Prova tecnica Photosi

Sistema di gestione ordini di acquisto “PhotoSi.OrderingSystem”

Autore: Marco Ciaschini (mciasco@libero.it)

Sommario

[Premessa 1](#_Toc162224240)

[Sorgenti GIT 1](#_Toc162224241)

[Note rapide di avvio 1](#_Toc162224242)

[Avvio servizi 1](#_Toc162224243)

[Avvio Test 3](#_Toc162224244)

[Descrizione del sistema 3](#_Toc162224245)

[Schematizzazione del sistema 3](#_Toc162224246)

[Descrizione del progetto 4](#_Toc162224247)

[Databases 6](#_Toc162224248)

[Descrizione delle singole tipologie di progetto 7](#_Toc162224249)

[Flusso tipico di una request/response 9](#_Toc162224250)

[Note sui test 9](#_Toc162224251)

# Premessa

I requisiti richiedono l’implementazione di un sistema di gestione ordini basilare, costituito da alcuni componenti indipendenti (servizi) che gestiscano i vari aspetti: ordini, prodotti, utenti e indirizzi di consegna. Inoltre, è richiesto un servizio aggiuntivo di “backoffice” per gestire le singole risorse dati.

# Sorgenti GIT

I sorgenti dell’intero sistema, compresa questa guida (oltre che in allegato) sono disponibili su repository git pubblico sul mio spazio GitHub qui: <https://github.com/mciasco/PhotoSi.OrderingSystem.git>

# Note rapide di avvio

Prima di addentrarsi nella descrizione esaustiva del sistema, ecco le istruzioni di avvio da me utilizzate. Per lo sviluppo ed i test ho usato Visual Studio 2022, quindi faccio riferimento a questo IDE.

### Avvio servizi

Per avviare i progetti principali, cioè tutti quelli definiti come “NomeServizio.WebApi” è sufficiente configurare i progetti di avvio multiplo della solution: Project -> Configure Setup Project. Nella dialog che si apre ho selezionato il radio button “Multiple startup project” e impostato con la voce “Start” tutti i progetti WebApi (AddressBook.WebApi, BackOffice.WebApi, Products.WebApi, Orders.WebApi, Users.WebApi).

A questo punto, l’avvio in debug di tutti i progetti è eseguibile dal classico menu “Start” su cui ho scelto sempre di eseguire i progetti nel web browser (Chrome, nel mio caso).

Tutti i progetti WebApi sono configurati per esporre una view SwaggerUi che ne faciliterà l’interazione, mostrando tutti metodi esposti dai controller con i necessari paramentri.

Avviando i progetti in questo modo, tutti i progetti dovrebbero partire con la configurazione di lancio specificata nel proprio file dentro Properties/launchSettings.json. La configurazione di lancio eseguita in questo modo è quella che usa la chiave “IIS Express” con la configurazione con chiave “iisSettings”. Tutti i progetti al momento partono quindi in localhost sulla porta indicata dal campo “sslPort”.

Quindi gli url di avvio (una pagina del browser per ogni servizio) saranno:

* AddressBook.WebApi:
  + url api: https://localhost:7131/api
  + url swagger: https://localhost:7131/swagger/index.html
* BackOffice.WebApi:
  + url api: https://localhost:7141/api
  + url swagger: https://localhost:7141/swagger/index.html
* Products.WebApi:
  + url api: https://localhost:7098/api
  + url swagger: https://localhost:7098/swagger/index.html
* Orders.WebApi:
  + url api: https://localhost:7109/api
  + url swagger: https://localhost:7109/swagger/index.html
* Users.WebApi:
  + url api: https://localhost:7159/api
  + url swagger: https://localhost:7159/swagger/index.html

E’ importante sincerarsi che questi url siano quelli effettivamente di avvio, perché alcuni servizi fra di loro si chiamano e fanno riferimento a questi url per poterlo fare (quelli delle api, di fatto).

I servizi che si chiamano fra di loro, specificano questi url nel proprio appsettings.json nella sezione “Clients”, al cui interno c’è una voce di configurazione per ogni client di cui si necessita.

