Backend RESTful: un approccio conforme al pattern CQRS ed ai principi della programmazione SOLID

Marcello Esposito
esposito.marce@gmail.com

22 giugno 2021

Sommario

In questo articolo si presenta un'impostazione allo sviluppo di backend applicativi RESTful, con un esempio concreto basato su tecnologia net core WebApi. La metodologia proposta incoraggia l'uso delle moderne best practices di sviluppo software: è impostata conformemente al pattern CQRS, aderisce ai principi della programmazione SOLID, gestisce con approccio Aspect-Oriented le funzionalità trasversali — autorizzazione, validazione, logging, notifica — favorisce un uso strutturale e coerente della Dependency Injection, uniforma la stesura dei layer applicativi fornendo precise collocazioni per i moduli, in base alle loro responsabilità. Questi aspetti migliorano la comprensione dell'architettura applicativa e fluidificano le interazioni tra i ruoli di un team di sviluppo software, incrementando così il livello di qualità del prodotto finale e riducendo il time-to-market. I risultati dell'uso di questo approccio in diversi contesti produttivi di livello enterprise ne hanno dimostrato la validità ed il valore.

1 Introduzione

La scrittura delle moderne applicazioni richiede il possesso di un set di competenze molto esteso. La buona conoscenza e l'adesione alle *best practices* di sviluppo software, infatti, garantisce la possibilità di eseguire l'applicazione sulle moderne infrastrutture — per es. in ambienti *cloud* — il rispetto delle leggi vigenti in tema di sicurezza informatica e riservatezza dei dati, oltre che alcuni importanti requisiti di qualità del software: per es. testabilità, correttezza, scalabilità, estensibilità. In compenso di un maggior carico di lavoro e di maggiori competenze necessarie in fase di realizzazione del software, si ottiene un notevole incremento della qualità del prodotto e, soprattutto, notevoli risparmi in fase di distribuzione, conduzione e manutenzione. In certi casi i risparmi possono derivare anche dalla possibilità di gestire questi processi mediante parziale o totale automazione.

Le architetture software applicative recentemente tendono verso un modello che vede fortemente disgiunti i moduli di backend e frontend, sempre più spesso realizzati con tecnologie software del tutto differenti. In conseguenza di ciò, restano disgiunti anche i relativi processi di sviluppo e le competenze tipiche di ciascuno dei due ambiti, che risultano più circoscritte e specialistiche. Questo è vero in particolar modo per le cosiddette applicazioni transazionali, quelle che elaborano atomicamente richieste complesse — dette *conversazioni* — alle quali sono associate molteplici transazioni in lettura e/o scrittura su una o (spesso) più basi dati. Le Single-Page-Application (SPA), per esempio, seguono esattamente questo modello e lo declinano sulle applicazioni per il web. Anche le *app* per terminali mobili di tipo transazionale, come un'app di banking online, seguono abbastanza fedelmente questo stesso modello. Il *frontend* contiene l'insieme delle componenti funzionali alla realizzazione dell'interfaccia utente: i componenti grafici — caselle di testo, bottoni, menù, etichette, barre di stato — insieme con la loro logica di interfaccia — abilitazione e disabilitazione guidata da stati ed eventi, legende di colore, logiche di navigazione e così via. Il *backend*, solitamente realizzato come servizio web che espone un'interfaccia RESTful, offre i veri e propri servizi applicativi — tra i quali la logica applicativa ed autorizzativa, archiviazione, gestione delle sessioni, logging, profiling, sicurezza. I moduli di frontend e backend hanno il loro punto d'incontro nell'interfaccia del modulo di backend esposta verso il frontend, la quale rappresenta anche l'unico modulo le cui specifiche devono essere concordate tra i gruppi di sviluppo del frontend e quello del backend. In questo modo, il progetto del modulo di presentazione può procedere in maniera sostanzialmente disgiunta dalla progettazione del modello di dominio, della logica applicativa e del livello di persistenza.

In quest'articolo ci si concentra esclusivamente sul sottosistema di backend, descrivendo l'architettura *RockAPI*, un template applicativo in tecnologia Microsoft net core WebApi, ispirato ai principi del pattern CQRS, della programmazione SOLID e distribuito nei termini della licenza MIT su GitHub [1]. Ogni commit è stato realizzato con un intento didattico ed è dettagliatamente commentato per chiarire la logica con cui l'architettura è stata progettata, passo per passo. Tutto il codice presentato in questo articolo è presente nello stesso repository nel branch `article`.

RockAPI ha dimostrato in diversi contesti un netto miglioramento delle caratteristiche di qualità del software, un incremento dei livelli di sicurezza, della produttività dei team ed una riduzione dei tempi di sviluppo.

Nella sezione 2 si descrivono i principi ed i pattern di base che ispirano l'architettura RockAPI. Nella sezione 3 si descrivono le fasi elaborative che ogni azione indirizzata verso il sistema RockAPI attraversa. La sezione 4 descrive i singoli progetti della soluzione RockAPI, mentre la sezione 5 descrive un esempio pratico di realizzazione di una semplice applicazione, evidenziando i ruoli dei componenti fondamentali dell'architettura. La sezione 6 descrive la modalità con cui si opera nel caso in cui intervengano, in corso d'opera, modifiche o aggiunte nell'implementazione di una funzionalità, con minimo impatto sul codice esistente e in accordo con i moderni paradigmi di sviluppo agile del software. Nella sezione 7 si mostra come procedere per inibire il log applicativo di

informazioni riservate. Infine, nella sezione 8, si elencano alcune conclusioni su questo lavoro.

2 I principi di base

L'architettura RockAPI si basa principalmente su alcuni mattoni fondamentali:

- i principi della programmazione SOLID;
- il pattern CQRS.

Di seguito si descrivono brevemente questi due concetti.

2.1 I principi della programmazione SOLID

Alcuni utili principi che devono ispirare lo sviluppo del software moderno, sono quelli che partecipano alla composizione dell'acronimo SOLID.

Il **Single Responsibility Principle** (SRP — la 'S' in SOLID) indica il principio secondo cui un modulo deve avere un'unica responsabilità, intesa al proprio livello di astrazione. Un esempio concreto è rappresentato da una classe, nell'accezione della programmazione orientata agli oggetti (OOP), che dovrebbe preoccuparsi di svolgere un solo preciso compito.

L'**Open/Closed Principle** (OCP — la 'O' in SOLID) è il principio secondo cui un modulo dovrebbe essere chiuso alle modifiche ed aperto alle estensioni. Sempre con riferimento alle classi nella OOP, questo significa che una volta redatto e collaudato il codice di una classe, questo dovrebbe essere consolidato e mai più modificato. Eventuali estensioni al comportamento realizzato devono poter essere realizzate utilizzando specifiche metodologie, come per esempio la composizione o, in alcuni casi meno comuni, l'ereditarietà tra classi.

Il **Liskov Substitution Principle** (la 'L' in SOLID) è il principio secondo cui l'implementazione di uno specificato comportamento (leggi: di un'*interface*) non dovrebbe limitarsi a rispettare esclusivamente la definizione formale del comportamento — aspetto che il compilatore riesce a garantire alla perfezione — ma anche le sue semantiche implicite — dalle quali il compilatore resta avulso. Secondo questo principio è errato ereditare la classe `Quadrato` dalla classe `Rettangolo`, nonostante un quadrato sia indiscutibilmente una particolare realizzazione di un rettangolo.

L'**Interface Segregation Principle** (la 'I' in SOLID) sancisce l'opportunità che l'interfaccia di un modulo, che ne pubblica i servizi offerti, contenga funzionalità aventi semantiche strettamente correlate tra loro. Secondo questo principio, sarebbe dunque improprio lasciare che una classe esponga allo stesso tempo un metodo che effettui un calcolo aritmetico, insieme ad un metodo che effettui una stampa, trattandosi di contesti troppo differenti.

Il **Dependency Inversion Principle** (la 'D' in SOLID) indica la caratteristica di ogni modulo di dipendere esclusivamente dalle definizioni astratte dei servizi offerti dagli altri moduli. In termini più concreti, ogni classe che dipenda

dai servizi di altre classi, deve limitarsi esclusivamente ad una dipendenza dalla loro interfaccia e non fare alcuna assunzione sulla specifica implementazione di quella interfaccia. Spesso questo principio viene enunciato con la frase:

> *"Program to an interface, not an implementation."*

L'impostazione dello sviluppo applicativo secondo questi cinque principi conferisce al software benefici in termini di qualità.

2.2 Il pattern CQRS

Nel contesto dello sviluppo SOLID delle applicazioni, si inquadra anche il pattern architetturale denominato *Command/Query Responsibility Segregation* ed abbreviato con l'acronimo CQRS. Seguendo questo pattern, l'esecuzione di azioni sul sistema è demandata a specifici oggetti, uno per ciascuna azione. Questi oggetti appartengono a due diverse macro-classi. Gli oggetti di tipo Command e gli oggetti di tipo Query:

- l'esecuzione di un Command modifica lo stato del sistema e non restituisce alcun risultato, eccetto eventualmente l'informazione sulla corretta o errata esecuzione dell'Action;
- gli oggetti di tipo Query lasciano inalterato lo stato del sistema ed hanno l'unico obiettivo di restituire un risultato.

