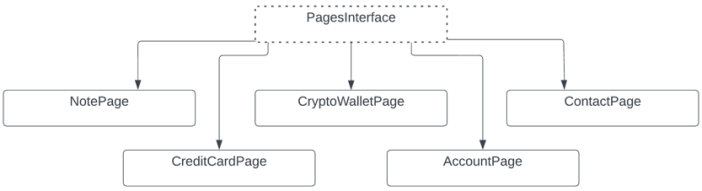
Il polimorfismo permea numerose aree del progetto, sia nella parte implementativa della gerarchia, sia nella parte grafica, mediante l'accurata progettazione di classi astratte sia per *Visitors* sia per le stesse pagine.

- **Gerarchia radicata in SerializableObject:**
  - Metodo **modify**: questo metodo permette, come suggerisce il nome, di modificare un oggetto, prendendo come argomento un puntatore a SerializableObject. Ogni classe concreta della gerarchia verificherà mediante un *dynamic_cast* se il puntatore passato è effettivamente del tipo dinamico corrente, e nel caso in cui lo fosse procede alla modifica, in caso contrario il metodo restituisce *false*, segnalando che la modifica non è avvenuta correttamente.
  - Metodo **serialize**: questo metodo si occupa della creazione della stringa contenente tutte le informazioni dell'oggetto da cui viene richiamata. Ogni classe concreta dovrà di conseguenza ridefinire il metodo al fine di produrre la rispettiva stringa per la memorizzazione su file.
  - Metodo **accept**: seguendo il *Visitor Design Pattern* per la realizzazione della parte grafica e dei widget rappresentativi degli oggetti, è stato definito un unico metodo in grado di accettare un visitor generico, prendendo come parametro un puntatore a SerializableObjectsVisitor, classe base astratta dei visitor utilizzati (polimorfismo aggiuntivo).
- **Visitor per la parte grafica:**
  - Data la natura profondamente diversa degli elementi gestiti dalla gerarchia, l'applicativo fa uso di un sistema di *visitor* in grado di poter creare i corretti widget per la rappresentazione degli elementi, predisponendo le rispettive pagine di creazione, visualizzazione e modifica.
  - All'interno della schermata principale del progetto, in cui sono visibili tutti gli elementi salvati, è estato reso possibile, sempre mediante il visitor, assegnare ad ogni elemento la rispettiva icona, migliorando sia dal punto di vista grafico sia dal punto di vista dell'utilizzo l'esperienza finale dell'utente.
  - L'approccio mediante *Visitor Design Pattern* contribuisce a mantenere la logica implementativa e la parte grafica separate, a patto della stesura di codice aggiuntivo.
- **Gerarchia dei visitor radicata in SerializableObjectsVisitor:**
  - Anche per i visitor è stata implementata una gerarchia, con radice in SerialiazableObjectVisitor in modo da sfruttare a pieno le potenzialità offerte dal polimorfismo, definendo un unico metodo **visit**, il quale, chiamato da un puntatore di tipo SerializableObjectsVisitor, permette di utilizzare il metodo definito nelle classi concrete derivate da SerializableObjectsVisitor. Mediante questo approccio, dunque, gli elementi della gerarchia possono avere un solo metodo *accept* che accetti un visitor generico, e lo stesso vale per i visitor implementanti il metodo *visit*.
- **Graphic User Interface (GUI):**
  - All'interno della pagina di selezione del tipo da creare, la logica implementativa prevede l'utilizzo di un segnale che permette alla pagina di gestire l'eventuale aggiunta di un nuovo elemento, predisponendo il corretto layout del widget da rappresentare. Essendo le pagine derivanti da

PagesInterface, esse implementano nuovamente i vari *slots* rendendo possibile la presenza di un solo sistema di *catching* dei segnali emessi, in quanto le pagine che dovranno gestirli si baseranno sulla classe generica *PagesInterface* e gestendo in modo dinamico i segnali ricevuti.

Descrizione del Graphic User Interface (GUI)

Per la parte grafica è stata prevista una gerarchia, riportata in forma minimale qui:



Analogamente agli elementi della gerarchia di SerializableObject, anche le relative classi per la rappresentazione grafica prevedono come radice *PagesInterface* e a seguire le classi concrete, una per tipo.
SerializableObjectsVisitor si specializza in due classi *VisitorManageItem* per la gestione degli oggetti (utilizzato in ambito di creazione, modifica e rimozione) e *VisitorListItem* per la loro rappresentazione all'interno della ListView nella schermata principale.
Tale approccio di inserire una classe generica SerializableOjectVisitor, permette dunque non solo di possedere un unico metodo *accept* per gli elementi della gerarchia, ma anche di racchiudere quella porzione di campi e metodi comuni che le classi *VisitorManageItem* e *VisitorListItem* altrimenti avrebbero avuto duplicate. (Storyboard grafica al seguente link)

Persistenza dei dati

Definita la natura degli elementi da gestire, si è resa obbligatoria una scelta di formato di salvataggio di questi. Si è optato dunque per il salvataggio su file di testo in formato CSV (*Comma-Separated values*). Mediante il metodo *serialize*, dunque, è possibile salvare all'interno del file la stringa relativa all'oggetto nel formato:

TYPE,*name*,*campo1*,campo2,campo3,...