Un esempio è certamente il servizio BackOffice.WebApi che, dovendo contattare tutti gli altri servizi, specifica praticamente tutti gli url. Nel suo file appsettings.json ci sono ad esempio questi dati:

"Clients": {

"ProductsServiceSettings": {

"BaseUrl": "https://localhost:7098/"

},

"AddressBookServiceSettings": {

"BaseUrl": "https://localhost:7131/"

},

"UsersServiceSettings": {

"BaseUrl": "https://localhost:7159/"

}

}

### Avvio Test

Per avviare i test invece, ho usato il Test Explorer integrato in Visual Studio 2022 che permette di operare con i test creati tramite xUnit. E’ sufficiente quindi aprire il menu Test 🡪 Test Explorer per visualizzare tutti i test ed eseguirli tutti o singolarmente.

# Descrizione del sistema

Il sistema denominato PhotoSi.OrderingSystem è costituito da componenti implementati come servizi standalone WebApi in NET 6.0 su framework Asp Net Core, linguaggio C#. La gestione dei dati è implementata tramite database relazionali, costituiti da file SqlLite. L’interazione fra i servizi e i database è gestita con Entity Framework Core.

I servizi implementati sono i seguenti:

* **Orders**: servizio di gestione ordini che permette la creazione, il recupero e l’eliminazione di ordini di acquisto da parte di utenti finali del sistema (clienti), che scelgono uno o piò prodotti per ciascun ordine e un indirizzo di consegna a loro associato.
* **Products**: servizio che implementa un catalogo prodotti, con relativo magazzino che ne fornisce la disponibilità in fase di creazione ordini
* **Users**: servizio di anagrafica degli utenti finali (clienti), intesi come clienti del sistema
* **AddressBook**: servizio che espone l’elenco di indirizzi di consegna associati ad utenti del sistema, che possono essere scelti in fase di creazione ordini.
* **BackOffice**: servizio che rappresenta un applicativo semplificato per la gestione delle singole risorse utilizzate nel sistema. Si suppone venga utilizzato da utenti amministrativi o di assistenza e non da utenti finali (clienti)

In accordo con una architettura a microservizi, ogni servizio suddetto gestisce in autonomia i propri dati applicativi con un database relazionale SqlLite dedicato.

NB: E’ stata introdotta una sola eccezione, riguardante i serivizi Users e AddressBook che simulano una parte del sistema che si può supporre sia stata implementata in precedenza (legacy) e che quindi utilizza uno stesso database. Tuttavia, il sistema mostra come, anche usando un unico database, si possa operare in maniera indipendente tramite due microservizi dedicati che separano la gestione dei dati a livello applicativo, anche se risiedono sullo stesso database.

# Schematizzazione del sistema

Di seguito una schematizzazione informale dei componenti del sistema, che mostra le interdipendenze applicative fra i servizi e i relativi database.

Immagine che contiene diagramma, testo, Piano, Disegno tecnico

Descrizione generata automaticamente

# Descrizione del progetto

Il sistema è stato implementato con Visual Studio 2022. E’ quindi costituito da un’unica solution internamente strutturata in maniera da riflettere i singoli componenti del sistema. La solution è quindi suddivisa in 5 solution folders, una per ciascun applicativo, più una ulteriore solution folder che contiene progetti di carattere condiviso fra gli altri. Di seguito una immagine complessiva della solution.

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

In generale, ogni servizio è implementato seguendo il pattern architetturale denominato Clean Architecture (<https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures#clean-architecture>). Questo pattern prevede la presenza di (almeno) 3 diversi progetti: un progetto di avvio principale, in cui generalmente risiede la logica presentazionale e applicativa, un progetto di contratti che contiene le astrazioni e il dominio applicativo e infine un progetto di infrastruttura, che contiene le implementazioni concrete e specifiche delle astrazioni presenti nei contratti, verticalizzandone le implementazioni con l’uso di specifiche tecnologie.