Il pattern CQRS mira proprio a suddividere in maniera esplicita queste due categorie di azioni partendo dal presupposto che il loro obiettivo è differente.

I Command richiedono al sistema l'esecuzione di logiche di business, che in certi casi possono essere anche particolarmente complesse. Si pensi ad esempio ad una prenotazione di una visita ambulatoriale in un sistema sanitario, che deve verificare la compatibilità tra la prestazione richiesta ed il medico selezionato, la disponibiltà di quel medico, la compatibilità dell'ambulatorio selezionato con la prestazione, la disponibilità dell'ambulatorio nell'intervallo di tempo selezionato, la validità del regime sanitario, ecc. In caso di successo, la prenotazione dovrà avere anche impatti con il sistema di customer care e di fatturazione, mediante integrazione applicativa, eseguendo così delle transazioni distribuite. Le Query mirano invece ad estrarre dal sistema informazioni nel modo più rapido possibile. Le informazioni sono talvolta elementari e richiedono una semplice interrogazione in una base dati. Capita spesso però che siano frutto di calcoli che coinvolgono logiche di business complesse ed elevate moli di dati. Un esempio in questo senso è rappresentato dalla pagina dei risultati di una ricerca svolta su un portale di e-commerce: oltre alla lista degli articoli, viene presentata la loro suddivisione (faceting) su varie dimensioni (marche, fasce di prezzo, condizioni dell'articolo, provenienza, ecc.), la distribuzione dei feedback storicamente ottenuti, la scontistica dovuta alle offerte correntemente attive, i tempi di consegna.

Così come differenti sono le nature dei sistemi OLTP rispetto ai sistemi OLAP, la separazione di Command e Query tiene in conto le differenze descritte e consente di ottimizzare separatamente le modalità di esecuzione di queste due classi di azioni, anche basandosi sull'evidenza statistica che le frequenze di esecuzione di Command e Query su un classico sistema transazionale stanno nel rapporto di circa 1:10. Nel seguito di questo articolo, quando non è importante mettere in evidenza la differente natura di Command o Query eseguite sul sistema, si parlerà più genericamente di Action.

3 La execution chain

RockAPI affronta strutturalmente tre problematiche comuni nella stesura di applicazioni transazionali.

- La **validazione** dei Data Transfer Objects (DTO). La validazione comporta la verifica del DTO, contenente i dati di ingresso che alimentano l'esecuzione dell'Action. Un DTO non valido produce l'interruzione dell'Action, prevenendo problematiche di sicurezza oppure errori di esecuzione.
- L'**autorizzazione** all'esecuzione delle Action. L'autorizzazione viene a valle della validazione, ed ha l'obiettivo di verificare che l'Action venga eseguita nel rispetto delle politiche autorizzative. Per ogni Action è possibile definire uno o più specifici criteri autorizzativi. Tipicamente questo consiste in un'analisi del DTO in ingresso contestualmente con l'identità dell'utente autenticato, con l'obiettivo di verificare che il DTO possa alimentare la specifica Action quando invocata dal particolare utente.
- La **notifica** di esecuzione delle Action. La notifica invia un segnale di avvenuta esecuzione di un'Action ad uno o più destinatari designati, eventualmente consegnandogli anche parte delle informazioni contenute nel DTO o una loro opportuna elaborazione. Per esempio, a valle dell'esecuzione di un Command che aggiorni un documento in un archivio, questa funzionalità potrebbe essere invocata per notificare ad un motore di indicizzazione full-text la variazione del documento, sollecitandone così la re-indicizzazione.

Questi tre diversi *aspetti* di un'applicazione vengono gestiti proprio con il paradigma dell'Aspect Oriented Programming (AOP) [4] attraverso l'uso di opportune convenzioni. Per ogni Action c'è un esatto punto in cui è possibile *iniettare* la logica di esecuzione di una validazione, di un'autorizzazione o di una notifica. Se è presente una classe che implementi una delle suddette logiche, il framework automaticamente la riconosce — grazie al suo legame di discendenza da una specifica classe generica — e la inserisce nella catena di esecuzione dell'Action a cui appartiene. Di conseguenza, anche in un applicativo complesso, è particolarmente semplice individuare la collocazione della classe contenente una delle logiche di una specifica Action, ed essere certi che tale

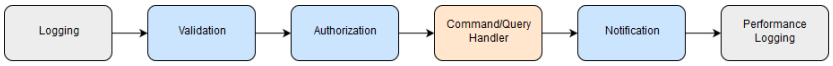

Figura 1: La execution chain

classe non contenga altro che quella logica, ancora una volta in osservanza al principio SRP (vedi sezione 2.1). Una logica complessa può essere implementata mediante la scrittura di più classi, ognuna delle quali prende in carico una parte del problema. Il framework riconosce ed inietta automaticamente tutte le classi in fase di esecuzione dell'Action. In assenza della classe che implementi una di queste logiche, il framework non adotta alcuna particolare strategia.

Oltre ai tre aspetti descritti, RockAPI esegue automaticamente il logging del DTO di ogni Action che viene eseguita sul sistema, con un duplice obiettivo:

- abilitare un auditing completo e dettagliato;
- tracciare costantemente i tempi di esecuzione di ciascuna Action.

Grazie alla versatilità della libreria di logging[1], mediante semplici operazioni di configurazione le informazioni possono essere archiviate su disco locale, su un syslog server, su un database, o anche su più di questi supporti contemporaneamente. È inoltre possibile effettuare la personalizzazione della funzionalità di deserializzazione per singolo DTO. Ciò è opportuno nel caso in cui il DTO contenga informazioni riservate — per es. il DTO di autenticazione all'applicazione, contenente la password dell'utente — al fine di inibirne il log in chiaro. Oppure per aggiungere informazioni a quelle strettamente contenute nel DTO e funzionali ad una migliore interpretazione dei log.

Un uso sistematico della Dependency Injection (DI) è alla base dell'uso di RockAPI. Grazie a questa metodologia è possibile configurare in un punto ben definito, denominato *composition root* [3], tutte le associazioni tra le interfacce dei servizi e le relative implementazioni, nonché il ciclo di vita di queste ultime. Tali associazioni possono essere statiche, oppure determinate da specifiche condizioni di utilizzo dell'applicativo (per es. variabili di configurazione), a beneficio della flessibilità conferita all'architettura software risultante.

In occasione dell'esecuzione di ogni Action, RockAPI definisce una *execution chain* costituita dai seguenti blocchi funzionali (vedi Figura 1), elencati in ordine di esecuzione.

- **Logging**. È il blocco che effettua il log dell'Action eseguita, immediatamente prima dell'esecuzione. Il log contiene l'istante di esecuzione, l'Action eseguita, la serializzazione del DTO che alimenta l'Action.
- **Validation**. È il blocco che effettua la validazione del DTO di ingresso. Nel caso in cui il DTO non risulti valido, la execution chain si interrompe in questo blocco.

[1] RockAPI utilizza la libreria Serilog [8].

- **Authorization**. È il blocco che effettua la autorizzazione sull'Action. Nel caso in cui l'Action non superi l'autorizzazione, la execution chain si interrompe in questo blocco.
- **Command/Query Handler**. È il blocco in cui viene concretamente eseguita l'Action invocata.
- **Notification**. È il blocco avente la responsabilità di inviare eventuali notifiche di avvenuta esecuzione di un Command[2].
- **Performance Logging**. È il blocco che ha in carico il tracciamento dei tempi di esecuzione dell'Action, principalmente al fine di poter effettuare delle analisi di performance durante tutto il ciclo di vita dell'applicazione.

In questa execution chain, i moduli di Logging e Performance Logging (colorati a sfondo grigio in Figura 1) hanno un'implementazione comune a tutte le Action invocate. Gli altri blocchi prevedono invece la definizione di una specifica classe per ogni diversa Action. Il blocco *Command/Query Handler* richiede obbligatoriamente la realizzazione di una classe. In caso contrario, il framework produce un errore. Tale errore viene generato al tempo di runtime immediatamente dopo l'avvio dell'applicazione. In questo modo si scongiura l'eventualità che una Action resti non implementata e che ci si possa accorgere del problema solo quando l'applicazione è stata già distribuita. Per i restanti moduli non è obbligatoria la creazione di una classe per ognuna delle Action: in occasione dell'esecuzione di un'Action priva di una o più di queste classi, le relative attività non vengono svolte.

Grazie alla definizione di questa execution chain, le attività di logging e di performance logging sono sistematiche. Inoltre resta ben chiara la collocazione delle classi responsabili delle varie funzionalità ortogonali. Un team leader che voglia controllare il lavoro svolto dal suo gruppo, saprebbe immediatamente orientarsi tra le diverse funzionalità legate all'esecuzione di ogni Action applicativa.

4 La suddivisione in progetti della solution

La solution RockAPI è costituita da una serie di progetti, descritti di seguito (vedi Figura 2).