Il campo "*TYPE*" indica il tipo dell'oggetto salvato, in quanto i successi vi campi, ad esclusione di "*name*" che funge da identificativo degli oggetti, variano a seconda del tipo di oggetto stesso.
(Nota: I metodi per la logica di salvataggio, quali serialize, deSerialize, sanitize e deSanitize, non sono importati esternamente ma implementati manualmente e interamente da noi).

Con l'obiettivo di dare più libertà possibile all'utente sull'utilizzo dell'applicativo, è stata garantita la possibilità di gestire molteplici archivi, ognuno protetto dalla rispettiva *Master-Password*.
Per verificare il sistema di salvataggio sono dunque forniti alcuni database predefiniti, di cui 2 pienamente funzionali e 2 danneggiati, al fine di dimostrare come il programma sia in grado di riconoscere file danneggiati/modificati dall'esterno. I file di prova sono i seguenti:

- prova1_correct (password: prova1);
- prova2_fail (password: prova2);
- prova3_correct (password: prova3);
- prova4_fail (password: prova4);

Come indica il nome stesso, i file con suffisso "*_correct*" sono funzionanti e popolati (2 elementi per ogni tipologia di oggetto della gerarchia), mentre i file che presentano il suffisso "*_fail*" sono corrotti, dunque provando ad aprirli, anche inserendo la corretta password, il programma segnalerà l'impossibilità di procedere, senza generare errori a run-time o crash. Di base, infatti, la logica con cui è implementato il programma, prevede l'accesso al database solamente mediante l'applicativo stesso; dunque, si riteneva necessaria l'implementazione di un sistema di salvaguardia da modifiche esterne.

Dato che all'apertura dei file di salvataggio dati si leggerebbero caratteri incomprensibili a seguito della tecnica di encryption adottata, è stata prevista nella MenuBar, all'interno della sezione "File", l'opzione "See decrypted database",

che visualizza in una text area l'intero contenuto del file decryptato. Questa opzione è chiaramente disponibile solamente una volta effettuato l'accesso al database a seguito dell'inserimento della password corretta.

Container realizzato: *Red-Black Binary Search Tree (RBBSTRee)*

Ponendo come principale obiettivo l'efficienza e le performance del progetto, si è optato per l'implementazione di un albero rosso-nero binario di ricerca. Infatti, questa struttura dati, a patto di una più complessa implementazione, permette di avere complessità ben definite per le operazioni di ricerca, inserimento e eliminazione, nell'ordine di O(log(n)). Un'ultreriore motivazione per la scelta di questa struttura, è stata la volontà di voler mostrare le informazioni in maniera ordinata al fine di migliorare l'esperienza dell'utente.

Una miglioria aggiuntiva implementata all'interno dell'albero è la presenza dei campi indicanti il predecessore e il successore per ogni nodo: questo, permette di utilizzare l'albero anche come eventuale lista doppiamente collegata, dando la possibilità dunque, partendo dal minimo (o dal massimo), di scorrere l'intero albero tramite appositi iteratori implementati. Essendo questi campi impostati alla creazione del nodo e modificati durante il suo inserimento, non vanno ad impattare sulla complessità totale dell'albero.

L'accesso, dunque, al minimo del sottoalbero destro (o al massimo del sottoalbero sinistro) diventa costante, impattando in modo positivo anche su operazioni (come l'eliminazione) che necessitano di queste informazioni. Inoltre, anche l'accesso al minimo e al massimo dell'intero albero diventa costante, in quanto questi valori sono memorizzati in appositi campi.

Una maggiore efficienza è inoltre dettata dall'utilizzo di metodi iterativi e non riscorsivi per le operazioni di scorrimento dell'albero, evitando dunque di sovraccaricare lo stack con eccessive chiamate a funzione.

Al fine di rendere l'albero quanto più generico possibile si è fatto uso di un template, in modo che la struttura diventasse indipendente dall'ambito d'uso.

Funzionalità implementate

Il carattere delle funzionalità implementate è suddivisibile in tre categorie:

- **Funzionali**: atte al fornire una quanto più soddisfacente esperienza all'utente, fornendo diverse possibilità di azione.
- **Estetiche**: atte alla rappresentazione semplice e intuitiva di ogni funzionalità implementata.
- **Documentative**: atte a fornire documentazione come il "manuale d'uso" (sezione *Manual* all'interno della MenuBar) o operazioni di testing per funzionalità del programma.

Funzionali:

- Gestione di cinque tipologie di dati;
- Conversione e salvataggio in formato CSV criptato tramite cifrario di Vigenère;
- Funzionalità di ricerca tramite nome identificativo;
- Funzionalità di filtraggio in base al tipo;
- Funzionalità di gestione di più archivi diversi;
- Possibilità di crearne nuovi;
- Possibilità di accedere ad archivi esistenti;
- Possibilità di effettuare logout dai vari archivi tramite il pulsante "Logout" (Dentro l'action "File") all'interno della MenuBar;
- Possibilità di rimuovere archivi esistenti;
- Alta efficienza e prestazioni nella struttura dati ad albero rosso-nero creata (RBBSTree);
- Salvataggio automatico dei dati;
- Per migliorare l'esperienza utente le modifiche agli archivi, di qualsiasi tipo, vengano automaticamente salvate, cosicché l'utente non debba attivamente preoccuparsi del salvataggio.

Grafiche:

- Pulsanti per filtrare per tipo gli elementi dell'archivio;
- Campo per la ricerca tramite nome nella parte alta della finestra per filtrare in real-time gli elementi dell'archivio tramite nome identificativo;
    - La ricerca avviene utilizzando le sotto sequenze di stringhe, per rendere più intelligente la ricerca per nome identificativo;
- Pulsante in alto a sinistra per rimuovere i filtri di ricerca e tornare alla visione ordinata degli elementi dell'archivio (pulsante "All");

- Richiesta all'utente, in caso di rimozione di un oggetto, di conferma della rimozione tramite una message box;
    - Questo per migliorare l'esperienza utente, e renderlo ulteriormente consapevole;

- Richiesta all'utente, in caso di rimozione di un archivio, di conferma della rimozione tramite una message box;
    - Sempre per migliorare la UX e rendere l'utente più consapevole;
- Realizzazione dell'intera applicazione in modalità Single Page Application, per migliorare la UX e rendere la navigazione all'interno dell'applicazione, più fluida e semplice;
- Possibilità di visualizzare i vari tipi di dato diversi, attraverso diverse interfacce per ogni tipo;
- Utilizzo di icone nei pulsanti;
- Creazione di un logo su misura da essere utilizzato come logo dell'applicazione;
    - La forma e la dimensione dell'icona sono appositamente curate per renderla quanto più simile alle altre applicazioni dei dispositivi Linux.
- Utilizzo di colori e stili grafici (QSS);
- Effetti grafici come cambio del colore al passaggio del mouse;
- Non si va ad utilizzare la solita finestra di dialogo per la selezione, creazione o rimozione di un file, ma si va ad utilizzare una schermata realizzata appositamente per rendere il più semplice possibile il processo di selezione, creazione o rimozione, senza doversi preoccupare di percorsi o directory;

Documentative:

- Presenza nella MenuBar, di un pulsante "Manual" (Dentro l'action "File") per accedere al manuale di utilizzo dell'applicazione.
    - Sebbene l'applicazione sia stata studiata per essere più intuitiva possibile, è stato aggiunto il manuale per poter chiarire ulteriormente, in caso di necessità, i dubbi dell'utente;
- Presenza nella menu bar, di un pulsante "*See decrypted database*" (Dentro l'action "*File*"), il quale permette (se si è effettuato l'accesso ad un archivio) di visualizzare il contenuto del file in maniera non cryptata;
- All'interno della cartella Testing, è possibile trovare gli unit test automatizzati, scritti dal mio collega (senza l'utilizzo di librerie esterne), utilizzati per provare in modo estensivo e mirato le parti più sensibili ad errori, nelle classi RBBSTree, EncDec_File e il Vault. Per andare ad eseguire questi test basta utilizzare i seguenti comandi in sequenza (I primi due comandi sono di compilazione, mentre il terzo è di esecuzione) (Bisogna prima ovviamente posizionarsi all'interno della cartella Testing con il terminale):
    - Per la compilazione mediante Qmake:
        - qmake
        - make
        - ./Bernie
    - Qualora di decidesse utilizzare CMake:
        - cmake CMakeLists.txt
        - make
        - ./Bernie
- Ogni classe appartenente al modello possiede per ogni metodo, della documentazione dettagliata sotto forma di commenti multi-riga, per renderne più chiaro l'utilizzo tramite una descrizione e PRE e POST condizioni ben definite.

(Nota: le funzionalità sovra elencate si riferiscono a quelle aggiunte rispetto alle richieste base della consegna)

Rendicontazione delle ore

Attività	Ore Previste	Ore Effettive
Studio e progettazione	8	11
Sviluppo del codice del modello	14.5	16
Studio del framework Qt	5	4
Sviluppo del codice della GUI	12.5	18
Test e debug	5	5
Stesura della relazione	5	5
totale	50	59

Il monte ore è stato superato per motivi prettamente implementativi vista la complessità del container utilizzato e le numerose funzionalità che volevamo fossero aggiunte per rendere l'applicativo quanto più completo e performante possibile.

- Lo sviluppo del container ha previsto ovviamente uno studio approfondito del programma di "Algoritmi e strutture dati" e la gestione di eventuali errori, unendo dunque una parte di studio teorica ad una parte di studio di implementazione pratica.
- L'interfaccia grafica ha ricevuto davvero molta attenzione proprio con l'obiettivo di presentare un prodotto quanto più completo, ricco e pratico possibile.

- La fase di testing, a seguito della implementazione di un sistema automatico scritto dal mio collega, ha reso possibile, almeno per la mia parte, ridurre considerevolmente il numero di ore necessarie per scrivere eventuali test. (Le ore contate fanno dunque riferimento sia a quelle precedenti alla creazione del sistema di testing, che a quelle successive, rese molto più rapide).

Funzionalità implementate

Essendo questo un progetto di gruppo è stato essenziale, a seguito di un'accurata progettazione, una suddivisione quanto più equa possibile dei lavori e delle funzionalità da dover implementare.

- **Gerarchia**:
  - Metodo *sanitize* per la classe SerializableObject (radice della gerarchia)
  - L'intera classe Note;
  - L'intera classe CreditCard;
  - L'intera classe Contatcs;
- **Classi aggiuntive:**
  - L'intera classe Date;
- **Container:**
  - Costruttore;
  - L'intera classe Nodo;
  - Metodo Insert e relativo metodo pubblico;
  - Metodo InsertFixUp;
  - Metodi di rotazione dell'albero:
    - RotateLeft;
    - RotateRight;
  - Metodo Search;
  - Metodo findMin;
  - Metodo toVector;
- **Vault:**
  - Metodo LoadToStorage;
  - Metodo Reset;
  - Metodo addSerializableObject;
  - Metodo deleteSerializableObject;
  - Metodo vectorize;
  - Metodo deleteDB;
- **EndDec_File:**
  - Metodo DecFromFile;
- **Testing:** solo alcune funzioni. Questa parte è stata prevalentemente gestita dal mio collega. Il testing per la parte grafica invece è stato effettuato in contemporanea.
- **GUI**:
  - Sviluppato con il collega in contemporanea sul medesimo file mediante la funzionalità "Code with me" offerta dall'IDE CLion (utilizzato per la scrittura del codice):
    - CreateDBPage;
    - DBSelectedPage;
    - DBSelectedToRemovePage;
    - HomePage;
    - LandingPage;
    - MainWindow;
    - SelectDBPage;
    - SelectDBToRemove;
    - TypeSelectionPage;
    - SerializableObjectsVisitor;
    - PagesInterface;
  - **Sviluppato da solo le pagine e classi:**
    - ContactPage;
    - NotePage;
    - CreditCardPage;
    - SerializableObjectsVisitor;
  - **Sviluppato da solo i seguenti componenti:**
    - DateComponent;

Compilazione ed esecuzione

Una volta posizionatisi nella cartella del progetto da terminale, per procedere alla compilazione e alla esecuzione dell'applicativo sarà necessario l'utilizzo dei seguenti comandi:

- QMake:
  - qmake
  - make
- CMake:
  - Cmake CMakeLists.txt
  - Make

(NOTA: è stata fornita la possibilità di compilazione mediante CMake a seguito dell'utilizzo dell'IDE CLion, il quale non dispone di QMake in modo nativo. Solo successivamente il QMake file è stato scritto)
L'esecuzione di questi programmi genererà un eseguibile "*Bernie"* (medesima cartella del file ".pro"), e per avviarlo sarà sufficiente la digitazione del comando "./Bernie".

Tutte le prove di compilazione ed esecuzione sono state effettuate sulla macchina virtuale fornita, in modo da garantire durante l'intero percorso del progetto, una completa compatibilità con il sistema di valutazione.

Note finali

Lo sviluppo di questo progetto, partendo dalla sua progettazione fino alla sua reale implementazione, è considerabile come una delle attività più utili e formative, soprattutto in ottica di organizzazione e gestione dei compiti. Infatti, la scelta di procedere in gruppo nella realizzazione è stata dovuta anche a questo. Il percorso di implementazione effettiva non ha riscontrato, fortunatamente, gravi difficoltà ed è stato anzi un valido mezzo per potersi mettere in gioco su qualcosa ideato e creato da zero. E' chiaro ovviamente che avendo utilizzato un nuovo framework quale Qt, spesso si è ricorsi a cercare informazioni sul web, in forum ufficiali, documentazioni e siti come StackOverflow.