Questa architettura ha l’obiettivo di definire in maniera chiara e pulita le responsabilità dei singoli layer e singoli componenti del sistema. Seguendo questi dettami, ogni servizio del sistema è quindi stato implementato usando questi 3 progetti:

* **[NomeServzio].WebApi**: rappresenta la web api avviabile e contiene quindi le logiche più esterne e presentazionali (es. controllers, web pages..). Inoltre, ma solo per semplificazione di questo progetto, è stato inserito qui lo strato applicativo, che rappresenta la logica business. L’indicazione architetturale vorrebbe infatti, che queste classi fossero a loro volta separate in un progetto standalone.
* **[NomeServizio].Contracts**: contiene le interfacce e le classi astratte comuni altri altri progetti del servizio e anche gli oggetti di dominio (entità).
* **[NomeServizio].Infrastructure**: contiene le implementazioni concrete delle interfacce presenti nel progetto Contracts. Qui ci sono quindi le classi concrete che si interfacciano con il DB ed eventuali sistemi esterni via http o altre tecnologie di comunicazione (ad esempio sarebbero qui eventuali classi che interagiscono con message bus).
* **[NomeServizio].Tests**: progetto aggiuntivo che contiene i soli unit test del servizio.

Di seguito un tipico schema che rappresenta le interdipendenze fra i progetti che costituiscono un singolo servizio:

Immagine che contiene testo, schermata, Rettangolo, linea

Descrizione generata automaticamente

# Databases

Come detto, lo strato database è stato implementato con singoli file SqlLite. Come richiesto e come indicato dalle best practice, i microservizi accedono ad un proprio database standalone. Ho voluto però mostrato un tipico esempio “legacy” in cui si può immaginare che il sistema sia stato costruito come evoluzione di un applicativo monolitico precedente. Per questo motivo ho immaginato l’esistenza di un database unico che unisse i dati utente ai loro indirizzi di spedizione.

I database, oltre che tramite i servizi (soprattutto il BackOffice) sono editabili facilmente tramite applicativi free come quello usato da me, DB Browser for SQLLite. A questo url ci sono i link di download, fra cui un comodo exe Windows portable: <https://sqlitebrowser.org/dl/>

I database, sottoforma di singoli file, sono presenti nel progetto versionati nel repository git e sono i seguenti:

* **OrderingSystem.Orders.Db**: contiene i dati sugli ordini di acquisto. Ecco le tabelle che lo compongono.
  + Orders: contiene le righe master, ognuna delle quali rappresenta un ordine di acquisto di un utente del sistema da spedire ad un suo indirizzo di consegna
  + OrderedProducts: contiene le righe slave, ognuna delle quali rappresenta un singolo prodotto dell’ordine di acquisto con la quantità e il prezzo unitario al momento di acquisto (si suppone che i prezzi possano variare in tempo reale, ma l’ordine mantiene il prezzo al momento di acquisto)
* **OrderingSystem.Products.Db**: contiene il catalaogo dei prodotti e la disponibilità a magazzino (verificata in fase di ordine). Le tabelle sono:
  + Categories: anagrafica delle categorie di prodotto (ogni prodotto appartiene ad una categoria)
  + Products: anagrafica dei prodotti a listino, con disponibilità a magazzino
* **OrderingSystem.Users.Db**: contiene i dati utente, tra cui quindi l’anagrafica degli account registrati (clienti del sistema) e gli indirizzi di spedizione utilizzabili in fase di ordine. Le tabella sono:
  + Accounts: ogni record rappresenta un utente registrato
  + Addresses: ogni record rappresenta un indirizzo di spedizione associato ad un utente (FK). Uno di questi indirizzi viene identificato come indirizzo di spedizione principale (da usarsi come indirizzo di default in fase di ordine).

L’accesso ad ogni database, tramite tecnologia EFCore, è permesso tramite connection string definita nel file appsettings.json di ogni progetto WebApi.

# Descrizione delle singole tipologie di progetto

Ogni servizio, essendo progettato in Clean Architecture, ha una struttura interna simile che separa nettamente tutti i concetti, raggruppandoli in maniera omonegea.