- `CQRS`: è un progetto infrastrutturale in cui l'utente di RockAPI molto probabilmente non dovrà intervenire con modifiche. Contiene le classi base utili alla definizione del pattern CQRS: Command, Query, le classi base relative alla validazione, la autorizzazione e alla notifica, le classi base relative alle eccezioni di validazione e di autorizzazione.

[2]Per le Queryil blocco dell notifiche non è attivo. Può comunque essere attivato con minimo sforzo mediante una semplice modifica al file `CQRSBindings.cs`.

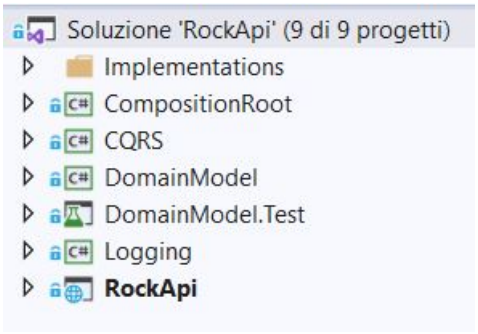

Figura 2: I progetti della soluzione

- `CompositionRoot`: è la *composition root* del progetto, che definisce tutte le associazioni tra i servizi e le relative implementazioni. All'interno di questo progetto, il file denominato `CQRSBindings.cs` contiene i collegamenti delle classi infrastrutturali del CQRS e molto probabilmente non ci sarà mai bisogno di modificarlo. Il file denominato `CustomBindings.cs` è invece il file all'interno del quale verranno posizionate tutte le regole applicative di associazione tra interfacce ed implementazioni. Questo file può essere ulteriormente modularizzato qualora le regole diventino numerose e sia opportuno suddividerle in gruppi. Il file `RootBindings.cs` richiama i precedenti due files e può essere modificato in occasione di una modularizzazione del secondo. Il progetto `CompositionRoot`, insieme con il frontend REST — contenuto nel progetto `RockAPI` — sono gli unici progetti a referenziare la libreria utilizzata come DI container. Tutti gli altri progetti restano agnostici rispetto al DI container utilizzato. Si limitano ad adottare la buona pratica dell'*inversione di dipendenza*, comportando benefici in termini di riusabilità, testabilità, estensibilità.

- `DomainModel.Test`: contiene degli unit tests scritti in NUnit3 [9] che verificano la corretta funzionalità degli handler di esempio. Può essere alimentato con i reali unit tests dell'applicativo.

- `Logging`: è anch'esso un progetto infrastrutturale. È opportuno apportare modifiche solo nel file `LogConfigurator.cs` per configurare la strategia di logging. Il log è realizzato mediante uso della libreria Serilog [8].

- `RockAPI`: è il vero e proprio progetto WebApi in tecnologia net core. Rispetto ad un progetto standard WebApi sono state inserite le istruzioni per l'integrazione con il container utilizzato per la dependency injection [6] e per l'attivazione del framework di logging [8]. Nessun'altra modifica è stata apportata al progetto, cosicché RockAPI non si discosta in nessun modo dai comportamenti di una normale solution WebApi.

- `Implementations`: è una *solution folder* contenente progetti che implementano i servizi di dominio (vedi sezione 4.1).

4.1 La cartella `Implementations`

Come vedremo, le specifiche dei servizi, nella forma di interfacce, sono contenute nel progetto `DomainModel`. Le implementazioni di tali servizi, nella maggior parte dei casi, si trovano in progetti ospitati all'interno della cartella `Implementations`. È una *solution folder* e riveste un'importanza fondamentale nell'architettura RockAPI. I progetti che implementano i servizi di dominio, infatti, spesso selezionano una specifica tecnologia per la loro implementazione. Ecco alcuni esempi.

- Un progetto che implementa lo strato di integrazione con il database principale dell'applicazione. Questo progetto adotta delle specifiche scelte sulla categoria di database utilizzata (relazionale, documentale, chiave-valore, ecc.), sulla sua tecnologia (PostgreSQL, MariaDB, MongoDB, Redis, ecc.) e sulla specifica strategia utilizzata per l'implementazione del servizio.
- Un servizio di integrazione con altri sistemi, che potrebbe utilizzare uno dei paradigmi di integrazione applicativa più comuni[3]:
  - file transfer;
  - shared database;
  - remote procedure invocation (tra cui i comuni web services).
  - un sistema di messaggistica, che potrebbe utilizzare uno tra i vari middleware oggi disponibili.
- Un servizio di indicizzazione *full-text* dei dati.
- Un servizio di crittografia, compressione o hashing dei dati.

Di seguito, servizi simili a tutti quelli elencati verranno denominati Volatile Dependencies, cioè dipendenze le cui implementazioni non sono necessariamente uniche e stabili nel tempo, in accordo con la definizione fornita in [3]. In linea di principio, per ognuno dei servizi elencati, più di un progetto implementativo potrebbe essere presente nella stessa applicazione, fornendo così implementazioni alternative per gli stessi servizi di dominio. Oppure una certa modalità di implementazione inizialmente selezionata potrebbe essere sostituita da un'altra nel tempo, perché più efficiente o perché nel frattempo sono cambiate le condizioni al contorno. Ed è possibile selezionare dinamicamente, volta per volta, l'implementazione da utilizzare tra le varie disponibili a seconda delle opportunità. La selezione avviene grazie all'uso della *dependency injection*. In sezione 6 vedremo chiaramente un esempio di tale *"implementation switch"*.

[3] Vedi [2].

5 Un esempio concreto

In questa sezione si descrive un esempio concreto di utilizzo di RockAPI. Vengono descritte le attività per creare Command e Query, astrarre VOLATILE DEPENDENCIES e realizzare le relative implementazioni, collegare le implementazioni ai servizi mediante uso della dependency injection.

5.1 Creare un Command

Come precedentemente descritto, un Command è un'Action dell'applicazione che riceve un DTO di ingresso, esegue un'elaborazione che modifica lo stato del sistema e non restituisce alcuna informazione. Per creare un nuovo Command è necessario scrivere almeno due classi:

- il DTO di input;
- l'handler del comando.

Sia il DTO che l'handler sono due normali classi che implementano le logiche di dominio di una Action. Per farlo usano a loro volta altre classi e servizi anche questi appartenenti al dominio. Pertanto, sono ospitate all'interno del progetto `DomainModel`. Ecco come vengono realizzate concretamente.

Nel progetto `DomainModel`, all'interno della cartella `CQRS/Commands`, si crea una cartella descrittiva dell'azione eseguita dal comando. Per esempio, si possono utilizzare i seguenti nomi per la cartella: `AddUser`, `UpdatePerson`, `DeleteArticle`.

All'interno della cartella appena creata si crea la classe del DTO di input. Il nome della classe è scelto aggiungendo il suffisso `Command` al nome della cartella. Per esempio: `AddUserCommand`. Il DTO è una normale classe `C#`, che non deriva da alcuna altra classe. A titolo esemplificativo, può apparire come segue.

```
public class AddUserCommand
{
    public string Username { get; set; }
    public string Password { get; set; }
}
```

Listato 1: Il DTO `AddUserCommand`

Nella stessa cartella del DTO, si crea l'handler del Command. Il nome di questa classe è scelto aggiungendo il suffisso `Handler` al nome del DTO. Per esempio: `AddUserCommandHandler`. La classe handler deriva dall'interfaccia generica `ICommandHandler<>`, in cui il tipo generico da istanziare è quello del DTO relativo all'handler che si sta scrivendo. Ecco un esempio.

```
public class AddUserCommandHandler :
    ICommandHandler<AddUserCommand>
{
```

```
    public void Handle(AddUserCommand command)
    {
        // Implementation here
    }
}
```

Listato 2: Scheletro dell'handler del Command `AddUserCommand`

Sarà necessario aggiungere la clausola di utilizzo del namespace `CQRS.Commands`[4].

Si completa l'implementazione dell'handler all'interno del metodo `Handle()`, che è il solo metodo definito dall'interfaccia implementata dall'handler.

È molto importante che l'handler resti una classe di dominio pura e, come tale, la sua implementazione non dipenda da altro che da interfacce e classi di dominio. La sua implementazione deve restare astratta rispetto ai dettagli implementativi dell'architettura applicativa ed alle tecnologie utilizzate. Qualora l'esecuzione dell'Action necessiti di VOLATILE DEPENDENCIES — per esempio la lettura o la scrittura su database, l'accesso ad un servizio web o l'invio di un messaggio di posta — è necessario astrarre tali servizi con la definizione di un'apposita interfaccia. In questo caso l'interfaccia del servizio di aggiunta di utente potrà chiamarsi `IAddUser`. Si procede come segue.

Nel progetto `DomainModel`, all'interno della cartella `Services`, si definisce l'interfaccia del servizio necessario.

```
public interface IAddUser {
  void Add(User user);
}
```

Listato 3: L'interfaccia del servizio `IAddUser`

La classe `User` può essere creata all'interno della cartella `Classes`, sempre nel progetto `DomainModel`.

```
public class User
{
    public User(string username, string password, bool active)
    {
        this.Username = username;
        this.Password = password;
        this.Active = active;
    }

    public string Username { get; protected set; }
    public string Password { get; protected set; }
    public bool Active { get; protected set; }
}
```

Listato 4: La classe di dominio `User`

[4]Nel seguito si trascurerà di indicare le necessarie clausole di inclusione dei namespace, che andranno aggiunte su necessità.

Si noti che questa interfaccia trova anch'essa la giusta collocazione all'interno del progetto `DomainModel`, perché l'aggiunta di un utente è un'operazione di logica di dominio.

Mediante uso della dependency injection, si inietta l'interfaccia appena creata all'interno dell'handler, che quindi diventa come indicato di seguito.

```
public class AddUserCommandHandler :
    ICommandHandler<AddUserCommand>
{
    private readonly IAddUser addUser;

    public AddUserCommandHandler(IAddUser addUser)
    {
        this.addUser = addUser ?? throw new
            ArgumentNullException(nameof(addUser));
    }

    public void Handle(AddUserCommand command)
    {
        var user = new User(command.Username,
            command.Password, true);
        this.addUser.Add(user);
    }
}
```

Listato 5: L'handler del Command `AddUserCommand`

Il codice così composto supera la compilazione senza alcun errore, anche se il servizio `IAddUser` non è stato ancora implementato. Grazie a questa tecnica, il programmatore resta concentrato sull'implementazione della logica del Command rimandando la realizzazione dei servizi ausiliari ad un momento successivo. Inoltre, questa implementazione non può essere involontariamente dimenticata: l'eventuale esecuzione dell'applicativo in assenza di un'implementazione del servizio correttamente configurata all'interno del DI container, genererebbe un errore di runtime immediatamente all'avvio dell'applicazione (in accordo con la prassi *fail-fast*). Successivamente vedremo dove implementare le Volatile Dependencies (§5.1.1) e come collegare le interfacce dei servizi di dominio a queste implementazioni (§5.1.2).

Dal punto di vista della execution chain, la realizzazione di queste due classi è sufficiente se non si devono realizzare logiche specifiche per le fasi di validazione del DTO di ingresso, di autorizzazione all'esecuzione del Command e alla notifica di esecuzione. L'Action verrà eseguita e, contemporaneamente, ne verrà effettuato il log all'inizio dell'esecuzione ed anche alla fine per tener traccia del tempo di esecuzione. L'unica cosa che resta da fare è predisporre un controller che colleghi la WebApi al Command appena realizzato. Per realizzare il controller si procede nella consueta maniera. All'interno del progetto `RockAPI`, nella cartella `Controllers`, si definisce il controller `AddUserController`, come segue.

```
[Route("api/[controller]")]
[ApiController]
public class AddUserController : ControllerBase
{
    private readonly ICommandHandler<AddUserCommand> handler;

    public AddUserController(ICommandHandler<AddUserCommand>
        handler)
    {
        this.handler = handler ?? throw new
            ArgumentNullException(nameof(handler));
    }

    // POST: api/AddUser
    [HttpPost]
    public void Post([FromBody] AddUserCommand command)
    {
        handler.Handle(command);
    }
}
```

Listato 6: Il controller `AddUser` con il suo metodo POST

Il controller è realizzato in piena aderenza alle comuni metodologie valide per le applicazioni WebApi. Una pratica sistematicamente adottata in RockAPI, consiste nell'utilizzo della constructor injection per acquisire il riferimento all'handler che ha in carico la reale esecuzione della logica di dominio relativa all'Action invocata. Come si può notare, il controller è quindi una classe di mero collegamento tra il framework WebApi ed il Command. In questa classe non vi è, e non vi deve essere, alcuna logica applicativa, ma esclusivamente una delega di esecuzione, effettuata grazie all'iniezione dell'handler del Command nel controller, ancora una volta sfruttando la dependency injection.

Il codice così composto può essere compilato senza errori, ma non appena si esegue l'applicazione, si genera il seguente errore di runtime.

```
System.InvalidOperationException: 'The configuration
is invalid. Creating the instance for type AddUser-
Controller failed. The constructor of type AddUser-
CommandHandler contains the parameter with name 'addUser'
and type IAddUser, but IAddUser is not registered. For
IAddUser to be resolved, it must be registered in the
container.'
```

L'errore è generato dalla libreria SimpleInjector che, all'avvio dell'applicativo, effettua una verifica di corretta istanziabilità di tutti i controllers presenti nell'applicazione. In questo caso l'errore informa che il controller `AddUser-Controller` non può essere istanziato perché una sua dipendenza, il Command `AddUserCommandHandler`, inietta un servizio per il quale non è stata

risolta la relativa implementazione. Vediamo pertanto come procedere a questa implementazione e, successivamente, collegarla al relativo servizio.

5.1.1 L'implementazione di una Volatile Dependency

Le implementazioni delle Volatile Dependencies dipendono strettamente dalle tecnologie selezionate per l'applicazione. È una Volatile Dependency l'accesso ad un database, che può essere implementato utilizzando numerose tecnologie, oltre che diversi paradigmi — database relazionali, documentali, chiave-valore, a grafo, ecc. È una Volatile Dependency l'accesso ad un servizio web, un servizio di ricerca a testo libero, il servizio offerto da un framework di messaggistica. Questi servizi sono implementati in appositi progetti appartenenti al gruppo dei progetti di implementazione e, a causa di questa loro natura, sarà conveniente raccogliere questi progetti all'interno di una *solution folder* denominata `Implementations`. Immaginiamo dunque di dover implementare il servizio astratto dall'interfaccia `IAddUser`, precedentemente definita all'interno del modello di dominio. Per semplicità, piuttosto che utilizzare una reale tecnologia database, salveremo i dati in memoria: chiameremo `FakeDatabase` la classe che implementa questo database.

Procediamo dunque con il creare un progetto, di tipo net core library, denominato `Persistence_FakeDatabase` all'interno della solution folder `Implementations`[5]. All'interno del progetto creiamo una classe `AddUser`, come indicato di seguito.

```
internal class AddUser : IAddUser
{
    private readonly IList<User> users = new List<User>();

    public void Add(User user)
    {
        this.users.Add(user);
    }
}
```

Listato 7: L'implementazione fake `AddUser` del servizio `IAddUser`

Si noti che la classe `AddUser` implementa l'interfaccia astratta del servizio di dominio. È pertanto necessario aggiungere nel progetto un riferimento al progetto `DomainModel`. Inoltre la classe è definita con visibilità `internal`. Questa regola deve essere sistematicamente rispettata da tutte le classi che implementano Volatile Dependencies, cosicché non siano direttamente visibili agli altri progetti. Vi sono due uniche eccezioni alla necessità di preservare l'opacità di queste classi:

[5] Nella creazione del nuovo progetto bisogna accertarsi di selezionare la stessa versione di net core utilizzata per gli altri progetti. In caso contrario è compromessa la corretta visibilità tra i progetti.

1. il progetto `CompositionRoot` deve avere visibilità delle classi delle VOLATILE DEPENDENCIES per poter effettuare l'associazione tra i servizi di dominio e le relative implementazioni;

2. eventuali progetti contenenti unit tests potranno accedere alle classi delle VOLATILE DEPENDENCIES al fine di poter correttamente effettuare il testing.

Vedremo nel prossimo paragrafo come garantire questa visibilità pur preservando la visibilità `internal` delle classi.

5.1.2 Il binding delle Volatile Dependencies

Come indicato nei precedenti paragrafi, le VOLATILE DEPENDENCIES hanno un'interfaccia localizzata nel progetto `DomainModel` ed una o più implementazioni all'interno di progetti infrastrutturali contenuti nella solution folder denominata `Implementations`. Il progetto `CompositionRoot` ha proprio la responsabilità di configurare tutte le regole di associazione tra le interfacce dei servizi e le relative implementazioni. Si opera all'interno del file `CustomBindings.cs` contenente, per il momento, un metodo `Bind()` con implementazione vuota. L'associazione può essere fatta scrivendo la seguente riga di codice.

```
container.Register<DomainModel.Services.IAddUser,
    Persistence_FakeDatabase.AddUser>();
```

Listato 8: La registrazione del servizio `IAddUser`

Perché questa riga di codice venga correttamente compilata è necessario che il progetto `CompositionRoot` referenzi i progetti `DomainModel` e `Persistence_FakeDatabase`. Ma questo non basta. La classe `AddUser` è infatti `internal` e pertanto non è visibile ai progetti esterni. La `CompositionRoot` è però un progetto speciale e, come tale, può essere autorizzato ad accedere anche agli internals di un progetto. Abiliteremo questo accesso inserendo la seguente riga di codice nel file `Persistence_FakeDatabase/AddUser.cs` subito prima del namespace `Persistence_FakeDatabase`.

```
[assembly: InternalsVisibleTo("CompositionRoot")]
```

Listato 9: La direttiva di compilazione per l'accesso alle classi `internal`

Questa direttiva di compilazione ha validità a livello di assembly, quindi di intero progetto. In altre parole, può essere inserita in una qualsiasi classe del progetto. Per chiarezza, potrebbe essere opportuno creare una classe deputata a contenere solo questa clausola, esplicitandone così il posizionamento.

Si noti inoltre come nel Listato 8 il nome delle classi sia stato espanso con il full-qualified namespace. In caso contrario in questo file, al crescere dei servizi disponibili, vi sarebbe in breve tempo un'esplosione delle clausole `using` con inevitabile inquinamento dello spazio dei nomi.

Con l'aggiunta di quest'ultima direttiva di compilazione, la soluzione può essere correttamente compilata ed eseguita.

5.1.3 Esecuzione di un Command

Il Command `AddUser` può essere invocato avviando l'applicazione WebApi ed effettuando una richiesta in POST all'indirizzo seguente[6].

```
http://localhost:53062/api/AddUser
```

con un payload costituito dal documento JSON seguente:

```
{
  "username": "foo",
  "password": "bar"
}
```

L'applicazione risponde con uno status code `200 OK` ed un body vuoto, che è la tipica risposta fornita da un Command. In seguito all'esecuzione si può notare che nella cartella del progetto `RockApi` è stato creato un file con un nome simile a `log20201010.txt` contenente le seguenti righe[7].

```
2020-10-10 19:53:18.251 +01:00 [INF] Action
  starting DomainModel.CQRS.Commands.AddUser
.AddUserCommand: {"Username":"foo","Password":"bar"}
2020-10-10 19:53:18.251 +01:00 [INF] Action
  executed (0 ms)
```

La configurazione della libreria di logging Serilog è contenuta nel progetto `Logging` all'interno della classe `LogConfigurator`. SeriLog è una libreria molto versatile. Può inviare i log verso differenti destinazioni, per es. file, log server, database. È inoltre possibile utilizzare per la sua configurazione il file di configurazione dell'applicazione.

5.1.4 L'aggiunta di un validator

In occasione della chiamata al Command `AddUser`, non tutti i DTO sono accettabili. Supponiamo per esempio che la password contenuta nel DTO debba essere composta da almeno otto caratteri. Questa logica può essere implementata all'interno di una classe validator, che possiamo chiamare `PasswordLongEnoughValidator`. La classe viene posizionata nella stessa cartella dell'handler del Command, e il suo corpo appare come segue.

```
public class PasswordLongEnoughValidator :
    ICommandValidator<AddUserCommand>
{
```

[6] Una richiesta in POST può essere esguita utilizzando un client REST. Un ottimo client è Insomnia, distribuito sotto licenza MIT. Esistono anche dei validi client forniti come plugin dei browser Mozilla Firefox e Google Chrome.

[7] Nel caso in cui non si stia usando il branch `production_ready` del repository RockAPI, il log viene scritto nella finestra di output dell'ambiente di sviluppo.

```
    public IEnumerable<ValidationResult>
        Validate(AddUserCommand command)
    {
        const int minLength = 8;
        if (command.Password.Length < minLength)
            yield return new ValidationResult($"Password is
                too short. Minimum length is { minLength }
                characters.");
    }
}
```

Listato 10: Una classe `Validator` garantisce una lunghezza minima della password

In questo caso sarà necessario aggiungere la clausola di utilizzo del namespace `CQRS.Validation`[8].

La sola creazione di questa classe è sufficiente ad abilitare il controllo di validazione. L'esecuzione della chiamata con il metodo HTTP POST ad `AddUser` precedentemente eseguita ora genera un errore di runtime. È necessario modificare la password per ottenere nuovamente la corretta esecuzione del metodo.

Nel caso in cui si voglia inserire un nuovo controllo di validazione è sufficiente aggiungere una nuova classe. Per esempio, se vogliamo che la password contenga almeno un simbolo, si può scrivere la classe `PasswordMustContainASymbolValidator` in questo modo.

```
public class PasswordMustContainASymbolValidator :
    ICommandValidator<AddUserCommand>
{
    public IEnumerable<ValidationResult>
        Validate(AddUserCommand command)
    {
        if (command.Password.All(c => char.IsLetterOrDigit(c)))
            yield return new ValidationResult("Password must
                contain a symbol.");
    }
}
```

Listato 11: Una seconda classe `Validator` garantisce che la password contenga un simbolo

Nel caso in cui i validatori aumentino notevolmente, è consigliabile raccoglierli in una cartella `Validators` da creare nello stesso punto in cui attualmente sono presenti tutti i validators.

Si consideri inoltre che i validators, in caso di validazione fallita, generano un'eccezione che non viene catturata e gorgoglia fino all'utente finale. Per evitare che questo accada, le eccezioni applicative devono essere gestite ed esistono differenti approcci per farlo. RockAPI non adotta alcun particolare approccio in questo senso, lasciando al programmatore la scelta su quale strategia preferire.

[8] Bisogna prestare attenzione al fatto che una classe `ValidationResult` è presente anche nel namespace `System.ComponentModel.DataAnnotation`.

5.1.5 L'aggiunta di un authorizer

L'aggiunta di un authorizer non differisce molto da quella di un validator. Immaginiamo di voler aggiungere un controllo di autorizzazione che verifichi che l'utente abbia i permessi di scrittura e che, per verificare questa condizione, si debba effettuare un accesso al database. Procediamo dunque creando una classe Authorizer che chiameremo `WritePermissionAuthorizer`. Creiamo la classe ancora una volta nella cartella `AddUser` dove attualmente è presente l'handler del Command. Eccone un esempio.

```
public class WritePermissionAuthorizer :
    ICommandAuthorizer<AddUserCommand>
{
    public IEnumerable<AuthorizationResult>
        Authorize(AddUserCommand command)
    {
                // here the authorizer body
    }
}
```

Listato 12: Lo scheletro di una classe `Authorizer`

Il corpo di questo metodo deve verificare l'autorizzazione e restituire eventualmente un oggetto che indica la violazione. Per fare questo è necessario ancora una volta usare un servizio. L'approccio è analogo a quello usato per il servizio `IAddUser` (vedi Listato 3). Chiameremo il servizio `ILoggedUserHasWritePermissions`. Creiamo l'interfaccia all'interno della cartella Services nel progetto `DomainModel`.

```
public interface ILoggedUserHasWritePermissions
{
    bool CanWrite();
}
```

Listato 13: L'interfaccia di un servizio di dominio per la verifica di un'autorizzazione

Implementiamo questo servizio nel progetto implementativo `Persistence_FakeDatabase`, creando due classi: la classe `LoggedUserHasWritePermissions_AlwaysTrue` e la classe `LoggedUserHasWritePermissions_AlwaysFalse`.

```
internal class LoggedUserHasWritePermissions_AlwaysTrue :
    ILoggedUserHasWritePermissions
{
    public bool CanWrite()
    {
        return true;
    }
}
```

```
internal class LoggedUserHasWritePermissions_AlwaysFalse :
    ILoggedUserHasWritePermissions
{
    public bool CanWrite()
    {
        return false;
    }
}
```

Listato 14: Due implementazioni fake del servizio per l'autorizzazione

Nel progetto `CompositionRoot`, all'interno del file `CustomBindings.cs`, aggiungiamo la seguente regola di binding di seguito a quella già esistente.

```
container.Register<
  DomainModel.Services.ILoggedUserHasWritePermissions,
  Persistence_FakeDatabase
          .LoggedUserHasWritePermissions_AlwaysFalse>();
```

Listato 15: La regola di binding per l'associazione del servizio di autorizzazione

Andiamo ad utilizzare il servizio nell'handler attraverso l'iniezione della dipendenza. La classe viene modificata come segue.

```
public class WritePermissionAuthorizer :
    ICommandAuthorizer<AddUserCommand>
{
    private readonly ILoggedUserHasWritePermissions
        loggedUserHasWritePermissions;

    public WritePermissionAuthorizer(
      ILoggedUserHasWritePermissions
          loggedUserHasWritePermissions)
    {
        this.loggedUserHasWritePermissions =
          loggedUserHasWritePermissions ??
          throw new ArgumentNullException(
            nameof(loggedUserHasWritePermissions));
    }

    public IEnumerable<AuthorizationResult>
        Authorize(AddUserCommand command)
    {
        if (!this.loggedUserHasWritePermissions.CanWrite())
            yield return new AuthorizationResult("User does
                not have write permissions");
    }
}
```

Listato 16: La classe `Authorizator` completamente implementata

L'esecuzione del Command genera un errore di autorizzazione. Se nella classe `CustomBindings` si seleziona l'implementazione `LoggedUserHasWritePermissions_AlwaysTrue` e si riesegue il Command, questa volta l'Action va a buon fine. In un'implementazione reale, quel servizio dovrebbe acquisire l'identità dell'utente autenticato — eventualmente utilizzando un ulteriore servizio da iniettare in se stesso — accedere realmente ad un database e verificare la presenza dell'autorizzazione.

Per realizzare un'altra regola di autorizzazione, come sempre, è sufficiente aggiungere un'altra classe simile all'authorizer appena scritto nella stessa cartella della prima.

In questo esempio è stato messo in evidenza anche come la *composition root* può selezionare un'implementazione specifica a fronte di un servizio. La scelta può essere statica, come nel caso descritto. Ma può basarsi anche su un parametro di configurazione applicativo. Per esempio, la regola di binding potrebbe essere scritta in questo modo.

```
const bool someConfigurationParameter = true; // this is read,
    for instance, from the appsettings.json
if (someConfigurationParameter)
  container.Register<
    DomainModel.Services.ILoggedUserHasWritePermissions,
    Persistence_FakeDatabase
      .LoggedUserHasWritePermissions_AlwaysTrue>();
else
    container.Register<
      DomainModel.Services.ILoggedUserHasWritePermissions,
      Persistence_FakeDatabase
        .LoggedUserHasWritePermissions_AlwaysFalse>();
```

Listato 17: La selezione di una tra due implementazioni per uno stesso servizio

Vi è anche la possibilità di far dipendere la regola di binding da una condizione dinamicamente valutata al tempo di esecuzione, dipendente per esempio dal risultato della chiamata ad un servizio terzo. Tale approccio, utilizzato sistematicamente per ognuna delle "condizioni al contorno" esistenti, può donare grande flessibilità all'applicativo nel suo complesso.

5.1.6 L'aggiunta di un notifier

In seguito all'esecuzione di un Command potrebbe essere opportuno generare delle notifiche o scatenare comunque delle azioni. Eccone alcuni casi:

- la modifica di un dato ne richiede la reindicizzazione da parte di un motore full-text;
- l'esecuzione del Command deve essere notificata ai client, oppure agli altri server di un cluster applicativo, mediante un middleware orientato alla distribuzione di messaggi;

- il verificarsi di un evento deve produrre l'invocazione ad un web service per dare seguito all'esecuzione di un workflow.

In questi casi, piuttosto che annegare questa logica all'interno dell'handler del Command, è possibile trattare la funzionalità come un *aspetto*, realizzando una classe notifier.

Supponiamo di aver incapsulato il servizio di messaggistica con la seguente semplice interfaccia, che come sempre appartiene al `DomainModel`, e dovrà essere opportunamente realizzata da un progetto infrastrutturale.

```
public interface IMessageBus
{
    void Send(string topic, string message);
}
```

Listato 18: Il servizio `IMessageBus`

Si procede allora creando una classe come la seguente.

```
public class AddUserNotifier : ICommandNotifier<AddUserCommand>
{
    private readonly IMessageBus messageBus;

    public AddUserNotifier(IMessageBus messageBus)
    {
        this.messageBus = messageBus ?? throw new
            ArgumentNullException(nameof(messageBus));
    }

    public void Notify(AddUserCommand command)
    {
        this.messageBus.Send("userTopic", $"Added user {
            command.Username }");
    }
}
```

Listato 19: Una classe `Notifier`

Anche in questo caso abbiamo utilizzato la *constructor injection* per consentire alla classe notifier di utilizzare il servizio di messaggistica. Tale servizio deve essere implementato, possibilmente all'interno di un progetto infrastrutturale. Una volta creato il progetto nella cartella `Implementations` della solution, ecco una possibile implementazione fake del servizio, che scrive su log il messaggio trasmesso.

```
internal class MessageBus : IMessageBus
{
    public void Send(string topic, string message)
    {
        Log.Information($"[MessageBus] Topic: { topic } -
            Message: { message }");
```

```
    }
}
```

Listato 20: Un'implementazione fake del servizio `IMessageBus`

Mediante la *composition root* è possibile collegare l'interfaccia a questa implementazione.

```
container.Register<DomainModel.Services.IMessageBus,
    MessageBus_Fake.MessageBus>();
```

Listato 21: La registrazione del servizio `IMessageBus`

Questi passi sono del tutto analoghi a quanto già fatto in precedenza con il servizio di accesso al database (vedi Listato 8).

Come per i validator e gli authorizer, la sola creazione di questa classe ne produce direttamente l'inserimento nella catena di esecuzione del Command.

Con l'introduzione della funzionalità di notifica si concludono i passi per la gestione completa di un Command, costituito da tutti i blocchi funzionali disponibili in RockAPI. Vediamo nel seguito come è possibile procedere alla creazione di una Action di tipo Query.

5.2 Creare una Query

La creazione di un'Action di tipo Query appare molto simile a quella utile alla creazione di un Command. La maggiore differenza sta nel fatto che la Query restituisce un valore e, nel farlo, non deve alterare lo stato del sistema[9].

Vediamo ora come è possibile creare una Query che restituisca l'elenco degli utenti presenti nel sistema, cioè tutti quelli aggiunti con l'utilizzo del Command `AddUser`. Chiameremo la Query `GetUsers`.

Si procede creando una cartella `GetUsers` all'interno del progetto `DomainModel` al percorso `CQRS/Queries`. Nella cartella si crea innanzitutto il DTO di input, che chiamiamo `GetUsersQuery` e che appare come segue.

```
public class GetUsersQuery : IQuery<GetUsersQueryResult>
{
}
```

Listato 22: Il DTO di input della Query `GetUsers`

Il DTO di input è una classe vuota. La Query non viene infatti alimentata da alcuna informazione in ingresso. Come si può notare, questa volta il DTO di input discende da un'interfaccia generica che è istanziata con la classe `GetUsersQueryResult`, che è la classe del DTO di output. Andiamo a crearla come segue nella stessa cartella.

[9] La non alterazione dello stato del sistema deve essere garantita dallo sviluppatore; RockAPI non effettua alcun controllo su questo aspetto, né preclude allo sviluppatore la possibilità di poter effettuare qualsiasi tipo di elaborazione.

```
public class GetUsersQueryResult
{
    public GetUsersQueryResult(IEnumerable<string> users)
    {
        Users = users ?? throw new
            ArgumentNullException(nameof(users));
    }

    public IEnumerable<string> Users { get; }
}
```

Listato 23: Il DTO di output della Query `GetUsers`

È il momento di creare l'handler della Query. Eccolo qua.

```
public class GetUsersQuerydHandler :
    IQueryHandler<GetUsersQuery, GetUsersQueryResult>
{
    public GetUsersQueryResult Handle(GetUsersQuery query)
    {
        throw new NotImplementedException();
    }
}
```

Listato 24: Lo scheletro dell'handler della Query `GetUsers`

L'handler della Query è una classe generica istanziata su entrambi i tipi del DTO di input ed output. Il suo metodo `Handle()` riceve un DTO di input e restituisce un DTO di output. Al momento il codice supera correttamente la compilazione.

Perché l'handler possa fare bene il suo lavoro, ha bisogno di dipendere da un servizio (di dominio) che restituisce le username di tutti gli utenti precedentemente aggiunti. Creiamo l'interfaccia di questo servizio, che chiameremo `IGetUsers`, all'interno della cartella `Services` sempre nel progetto `DomainModel`.

```
public interface IGetUsers
{
    IEnumerable<string> Get();
}
```

Listato 25: Il servizio di dominio `IGetUsers`

Iniettiamo la dipendenza da questo servizio nell'handler appena creato e, di conseguenza, scriviamo l'implementazione del metodo `Handle()` nella Query. La classe appare come segue.

```
public class GetUsersQuerydHandler :
    IQueryHandler<GetUsersQuery, GetUsersQueryResult>
{
    private readonly IGetUsers getUsers;
```

```
    public GetUsersQuerydHandler(IGetUsers getUsers)
    {
        this.getUsers = getUsers ?? throw new
            ArgumentNullException(nameof(getUsers));
    }

    public GetUsersQueryResult Handle(GetUsersQuery query)
    {
        var users = this.getUsers.Get();
        return new GetUsersQueryResult(users);
    }
}
```

Listato 26: L'handler della Query `GetUsers`

Si noti come il metodo `Handle()` in questo caso non utilizzi nella sua implementazione il DTO di ingresso, che peraltro è vuoto. Ancora una volta il codice compila correttamente.

Procediamo ora alla creazione del controller, che espone la Query appena creata all'utente dell'API REST. Come per il Command, il controller può essere creato nel progetto `RockAPI` all'interno della cartella controllers, dove è già presente il controller `AddUsersController`. Ecco come appare.

```
[Route("api/[controller]")]
[ApiController]
public class GetUsersController : ControllerBase
{
    private readonly IQueryHandler<GetUsersQuery,
        GetUsersQueryResult> handler;

    public GetUsersController(IQueryHandler<GetUsersQuery,
        GetUsersQueryResult> handler)
    {
        this.handler = handler ?? throw new
            ArgumentNullException(nameof(handler));
    }

    // GET: api/GetUsers
    [HttpGet]
    public GetUsersQueryResult Get([FromQuery] GetUsersQuery
        query)
    {
        return this.handler.Handle(query);
    }
}
```

Listato 27: Il controller dell'action `GetUsers` con il suo metodo GET

Il Controller, ancora una volta, non fa altro che delegare all'handler l'esecuzione dell'Action, iniettandola mediante *constructor injection*. Questo codice

compila correttamente, mentre l'esecuzione genera immediatamente un errore di runtime grazie al metodo `Verify()` della libreria SimpleInjector. L'errore ci avverte che il controller non può essere creato poiché nel grafo di composizione degli oggetti vi è un servizio che non è stato risolto con un'implementaione: `IGetUsers`. Procediamo con l'implementazione che, per coerenza con il servizio `IAddUser`, realizzeremo nel progetto infrastrutturale `Persistence_FakeDatabase`. Siccome la classe `AddUser` già contiene il database degli utenti, e la classe `GetUsers` che andiamo ad implementare deve accedere allo stesso database, per semplicità faremo sì che la classe `AddUser` implementi anche l'interfaccia `IGetUsers`. È opportuno però modificarne il nome, utilizzando le funzionalità di rifattorizzazione del codice dell'ambiente di sviluppo, e chiamandola `UserFakeDatabase`. Dopo la modifica al nome e l'implementazione del servizio `IGetUsers`, la classe appare come segue.

```
internal class UserFakeDatabase : IAddUser, IGetUsers
{
    private readonly IList<User> users = new List<User>();

    public void Add(User user)
    {
        this.users.Add(user);
    }

    public IEnumerable<string> Get()
    {
        return this.users.Select(u => u.Username);
    }
}
```

Listato 28: L'implementazione `AddUser` del Listato 7 cambia nome e implementa i due servizi di accesso ai dati

Aggiungiamo nella *composition root* la regola di dependency injection che associa il servizio alla sua implementazione.

```
container.Register<DomainModel.Services.IGetUsers,
    Persistence_FakeDatabase.UserFakeDatabase>();
```

Listato 29: La registrazione del servizio `IGetUsers`

Per procedere all'esecuzione della Query, inviamo una richiesta di tipo `GET` con `Content-Type: application/json` al seguente indirizzo.

```
http://localhost:53062/api/GetUsers
```

L'esecuzione della Query va a buon fine e presenta una lista di utenti vuota. La cosa che potrebbe sorprendere è che la lista degli utenti resta vuota anche nel caso in cui facciamo precedere una chiamata `AddUser` ad una chiamata `GetUsers`. Questo in realtà accade perché il lifestyle di default che viene utilizzato da SimpleInjector qui è `Lifestyle.Scoped`: una nuova istanza

della classe database viene creata per ogni invocazione di una action dell'API REST e ad ogni nuova chiamata il database è vuoto.

Una possibile soluzione a questo problema potrebbe essere quello di dichiarare l'attributo `Users` della classe `UserFakeDatabase` con il qualificatore `static`. In questo modo, la lista avrebbe un'unica istanza creata all'atto della prima istanziazione e tale unica istanza verrebbe condivisa tra tutte le istanze della classe `UserFakeDatabase` a mano a mano create. Questa soluzione però ha lo svantaggio di delegare alla classe stessa parte della gestione del suo ciclo di vita, forzandone per sempre la sua natura di singleton.

Una migliore soluzione consiste nel trarre vantaggio dalle potenzialità del paradigma della dependency-injection, delegandole anche il problema del ciclo di vita delle istanze create. Ciò si ottiene modificando come segue le regole di binding dei servizi `IAddUser` e `IGetUsers`.

```
internal static void Bind(Container container)
{
    var fakeDatabase = new
        Persistence_FakeDatabase.UserFakeDatabase();
    container.Register<DomainModel.Services.IAddUser>(() =>
        fakeDatabase);
    container.Register<DomainModel.Services.IGetUsers>(() =>
        fakeDatabase);

        // ... qui il resto delle regole
}
```

Listato 30: La registrazione dei servizi `IAddUser` e `IGetUsers` con un singleton

Come è possibile notare, è il container ora a creare una istanza della classe `UserFakeDatabase`, fornendo quella sola istanza per tutte le richieste di risoluzione di entrambi i servizi che implementa: `IAddUser` e `IGetUsers`. Così facendo, però, in linea di principio resta la possibilità di creare con semplicità un numero arbitrario di istanze.

A questo punto si può verificare che eseguendo in sequenza le due azioni esposte dall'API REST, il sistema risponde coerentemente con tutti gli utenti aggiunti fino a quel momento.

5.2.1 Validation e Authorization per una Query

L'aggiunta di stage di Validation ed Authorization per le Query può essere svolta in maniera sostanzialmente analoga a quella dei Command. Si noti invece come, a differenza dei Command, non è previsto uno stage di notification per le Query.

5.3 Una considerazione sui DTO

I DTO creati per convogliare informazioni verso l'handler — input DTO — e fuori dall'handler — output DTO — vengono utilizzati esattamente così come

sono anche dai controller. Le firme dei metodi `Get()` e `Post()` dei due controlli realizzati sono in pratica equivalenti a quelle dei metodi `Handle()` dei rispettivi handler. Questa caratteristica è in generale spesso verificata. Il controller, infatti, agisce esclusivamente come componente di puro interfacciamento tra l'utente e l'handler dell'Action invocata. Come tale, non dovrebbe introdurre alcuna trasformazione nelle informazioni, poiché una trasformazione sarebbe pur sempre sintomo dell'esistenza di una logica applicativa; e la logica di dominio appartiene più correttamente al modello di dominio. In ogni caso, è talvolta conveniente derogare a questa rigida separazione delle responsabilità e consentire che il controller realizzi delle semplici trasformazioni — cioè dei mapping — tra il dato in ingresso dal controller e il DTO che arriva all'handler, e viceversa per il DTO di output. Ad esempio, è comodo delle volte utilizzare come DTO delle classi di dominio pure, facendo sì che sia il controller a risagomare questa classe verso il dato che dovrà essere trasmesso all'utente dell'API REST.

6 Come affrontare modifiche implementative

La dependency injection favorisce il rispetto del principio *"program to an interface, not an implementation"*. Questo consente di realizzare programmi altamente flessibili, conservando nel contempo elevati livelli di affidabilità in caso di modifiche e estensioni. In questo paragrafo vedremo concretamente come è possibile modificare profondamente l'implementazione di un servizio e rendere perfino possibile la selezione di una tra due differenti implementazioni, anche molto differenti tra loro, attraverso una minima modifica limitata ad un singolo punto del codice sorgente.

Il servizio di accesso ai dati è attualmente realizzato mediante uso di un database fake che salva i propri dati in memoria. Vogliamo realizzarne una seconda implementazione basandoci su un database reale, utilizzando la tecnologia database MongoDB [7]. Ecco come si procede.

Si inizia con la creazione di un nuovo progetto infrastrutturale, quindi all'interno della solution folder `Implementation`, che chiameremo `Persistence_MongoDB`. Il progetto creato è di tipo *net core library*. Introduciamo un riferimento alla libreria che contiene il driver di MongoDB: `MongoDB.Driver`.

```
PM> Install-Package MongoDB.Driver
```

Listato 31: Il comando per l'installazione del package MongoDB.Driver

All'interno del progetto creiamo una classe `Database`, definita come segue[10].

[10] Questo file contiene, da solo, tutta la logica per l'inizializzazione della connessione al database, il mapping delle classi di dominio e l'implementazione dei servizi `IAddUser` e `IGetUsers`. La classe `User` è mappata con l'attributo `Username` che rappresenta la chiave primaria della collezione degli utenti. Questa non è una pratica consigliata poiché sarebbe più opportuno suddividere su più classi le varie responsabilità. Qui useremo l'approccio mostrato per ridurre al minimo il codice da scrivere ed il numero di classi necessarie. Ciò che si vuole mettere in evidenza e la facilità con cui una nuova implementazione del servizio può essere utilizzata al posto di una già esistente.

```
[assembly: InternalsVisibleTo("CompositionRoot")]

namespace Persistence_MongoDB
{
    internal class Database : IAddUser, IGetUsers
    {
        private IMongoCollection<User> usersCollection = null;

        public Database()
        {
            if (this.usersCollection == null)
                this.initializeDbConnection();
        }

        public void Add(User user)
        {
            this.usersCollection.InsertOne(user);
        }

        public IEnumerable<string> Get()
        {
            return this.usersCollection.Find(_ => true)
                .ToEnumerable()
                .Select(u => u.Username);
        }

        internal void initializeDbConnection()
        {
            BsonClassMap.RegisterClassMap<User>(cm =>
            {
                cm.AutoMap();
                cm.MapIdMember(c => c.Username);
            });

            var client = new MongoClient();
            var database = client.GetDatabase("rockAPI");
            this.usersCollection =
                database.GetCollection<User>("users");
        }
    }
}
```

Listato 32: L'implementazione dei servizi `IAddUser` e `IGetUsers` con MongoDB

Aggiungiamo al progetto `CompositionRoot` il riferimento al progetto `Persistence_MongoDB`. Modifichiamo le regole di binding dei servizi `IAddUser` e `IGetUsers`, che attualmente puntano all'implementazione fake del database, facendole puntare alla nuova implementazione. Le nuove regole appariranno

così.

```
var mongoDatabase = new Persistence_MongoDB.Database();
container.Register<DomainModel.Services.IAddUser>(() =>
    mongoDatabase);
container.Register<DomainModel.Services.IGetUsers>(() =>
    mongoDatabase);
```

Listato 33: La registrazione dei servizi `IAddUser` e `IGetUsers` con un singleton

L'esecuzione dell'applicativo richiede un'istanza di MongoDB attiva in ascolto sull'indirizzo di default `mongodb://localhost:27017` senza alcuna autenticazione attiva. Per questo, è sufficiente scaricare il pacchetto MongoDB Community Edition, installarlo, creare la cartella `C:\data\db` ed avviare il servizio `mongod`.

L'applicativo in fase di esecuzione si comporta quasi allo stesso modo della versione precedente, che ospitava le sue informazioni in memoria. Una prima differenza è che non potranno essere inseriti due utenti aventi la stessa username, poiché la username è stata impostata come chiave primaria della collezione appena creata. La seconda differenza sta nel fatto che questa volta i dati sugli utenti, essendo registrati in un vero database, persisteranno e sopravviveranno indefinitamente.

Per poter agilmente selezionare una tra le due implementazioni disponibili dei servizi `IAddUser` e `IGetUsers`, le regole di binding dei servizi possono essere modificate come segue.

```
bool useFakeDatabase = false;
if (useFakeDatabase)
{
    var fakeDatabase = new
        Persistence_FakeDatabase.UserFakeDatabase();
    container.Register<DomainModel.Services.IAddUser>(() =>
        fakeDatabase);
    container.Register<DomainModel.Services.IGetUsers>(() =>
        fakeDatabase);
}
else
{
    var mongoDatabase = new Persistence_MongoDB.Database();
    container.Register<DomainModel.Services.IAddUser>(() =>
        mongoDatabase);
    container.Register<DomainModel.Services.IGetUsers>(() =>
        mongoDatabase);
}
```

Listato 34: Due diverse implementazioni possono essere iniettate in base al valore di un parametro

Variando il valore del parametro `useFakeDatabase` si seleziona una tra le due implementazioni disponibili. In questo caso è utile rispettare il principio di

sostituzione di Liskov per garantire che non vi siano impatti sul funzionamento dell'applicazione in seguito allo scambio. Il valore del parametro può essere anche attinto dal file di configurazione dell'applicativo oppure dinamicamente calcolato attraverso apposite funzioni applicative. In questo modo l'applicazione guadagna in flessibilità divenendo capace di reagire al tempo di runtime alle mutevoli condizioni al contorno.

Nel semplice esempio mostrato, l'*implementation switch* è effettuato mediante variazione di due sole associazioni interfaccia-implementazione. Nei casi reali, un servizio è implementato da una batteria di classi. È opportuno allora creare delle sezioni contenenti gruppi di regole che selezionano coerentemente tutte le classi le quali, nel loro insieme, costituiscono l'implementazione di un servizio composito. E la sostituzione dell'implementazione di un servizio avviene selezionando un set di regole piuttosto che un altro.

L'uso della tecnica mostrata in questo paragrafo è abilitata dall'aver astratto i servizi di dominio con opportune interfacce. Ne deriva un incremento nella versatilità del prodotto, più incline ad assorbire in modo naturale e meno invasivo variazioni che sopraggiungano nelle specifiche. E questo vale anche se tali variazioni giungessero in fase avanzata di sviluppo, eventualità particolarmente sgradita ai team di sviluppo, e tuttavia in linea con uno dei princìpi fondanti del manifesto della programmazione agile [5]:

> *"Welcome changing requirements, even late in development."*

7 Inibire il logging di informazioni riservate

Può capitare che alcune informazioni contenute in un DTO siano riservate e pertanto non sia opportuno che vengano inserite nel log di audit che RockAPI esegue sistematicamente. Prendiamo per esempio il DTO del Command `AddUser` (vedi Listato 5). Per un tale DTO è opportuno evitare che il campo password venga inserito nel log. Un DTO che abbia queste caratteristiche deve implementare l'interfaccia `IHasCustomAudit`. Tale interfaccia definisce il metodo seguente.

```
string SerializeForAudit();
```

Listato 35: Il metodo implementato dall'interfaccia `ICustomAudit`

che deve restituire la stringa da inserire nel log. Il DTO `AddUser` viene pertanto costruito come segue.

```
public class AddUserCommand : IHasCustomAudit
{
    public string Username { get; set; }
    public string Password { get; set; }

    public string SerializeForAudit()
    {
        return this.ToString();
```

```
    }

    public override string ToString()
    {
        return $"{{ Username: \"{ this.Username }\", Password:
            \"***\" }}";
    }
}
```

Listato 36: Il DTO di input del Command `AddUser` dotato di un log personalizzato

Così facendo, la riga inserita nel log appare come segue.

```
2020-01-01 10:00:00.000 [INF] Action starting
    DomainModel.CQRS.Commands.AddUser.AddUserCommand: {
    "Username":"foo", "Password":"***" }
```

Listato 37: Il log personalizzato del DTO di input del metodo `AddUser`

8 Conclusioni

In questo articolo è stata presentata l'architettura software RockAPI, basata su un progetto net core WebApi. Il modello di sviluppo proposto, conforme al pattern CQRS ed ispirato ai principi della programmazione SOLID, solleva il programmatore dal dover implementare una serie di comportamenti comuni, gestiti da RockAPI con approccio *aspect oriented*. Una chiara indicazione sul posizionamento dei moduli applicativi, propone una standardizzazione del processo di sviluppo che aiuta il programmatore ad orientarsi nel codice e facilita un controllo sistematico nel caso di coordinamento di uno sviluppo collaborativo.

RockAPI è stato usato con successo in diversi contesti di sviluppo software condotti da personale esperto in tecnologia Microsoft con il linguaggio `C#`. A fronte di una breve attività di formazione sull'architettura del progetto, si sono evidenziati i seguenti vantaggi:

- i gruppi sono diventati ben presto capaci di dominare i meccanismi, acquisendo progressivamente autonomia di utilizzo del framework;
- si è notato un incremento nella produttività del gruppo per via di un'architettura software che gestisce con approccio convention-based gli aspetti della validazione, del logging, del performance monitoring, dell'auditing, dell'autorizzazione e delle notifiche;
- è migliorato il coordinamento da parte dei project manager sulle fasi dello sviluppo, rendendo più semplice orientarsi nell'architettura applicativa e fornire ai team il necessario supporto per la risoluzione di problemi;
- la qualità del prodotto è stata soddisfacente fin dalle prime fasi di rilascio del software, limitando fortemente il tempo di convergenza verso la stabilità applicativa.

8.1 Le tecnologie utilizzate

RockAPI fa uso di alcune librerie molto utili ad abilitare le funzionalità descritte.

- **SimpleInjector**: una libreria potente e ben documentata per integrare la Dependency Injection all'interno del framework WebApi. L'inversione di dipendenza è un pattern estremamente potente, spesso sottovalutato o frainteso. In [3] i concetti sull'inversione di dipendenza vengono chiaramente descritti. La libreria ha un esteso supporto per i generics. Tale supporto è stato intensamente usato per l'implementazione di RockAPI. Un'altra utile funzionalità è legata all'esistenza del metodo `Verify()`, presente all'interno del progetto `RockAPI` nella classe `Startup.cs`. Questo metodo, all'atto dell'invocazione, visita tutte le classi controller per verificare la loro corretta istanziazione. Il successo di questa attività è la prova che:

  - siano state effettivamente configurate tutte le associazioni tra i servizi astratti (le interfacce) e le relative implementazioni;
  - non vi siano regole di associazioni multiple per uno stesso servizio, le quali genererebbero ambiguità in fase di istanziazione dei servizi;
  - non vi siano incoerenze nel lifestyle impostato sulle regole.

  Tutte queste verifiche danno una preziosa garanzia sul fatto che eventuali errori di configurazione della *composition root* vengano evidenziati immediatamente all'avvio dell'applicativo (quindi quasi al tempo di compilazione) e non solo al run-time, a valle della distribuzione del software.

- **Serilog**. Una libreria versatile e moderna per realizzare il logging nelle applicazioni net core. Compatibile anche con approcci di logging centralizzato, propedeutiche alla realizzazione di applicazioni cloud native.

- **NUnit**. Una utile libreria per lo unit testing disponibile in ambiente net core.

8.2 Ringraziamenti

Si ringraziano tutti i colleghi e gli ex-colleghi che hanno riposto fiducia in RockAPI, impegnandosi a comprenderne i meccanismi ed utilizzandolo professionalmente. Queste esperienze hanno consolidato una consapevolezza sui vantaggi derivanti dal suo utilizzo, facendo anche emergere problematiche che hanno consentito un perfezionamento di alcuni aspetti dell'architettura di RockAPI.

Si ringrazia Steven van Deursen perché con i suoi articoli, il suo libro [3] e l'eccellente libreria Simple Injector [6] ha ispirato questo lavoro.

Si ringrazia Simon Pietro Romano (`@spromano`) per la review ed i suoi utili consigli.

Riferimenti bibliografici

[1] https://github.com/supix/rockapi

[2] Hohpe, G. and Woolf, B.: *Enterprise Integration Patterns*, Addison Wesley, 2004, ISBN: 0-321-20068-3.

[3] Seemann, M. and van Deursen, S.: *Dependency Injection Principles, Practices, and Patterns*, Manning Publications, 2019, ISBN: 9781617294730.

[4] https://en.wikipedia.org/wiki/Aspect-oriented_programming

[5] https://agilemanifesto.org/principles.html

[6] https://simpleinjector.org

[7] https://www.mongodb.com

[8] https://serilog.net

[9] https://nunit.org