LocknShop: documento integrativo finale

Autori: Adjetey Isabelle, Andreetti Luca, Carillo Vincenzo, Masa Biniam Abraha, Pinto Sabrina, Polzoni David

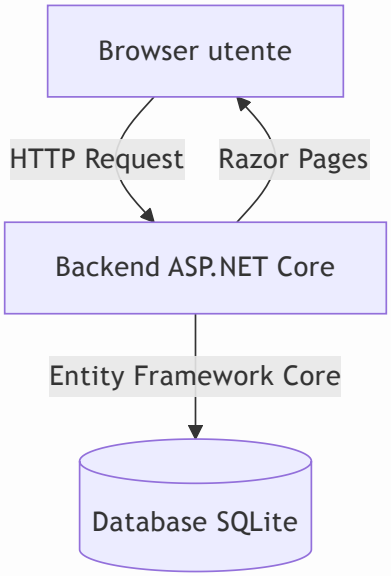
Team 1: CRIF Cyberguard

Panoramica generale

LocknShop è un'applicazione web e-commerce sviluppata con ASP.NET Core Razor Pages. L'applicazione segue un'architettura classica a tre livelli:

- **Frontend**: Razor Pages (.cshtml) renderizzate dal server
- **Backend**: ASP.NET Core (C#) che gestisce la logica applicativa e l'accesso ai dati
- **Database**: SQLite (file `locknshop-dev.db`) gestito tramite Entity Framework Core

Architettura



1. Frontend

- **Tecnologia**: ASP.NET Core Razor Pages (`.cshtml`)
- **Funzionalità**:
  - Visualizzazione prodotti, categorie, carrello, ordini
  - Form di registrazione, login, checkout

    - Layout responsive con Bootstrap e FontAwesome
- **Comunicazione**: il frontend non comunica mai direttamente con il database, ma solo tramite il backend

2. Backend

- **Tecnologia**: ASP.NET Core (C#)
- **Pattern**: Razor Pages con file code-behind (`.cshtml.cs`)
- **Funzionalità**:
    - Gestione autenticazione e autorizzazione (ASP.NET Core Identity)
    - Logica di business per prodotti, categorie, carrello, ordini
    - Accesso e manipolazione dati tramite Entity Framework Core

3. Database

- **Tecnologia**: SQLite (file locale `locknshop-dev.db`)
- **Gestione**: Entity Framework Core (migrazioni, modelli, query LINQ)
- **Tabelle principali**:
    - Utenti (gestiti da Identity)
    - Prodotti
    - Categorie
    - Carrello utente
    - Ordini

Funzionalità principali

1. Autenticazione e gestione utenti

- **Registrazione/Login**: gestiti tramite ASP.NET Core Identity
- **Ruoli**: almeno due ruoli (User, Admin)
- **Sessione**: gestita tramite cookie di autenticazione
- **Gestione interna**: nessun servizio esterno per l'autenticazione

2. Gestione prodotti e categorie

- **CRUD prodotti**: creazione, modifica, eliminazione e visualizzazione prodotti (per admin)
- **CRUD categorie**: gestione categorie prodotti (per admin)
- **Visualizzazione**: tutti gli utenti possono vedere prodotti e categorie

3. Carrello utente

- **Aggiunta/rimozione prodotti**: gli utenti possono aggiungere o rimuovere prodotti dal carrello
- **Visualizzazione carrello**: pagina dedicata con riepilogo prodotti, quantità, prezzi e sconti
- **Gestione carrello**: tutto gestito lato backend tramite helper e database

4. Checkout e ordini

- **Checkout**: form con dati personali, indirizzo e (simulazione) dati di pagamento
- **Salvataggio ordine**: gli ordini vengono salvati nel database locale
- **Svuotamento carrello**: dopo il checkout, il carrello viene svuotato
- **Storico ordini**: gli utenti possono vedere i propri ordini passati
- **Nessun pagamento reale**: dati di pagamento non vengono inviati a servizi esterni

5. Amministrazione

- **Gestione prodotti/categorie**: sezioni dedicate per l'amministratore
- **Ruoli**: possibilità di assegnare ruoli agli utenti (gestito da Identity)

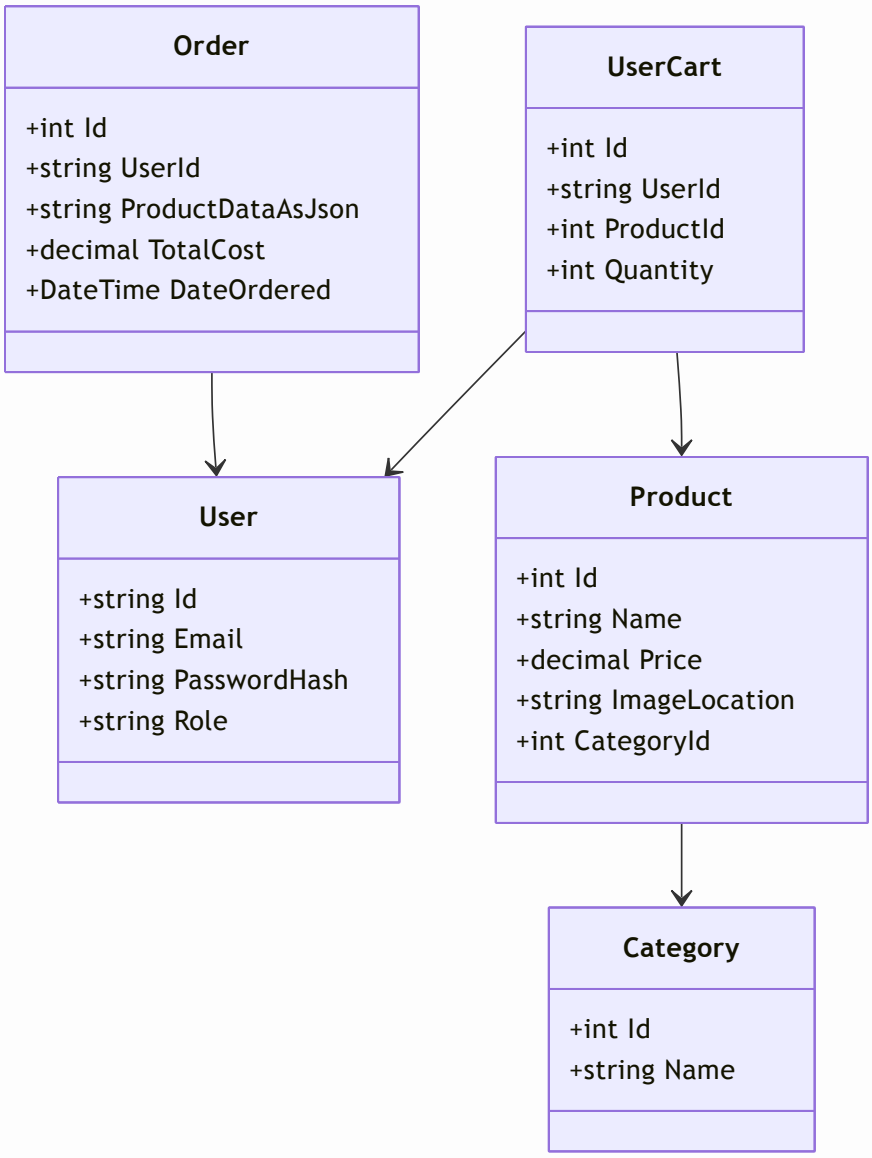
Sicurezza

- **Autenticazione**: solo utenti autenticati possono accedere a carrello, checkout e ordini
- **Autorizzazione**: solo admin possono gestire prodotti e categorie
- **Protezione dati**: password e dati sensibili gestiti tramite Identity

Estendibilità

- **Integrazione servizi esterni**: possibile aggiungere gateway di pagamento (Stripe, PayPal) o provider di autenticazione esterni
- **Scalabilità**: passaggio a database più robusti (SQL Server, PostgreSQL) possibile tramite Entity Framework

Schema delle principali entità



- Tutte le operazioni critiche (autenticazione, ordini, carrello) sono gestite dal backend
- Il frontend si limita a visualizzare dati e inviare richieste tramite form
- L'applicazione è pensata per essere facilmente estendibile e sicura

Motivazioni delle scelte architetturali e tecnologiche

1. ASP.NET Core Razor Pages

- **Motivazione**: Razor Pages offre un approccio semplice e strutturato per sviluppare applicazioni web con C#, separando chiaramente la logica di presentazione (frontend) dalla logica applicativa (backend). È ideale per progetti dove la maggior parte delle interazioni avviene tramite form e pagine server-side, garantendo sicurezza e facilità di manutenzione.
- **Vantaggi**:

- Sviluppo rapido e produttivo
- Integrazione nativa con Identity e Entity Framework
- Sicurezza elevata grazie alla gestione server-side

2. Entity Framework Core e database SQLite

- **Motivazione**: Entity Framework Core permette di gestire il database in modo dichiarativo tramite modelli C#, semplificando le operazioni CRUD e la gestione delle migrazioni. SQLite è stato scelto per la sua leggerezza e facilità di setup, ideale per ambienti di sviluppo, demo o piccoli progetti.
- **Vantaggi**:
  - Sviluppo e test locali senza necessità di server DB esterni
  - Facilità di migrazione verso database più robusti in futuro
  - Gestione automatica delle relazioni e integrità dei dati

3. ASP.NET Core Identity per l'autenticazione

- **Motivazione**: Identity è il sistema standard e sicuro per la gestione di utenti, ruoli e autenticazione in ambiente .NET. Permette di implementare rapidamente login, registrazione, gestione ruoli e protezione delle pagine.
- **Vantaggi**:
  - Password gestite in modo sicuro (hashing, salting, lockout, ecc.)
  - Estendibilità per provider esterni (Google, Facebook, ecc.)
  - Integrazione nativa con il framework

4. Gestione interna di carrello e ordini

- **Motivazione**: gestire carrello e ordini internamente permette di avere pieno controllo sulla logica di business, senza dipendere da servizi esterni. Questo garantisce maggiore sicurezza e personalizzazione.
- **Vantaggi**:
  - Nessuna dipendenza da terze parti
  - Possibilità di personalizzare la logica di checkout e gestione ordini
  - Maggiore controllo sui dati degli utenti

5. Nessuna integrazione con gateway di pagamento (per ora)

- **Motivazione**: in fase iniziale o per progetti demo, può essere preferibile simulare il pagamento senza integrare gateway reali, per semplificare lo sviluppo e i test.
- **Vantaggi**:
  - Sviluppo più rapido

- Nessuna gestione di dati sensibili di pagamento
- Possibilità di aggiungere facilmente l'integrazione in futuro

6. Utilizzo di ruoli (user, admin)

- **Motivazione**: separare i permessi tra utenti normali e amministratori è fondamentale per la sicurezza e la gestione delle funzionalità avanzate (es. CRUD prodotti/categorie).
- **Vantaggi**:
  - Maggiore sicurezza
  - Gestione chiara delle responsabilità
  - Facilità di estensione per altri ruoli

7. Layout responsive e librerie frontend

- **Motivazione**: l'uso di Bootstrap e FontAwesome garantisce un'interfaccia moderna, responsive e facilmente personalizzabile, migliorando l'esperienza utente.
- **Vantaggi**:
  - UI accattivante e usabile su tutti i dispositivi
  - Sviluppo frontend più veloce grazie a componenti pronti all'uso

Motivazioni contestuali e confronto con architettura avanzata

Scelte attuali: motivazioni di tempo e semplicità

Le scelte architetturali adottate in questa versione di LocknShop sono state guidate principalmente da:

- **Vincoli di tempo**: la necessità di consegnare un prodotto funzionante in tempi rapidi ha portato a privilegiare tecnologie e pattern consolidati, facilmente integrabili e con una curva di apprendimento ridotta.
- **Semplicità di deploy e sviluppo**: l'uso di SQLite, Entity Framework Core e Razor Pages permette di sviluppare, testare e distribuire l'applicazione senza infrastrutture cloud complesse o costi aggiuntivi.
- **Focus sulla concretezza**: l'obiettivo era dimostrare competenze tecniche e organizzative con un prodotto reale, funzionante e facilmente estendibile.

Differenze rispetto all'architettura progettata

L'architettura attuale si differenzia dalla soluzione cloud progettata in diversi punti chiave:

Aspetto	Soluzione attuale (Local)	Soluzione avanzata (Cloud)
Frontend	Razor Pages server-side	Frontend statico (HTML/CSS/JS), SPA
Backend	ASP.NET Core Razor Pages	ASP.NET Core Web API (RESTful)
Comunicazione	HTML generato server-side	Fetch API/REST tra frontend e backend
Autenticazione	Cookie-based, ASP.NET Identity	JWT, Microsoft Entra ID, MFA
Database	SQLite locale	Azure SQL Database
Sicurezza	Identity, ruoli, protezione server	Rate limiting, logging avanzato, CSP, Key Vault
Deploy	Locale o server semplice	Azure App Service, Static Web App
Gestione segreti	Config locale	Azure Key Vault
Monitoraggio	Limitato	Azure Monitor, Application Insights
Scalabilità	Limitata	Scalabilità cloud
Gestione asset statici	wwwroot/ locale	Azure Storage Account

- **Tempo e risorse**: l'adozione di servizi Azure, autenticazione avanzata (JWT, MFA), logging e monitoraggio cloud richiede più tempo, conoscenze specifiche e costi di gestione.
- **Obiettivo dimostrativo**: la soluzione attuale è pensata per essere facilmente eseguibile da chiunque, senza prerequisiti cloud o costi aggiuntivi.
- **Espandibilità futura**: l'architettura è stata progettata per poter essere evoluta facilmente verso una soluzione cloud/enterprise, aggiungendo servizi come Azure, autenticazione esterna, logging avanzato, ecc.

Le scelte attuali sono state fatte per:

- Massimizzare la produttività e la consegna rapida
- Ridurre la complessità iniziale
- Consentire una facile evoluzione futura verso architetture più avanzate e sicure

Confronto con Application Security Policy avanzata

L'architettura attuale di LocknShop, pensata per semplicità e rapidità di sviluppo, si differenzia da una soluzione che implementa una **Application Security Policy (ASP)** avanzata come quella descritta nei seguenti punti chiave:

1. Esposizione accidentale delle API (path traversal, IDOR)

- **Soluzione attuale**: le pagine Razor e le azioni backend sono protette da autenticazione e ruoli tramite ASP.NET Identity, ma non esistono endpoint API REST pubblici.
- **Soluzione avanzata**: ogni endpoint API è protetto da JWT, con controlli di autorizzazione granulari e CORS ristretto.
- **Gap**: manca la protezione tramite JWT e la restrizione CORS, perché la comunicazione è server-side.

2. Gestione delle credenziali e segreti

- **Soluzione attuale**: le chiavi e le stringhe di connessione sono gestite tramite file di configurazione locale.
- **Soluzione avanzata**: uso di Azure Key Vault, esclusione file sensibili dal versionamento, logging accessi anomali.
- **Gap**: non viene usato un vault centralizzato; la soluzione attuale è sufficiente per sviluppo locale ma non per produzione.

3. Rate Limiting e protezione brute-force

- **Soluzione attuale**: ASP.NET Identity gestisce lockout dopo tentativi falliti di login, ma non c'è un vero middleware di rate limiting sulle richieste.
- **Soluzione avanzata**: middleware dedicato, alert automatici, blocchi temporanei.
- **Gap**: rate limiting e alerting non implementati.

4. Logging e monitoraggio

- **Soluzione attuale**: logging di base tramite strumenti .NET, senza centralizzazione o alert.
- **Soluzione avanzata**: logging centralizzato, log sanificati, alert su eventi sospetti tramite Azure Monitor.
- **Gap**: logging avanzato e alerting non presenti.

5. Errori di configurazione e ambienti

- **Soluzione attuale**: separazione tra sviluppo e produzione tramite variabili d'ambiente, ma senza automazione cloud.
- **Soluzione avanzata**: ambienti separati, accessi limitati, debug disattivato in produzione.
- **Gap**: la separazione è manuale e locale, non cloud-native.

6. Scalabilità

- **Soluzione attuale**: scalabilità limitata all'ambiente locale/server singolo.
- **Soluzione avanzata**: scalabilità orizzontale su Azure, monitoraggio risorse, upgrade automatico.
- **Gap**: non predisposto per scalabilità cloud.

7. Protezione XSS e sicurezza client-side

- **Soluzione attuale**: Razor Pages esegue escaping automatico, ma non sono implementate CSP avanzate.
- **Soluzione avanzata**: CSP, escaping sistematico, binding sicuro.
- **Gap**: CSP e policy avanzate non presenti.

8. Gestione avanzata della sessione

- **Soluzione attuale**: sessione gestita via cookie, logout effettivo, ma nessun refresh token o blacklist JWT.
- **Soluzione avanzata**: JWT a scadenza breve, refresh token, blacklist token.
- **Gap**: mancano refresh token e gestione avanzata della sessione.

9. Upload sicuro di file

- **Soluzione attuale**: non sono previsti upload di file nella versione attuale.
- **Soluzione avanzata**: validazione MIME, scansione antivirus, quarantena file.
- **Gap**: upload non implementato, ma da prevedere in futuro.

10. Autenticazione forte (MFA)

- **Soluzione attuale**: solo email e password, senza MFA.
- **Soluzione avanzata**: MFA tramite provider esterni (es. Entra ID), OTP, verifica email.
- **Gap**: MFA non implementato, ma previsto come estensione futura.

Sintesi e prospettive

- L'architettura attuale copre i requisiti minimi di sicurezza per un ambiente di sviluppo e test, ma **non implementa molte delle mitigazioni avanzate** previste da una Application

Security Policy enterprise/cloud.

- Queste scelte sono state fatte per motivi di tempo, semplicità e facilità di deploy, ma la struttura del progetto è pensata per poter **evolvere facilmente** verso standard di sicurezza più elevati, integrando servizi cloud, policy avanzate e strumenti di monitoraggio e protezione.

Implementare le soluzioni avanzate (esempi di codice)

Di seguito una panoramica su come evolvere LocknShop verso una soluzione enterprise/cloud-ready, con esempi pratici di codice e spiegazioni:

1. API protette da JWT e CORS

Obiettivo: esporre solo API protette da token JWT e accettare richieste solo dal frontend autorizzato.

Codice:

```
// In Program.cs o Startup.cs
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;

services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options => {
        options.TokenValidationParameters = new TokenValidationParameters {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = "https://locknshop.azuredomain.com",
            ValidAudience = "https://locknshop.azuredomain.com",
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(C
        };
    });

app.UseCors(policy => policy
    .WithOrigins("https://frontend.azuredomain.com")
    .AllowAnyHeader()
    .AllowAnyMethod());

app.UseAuthentication();
app.UseAuthorization();
```

Spiegazione:

- Si configura JWT come schema di autenticazione.
- Si limita CORS al solo dominio frontend.
- Ogni endpoint API richiede un token JWT valido.

2. Gestione sicura delle chiavi (Key Vault)

Obiettivo: non salvare segreti nel codice o in file di configurazione.

Codice:

```
// In Program.cs
using Azure.Identity;

builder.Configuration.AddAzureKeyVault(
    new Uri(builder.Configuration["KeyVaultUri"]),
    new DefaultAzureCredential());
```

Spiegazione:

- I segreti (es. chiave JWT, stringhe di connessione) vengono letti da Azure Key Vault.
- Nessun segreto è presente nel repository o nei file di configurazione locali.

3. Rate Limiting e protezione brute-force

Obiettivo: limitare il numero di richieste per utente/IP e bloccare tentativi di login ripetuti.

Codice:

```
// In Program.cs
services.AddInMemoryRateLimiting();
services.Configure<IpRateLimitOptions>(options => {
    options.GeneralRules = new List<RateLimitRule> {
        new RateLimitRule {
            Endpoint = "*",
            Limit = 100,
            Period = "1m"
        }
    };
});
app.UseIpRateLimiting();
```

Spiegazione:

- Si limita il numero di richieste per IP.
- Si possono aggiungere regole specifiche per endpoint sensibili (es. login).

---

4. Logging avanzato e monitoraggio

Obiettivo: centralizzare i log, sanificare i dati sensibili, impostare alert su eventi critici.

Codice:

```
// In Program.cs
builder.Services.AddApplicationInsightsTelemetry();

// Esempio con Serilog
Log.Logger = new LoggerConfiguration()
    .WriteTo.Console()
    .WriteTo.AzureAnalytics(workspaceId, authenticationId)
    .CreateLogger();
```

Spiegazione:

- Application Insights raccoglie metriche e log.
- Serilog permette log strutturati e invio a sistemi esterni.
- I log devono essere sanificati (mai scrivere email, token, password).

5. Ambienti separati e configurazione sicura

Obiettivo: separare dev, test, prod e usare configurazioni sicure.

Codice:

```
// In Program.cs
if (app.Environment.IsProduction())
{
    // Disabilita stacktrace dettagliati
    app.UseExceptionHandler("/Error");
    // Usa solo segreti da Key Vault
}
else
{
    // Ambiente di sviluppo
    app.UseDeveloperExceptionPage();
}
```

Spiegazione:

- In produzione, niente stacktrace dettagliati o debug.
- Configurazioni e segreti solo tramite Key Vault/variabili ambiente.

6. Scalabilità cloud

Obiettivo: deploy su Azure App Service (backend), Azure Static Web App (frontend), Azure SQL.

Codice:

- **Deploy**: usare pipeline CI/CD (GitHub Actions, Azure DevOps) per pubblicare su Azure.
- **Configurazione**: abilitare scaling automatico su App Service e database.

Spiegazione:

- L’infrastruttura cloud permette di scalare orizzontalmente e verticalmente.
- Asset statici (immagini, file) su Azure Storage Account.

7. Protezione XSS e CSP

Obiettivo: prevenire l’iniezione di script malevoli.

Codice:

```
// In Program.cs
app.Use(async (context, next) => {
    context.Response.Headers.Add("Content-Security-Policy", "default-src 'self';
    await next();
});
```

Spiegazione:

- L’header CSP limita le risorse caricabili dal browser.
- Tutti i dati visualizzati devono essere sanificati lato backend e frontend.

---

8. Gestione avanzata della sessione (JWT, refresh token, blacklist)

Obiettivo: permettere logout effettivo e gestione sicura delle sessioni.

Codice:

```
// Generazione JWT breve
var token = new JwtSecurityToken(
    expires: DateTime.UtcNow.AddMinutes(15),
    // ...
);

// Gestione refresh token
// Salvare refresh token sicuro in DB e associarlo all’utente

// Blacklist token
// Mantenere una lista di token revocati e verificarli ad ogni richiesta
```

Spiegazione:

- JWT a scadenza breve, refresh token per rinnovo.
- Blacklist per revocare token compromessi.

---

9. Upload sicuro di file

Obiettivo: prevenire upload di file pericolosi.

Codice:

```
// Validazione lato backend
if (!allowedMimeTypes.Contains(file.ContentType))
    return BadRequest("Tipo file non consentito");

// Scansione antivirus (es. CrowdStrike)
// ... chiamata a servizio antivirus ...

// Salvataggio su Azure Storage in quarantena
```

Spiegazione:

- Validare estensione e MIME type.
- Scansionare file caricati.
- Usare storage sicuro e accesso limitato.

10. Autenticazione forte (MFA)

Obiettivo: richiedere un secondo fattore di autenticazione.

Codice:

```
// In Identity
services.Configure<IdentityOptions>(options => {
    options.SignIn.RequireConfirmedEmail = true;
    options.Tokens.AuthenticatorTokenProvider = TokenOptions.DefaultAuthenticato
});
// Abilitare MFA tramite provider esterni (es. Entra ID)
```

Spiegazione:

- MFA può essere abilitato tramite email, app Authenticator, SMS o provider esterni.
- Richiedere MFA per login o operazioni sensibili.

Questi interventi, con codice di esempio e spiegazione, permetterebbero a LocknShop di raggiungere un livello di sicurezza, scalabilità e affidabilità conforme agli standard enterprise e cloud-native, mantenendo la possibilità di evolvere e integrare nuove funzionalità in futuro.

Suite di test

Per garantire la sicurezza, l'affidabilità e la qualità di una piattaforma e-commerce come LocknShop, soprattutto in una versione avanzata e cloud-native, sarebbe fondamentale sviluppare una suite di test completa e strutturata. Di seguito vengono descritte le

tipologie di test che avremmo dovuto implementare e le motivazioni legate alle feature di sicurezza dell'applicazione.

Tipologie di test

- **Unit test**
  - Verifica delle singole funzioni e metodi (es. validazione input, calcolo prezzi, logica di business).
  - Fondamentali per garantire la correttezza delle funzioni critiche (es. calcolo totale carrello, gestione sconti).

- **Integration test**
  - Testare l'interazione tra componenti (es. API, database, servizi esterni come Key Vault o provider MFA).
  - Essenziali per verificare che l'autenticazione JWT, la gestione dei ruoli e la comunicazione con servizi cloud funzionino correttamente.

- **End-to-End (E2E) test**
  - Simulano i flussi utente reali (es. registrazione, login, acquisto, checkout, logout).
  - Utili per validare la user experience e la sicurezza dei flussi principali.

- **Test di sicurezza**
  - **Autenticazione e autorizzazione**: test su login, MFA, gestione token JWT, revoca e refresh token.
  - **Rate limiting**: test per verificare che gli endpoint sensibili blocchino richieste eccessive o brute-force.
  - **Gestione errori e logging**: test per assicurarsi che i log non espongano dati sensibili e che gli errori non rivelino dettagli tecnici.
  - **XSS e input validation**: test automatici e manuali per verificare la sanificazione degli input e la protezione contro attacchi XSS.
  - **Upload sicuro**: test su validazione MIME type, estensione file, e scansione antivirus.
  - **CSP e header di sicurezza**: test per assicurarsi che le risposte HTTP includano header di sicurezza corretti.

Motivazioni delle scelte di test in base alle feature di sicurezza:

- **MFA e autenticazione avanzata**: test specifici per garantire che il secondo fattore sia sempre richiesto dove previsto e che non sia aggirabile.

- **Rate limiting**: test automatici per simulare attacchi brute-force e verificare che l'utente/IP venga bloccato dopo N tentativi.
- **Gestione errori**: test per assicurarsi che in produzione non vengano mai restituiti stacktrace o messaggi dettagliati.
- **XSS e input validation**: test automatici (es. con strumenti come OWASP ZAP) per iniettare script e verificare la robustezza della sanificazione.
- **Upload**: test per caricare file di vari tipi e dimensioni, inclusi file malevoli, per verificare la validazione e la quarantena.
- **Logging**: test per assicurarsi che i log non contengano dati sensibili e che gli alert vengano generati su eventi sospetti.
- **CSP e header**: test automatici per verificare la presenza e la correttezza degli header di sicurezza.

Strumenti di test:

- **MSTest**: framework di test Microsoft per .NET, integrato in Visual Studio. Permette di scrivere e gestire unit e integration test facilmente (o **xUnit** alternativamente).
- **Visual Studio test explorer**: interfaccia grafica in Visual Studio per eseguire, monitorare e analizzare i risultati dei test (MSTest, xUnit, NUnit).
- **Azure DevOps test plans**: suite cloud per la gestione di test manuali, automatizzati, tracciamento bug e reportistica. Permetterebbe di integrare i test nelle pipeline CI/CD.
- **Playwright for .NET**: strumento Microsoft per test end-to-end cross-browser, ideale per testare l'interfaccia utente e i flussi utente reali.
- **Application Insights**: oltre al monitoraggio, può essere usato per testare la salute e le performance dell'applicazione in produzione (availability tests, alert su errori).

Collocazione nel ciclo di vita dei test di LocknShop:

- **Unit/Integration test**: MSTest, xUnit, Test Explorer
- **E2E test**: Playwright for .NET
- **Gestione e report**: Azure DevOps Test Plans
- **Monitoraggio e alert**: Application Insights

Questa suite di test, se implementata, garantirebbe che tutte le feature di sicurezza avanzata siano effettivamente efficaci e che l'applicazione sia pronta per ambienti di produzione ad alta affidabilità e sicurezza.

La soluzione attuale di LocknShop rappresenta un **piano B** rispetto all'idea originaria (piano A) di sviluppare un'applicazione e-commerce complessa, cloud-native, con architettura avanzata e tutte le feature di sicurezza e scalabilità descritte nei documenti precedenti.

- **Vincoli di tempo**: il tempo a disposizione per la consegna del progetto era limitato e non avrebbe permesso di implementare tutte le componenti cloud, di sicurezza avanzata e di test automatizzati previsti dal piano A.
- **Risorse limitate**: il team ha dovuto lavorare con risorse tecniche e infrastrutturali ridotte, privilegiando strumenti e tecnologie facilmente accessibili e gestibili localmente.
- **Necessità di consegna**: era fondamentale consegnare un prodotto funzionante, testabile e dimostrabile, anche a costo di rinunciare temporaneamente a feature enterprise/cloud.
- **Semplicità di deploy e testabilità locale**: la scelta di una soluzione monolitica, con database locale e deploy semplificato, ha permesso di velocizzare lo sviluppo e facilitare la verifica delle funzionalità.
- **Evoluzione futura**: l'architettura è stata comunque progettata per poter essere evoluta agilmente verso il piano A, integrando servizi cloud, sicurezza avanzata e automazione dei test.

Questa soluzione è stata adottata per garantire la consegna di un prodotto concreto e funzionante nei tempi richiesti, senza precludere la possibilità di evolvere verso una piattaforma cloud-native e altamente sicura in futuro.

Reference codice sviluppato

Il codice sorgente sviluppato per LocknShop è disponibile al seguente link:

https://github.com/enzcarix7/Cyberguard2025