Ecco una spiegazione di un progetto tipo (es. Users):

**Users.Contracts:** contiene tutte le astrazioni del servizio e le classi di dominio

* **Clients**: contiene le interfacce che rappresentano i client verso servizi esterni, senza una diretta implementazione che imporrebbe la verticalizzazione di una specifica tecnologia (http, message bus…)
* **Domain**: contiene le classi del dominio applicativo specifico del servizio (es. utenti)
* **Persistence**: contiene le interfacce che astraggono il servizio dalla reale tecnologia di persistenza dei dati. Ho scelto di usare il pattern Repository (<https://martinfowler.com/eaaCatalog/repository.html>)

**User.Infrastructure:** In generale riflette la stessa struttura dei Contracts, dandone una implementazione concreta

* **Clients**: implementazione concreta dei client per le comunicazioni esterne (es. http client)
* **Domain**: generalmente vuota (le entità di dominio sono nei Contracts), ma può contenere implementazioni specifiche di quelli che di solito vengono denominati Domain Service nella terminologia DDD, cioè componenti specifici che operano su più oggetti di dominio, implementando quindi delle logiche di dominio specifiche, ma non di business (<https://ddd-practitioners.com/home/glossary/domain-service/>).
* **Persistence**: implementa i repository con l’utilizzo di una specifica tecnologia di persistenza e accesso ai dati. In questo caso, come richiesto, Entity Framework Core. In questo contesto va menzionato un ulteriore pattern utilizzato in accoppiata con i repository, cioè la Unit Of Work (<https://martinfowler.com/eaaCatalog/unitOfWork.html>). Entity Framework implementa già una sorta di unit of work (il DbContext), ma avendo scelto una architettura Clean, anche questo aspetto va astratto con una opportuna IUnitOfWork (fra i Contracts) e la sua relativa implementazione concreta per EFCore che di fatto non è altro che un wrapper attorno al DbContext. Va inoltre sottolineato come l’uso combinato di Repository e UnitOfWork, permette l’uso dei Repository nella loro forma più pura, cioè come collection astratte che quindi non hanno un vero e proprio metodo di salvataggio dei dati (come di fatto in una normale collection non avrebbe senso). E’ infatti la UnitOfWork che tiene traccia delle modifiche effettuate sul dominio e le riflette in maniera transazionale solo quando si committano esplicitamente le modifiche (così come impone di fatto il DbContext nativo di EFCore).

**Users.WebApi:** rappresenta il progetto avviabile del servizio e contiene le logiche applicative (business logic)

* **Application**: contiene le logiche business che rappresentano i singoli flussi applicativi. Con una metodologia Agile, qui sarebbero implementati i singoli Use-Case. Per questo motivo ho scelto di implementare queste logiche seguendo il pattern CQS/CQRS (<https://martinfowler.com/bliki/CommandQuerySeparation.html>). In altre parole, ogni singolo flusso applicativo (use-case) è implementato con una classe detta “command handler”. Questo approccio permette di separare molto bene le singole funzionalità e renderle facilmente testabili in maniera singola, esponendone le dipendenze in maniera chiara. Questo approccio architetturale si distingue da quello più classico a 3-layers, dove lo strato business è implementato tramite delle classi “service” dove ogni flusso funzionale è implementato con un metodo. Questo approccio però si presta a molti aspetti negativi, primo fra tutti quello di forzare il raggruppamento di metodi all’interno di una classe “service” che impone per tutti quanti i suoi metodi un set di dipendenze che, di fatto, è il massimale delle dipendenze dei metodi della classe stessa.  
  Inoltre, ogni command handler può implementare (opzionale) un sistema integrato di validazione dell’input, appoggiandosi alla libreria FluentValidation (<https://github.com/FluentValidation/FluentValidation>).
* **Configuration**: contiene le classi che rappresentano le configurazioni applicative. Generalmente riflettono sezioni specifiche degli appsettings.
* Controllers: contiene le classi controller che rappresentano lo strato “presentazionale” di una web api
* **Middlewares**: contiene eventuali middlewares custom della pipeline di Asp Net Core. In questo progetto mi sono avvalso di un solo middleware custom, ErrorHandlingMiddleware, che viene installato all’inizio della pipeline, in modo da catturare qualsiasi eccezione e trasformarla in un oggetto di output unificato, seguendo lo standard ProblemDetails definito dal protocollo omonimo (<https://datatracker.ietf.org/doc/html/rfc7807>)
* **Models**: contiene le classi di input e output della web api. Queste classi vanno distinte da quelle dello strato business, cioè gli input e gli output dei command handlers. Anche se spesso la relazione fra queste classi è molto stressa, spesso 1:1, è importante mantenere le classi concettualmente separate ed eventualmente avvalersi di librerie di mapping automatico (Automapper, Mapster…). In questo progetto, per semplicità, ho eseguito tutti i mapping in maniera manuale, avvalendomi di extensions method raccolti convenzionalmente nelle classi denominate ControllersHelper. Quest contengono quindi tutti i metodi di conversione fra le classi di input e output dello strato api, quelle dello strato business (command handlers) e quelle di dominio (entità).
* **appsettings.json**: contiene le configurazioni applicative di avvio, le connection strings, e le url ai servizi esterni accedibili tramite gli http clients
* **Program.cs**: contiene la configurazione di avvio dell’applicativo web, con la registrazione di tutti i componenti registrati nel motore di dependency injection di Asp Net Core e la pipeline di gestione delle request.

**Users.Tests**: contiene le classi e i metodi di unit testing per il servizio. Idealmente potrebbe contenere i test di tutti gli strati, in modo da implementare sia test di unità puri (singole classi), sia test di integrazione. Per semplicità ho implementato solo i test di unità sullo strato applicativo, concentrandomi quindi sui test dei command handler, dato che questi rappresentano appunto le singole logiche business. Per fare questo, ogni classe di test si concentra su uno specifico handler (e quindi su uno specifico flusso business). Per un dato handler quindi, vengono istanziati i mock delle sue dipendenze, tramite la libreria Moq (<https://github.com/devlooped/moq>). Dato il flusso di esecuzione di un handler, vengono quindi implementati tanti test singoli quanti sono i possibili rami di esecuzione (sia casi negativi, sia positivi). Generalmente, quindi, si tratta di avere N test che coprano tutti i casi negativi ed almeno un caso identificato come “happy path” dove il flusso business si conclude con successo. Lo scopo, in questo modo, è anche quello di raggiungere una percentuale di code-coverage del singolo handler che sia quanto più vicina possibile al 100%.

# Flusso tipico di una request/response

Il modello architetturale scelto, combinato con i vari pattern descritti (Domain Driver Design, CQS/CQRS, Repository, UnitOfWork…) definisce un flusso di gestione delle singole richieste alla web api specifico e coerente per tutti i servizi del sistema. Di seguito un tipico flusso:

Immagine che contiene testo, schermata, diagramma, numero

Descrizione generata automaticamente

# Note sui test

I progetti “Tests”, come detto contengono gli unit test. Per semplicità e tempistiche ridotte, non ho implementato tutti i possibili test e mi sono invece concentrato su alcuni esempi più interessanti. Come detto, inoltre, i test implementati si sono concentrati sulla logica business. Quindi ogni classe di test rappresenta i test focalizzati su un determinato command handler, che di fatto ne costituisce lo specifico “system under test” (sut).

In generale, ogni classe di test è definita così:

* Un metodo statico che permette di creare rapidamente una istanza di input del command handler (non necessario se l’input è un dato di tipo primitivo come int, string…)
* Una serie di field readonly, uno per ogni dipendenza del command handler sottoforma di mock della dipendenza stessa rappresentata tipicamente da una interfaccia (es. Mock<IAccountsRepository>)
* Un costruttore che crea i mock senza alcun setup comportamentale e istanzia il command handler da testare a cui vengono passati i mock nel costruttore
* Un metodo di test (Fact) che rappresenta un singolo ramo di esecuzione possibile all’interno del metodo Execute() del command handler. Ogni metodo di test, è implementato con il pattern AAA: Arrange (setup dei mock), Act (esecuzione del command handler), Assert (verifica degli esiti).

A proposito delle verifiche finali, oltre alle classiche Assert sul risultato, ho sfruttato ampiamente anche la possibilità offerta dalla libreria Moq per verificare anche che i singoli mock delle dipendenze iniettate nel command handler venissero eseguite un numero certo di volte, coerentemente con quanto ci si aspetta dall’esecuzione della logica business che, non essendo sempre completa, non sempre coinvolge tutte le dipendenze.

Per una visione rapida di tutti i test implementati, da Visual Studio ho sempre usato la view Test Explorer (Menu Test -> Test Explorer). Qui una vista della situazione attuale:

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente