ALMA MATER STUDIORUM - UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

SVILUPPO DI UN PORTALE PER GESTIRE LA FORMAZIONE DEL PERSONALE CON APPROCCIO A MICROSERVIZI

Elaborato in Tecnologie Web

Relatore

Prof. BRAVETTI MARIO

Presentata da

PERUGINI LEONARDO

Correlatore

GREGORI CESARE

Anno Accademico 2018/2019

Indice

Introduzione

Le moderne tecniche di progettazione e sviluppo di sistemi software orientati al web hanno visto l'affermazione di modalità operative che prevedono una sempre maggiore adattabilità e flessibilità. Sistemi di questo tipo devono essere composti da tanti componenti interdipendenti, ciascuno dei quali offre funzionalità ben delineate. In questo contesto emerge come vero e proprio pattern architetturale l'approccio a microservizi [1] [2] [3], cioè servizi piccoli e autonomi. Questa architettura è fortemente influenzata da metodi di sviluppo "agile", nonché dalla virtualizzazione e dal cloud, al fine di rilasciare software con il più alto valore per i clienti con la maggiore frequenza possibile. In particolare, le tecnologie di sviluppo in questione devono prevedere meccanismi automatici di deployment che consentano la continuous integration [5], per fare in modo che il software possa essere rilasciato nell'ambiente di produzione in ogni momento e in modo automatico, e tecniche avanzate di gestione automatica dello scaling dei servizi a runtime [7], al fine di poter avere un bilanciamento automatico delle risorse in base all'utilizzo o alla mole di richieste che giungono al sistema.

Nella mia tesi, svolta presso l'azienda FastCode Consulting Srl di Cesena, mi sono occupato inizialmente della riprogettazione dell'applicazione aziendale WorkManager adottando l'architettura a microservizi, per poi iniziarne lo sviluppo implementando un microservizio che consentisse di gestire i corsi aziendali e quindi la formazione del personale.

Struttura della tesi

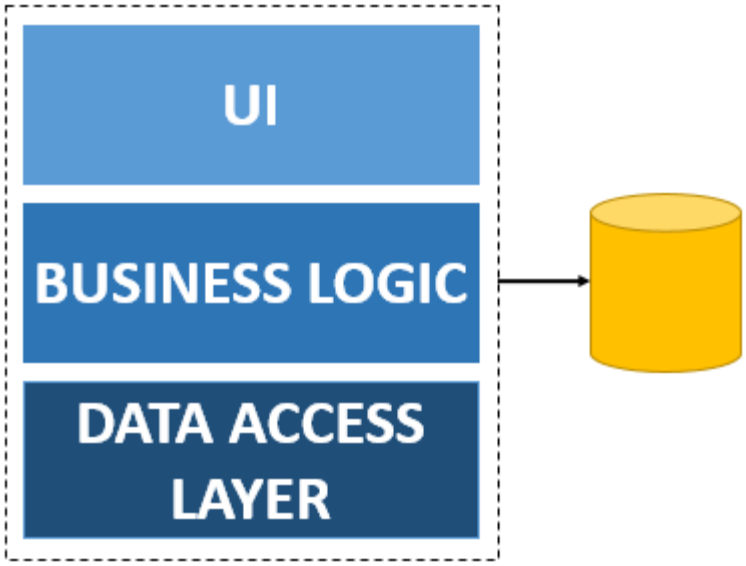
- Nel capitolo 1 verranno descritte le tecnologie, programmi e framework utilizzati per lo sviluppo del progetto, nonché approcci e meccanismi integrabili in futuro. In particolare verranno introdotte la struttura monolitica, il pattern architetturale a microservizi e il framework ASP.NET Core MVC [14].
- Nel capitolo 2 si parlerà dell'applicazione aziendale nella sua versione iniziale analizzando la sua struttura monolitica, le parti che la compongono ed i servizi che utilizza, concentrandosi prevalentemente sul modulo di gestione dei corsi.
- Nel capitolo 3 si parlerà della struttura del sistema in seguito alla riprogettazione, evidenziando benefici e criticità. Successivamente verrà presentato il microservizio di gestione della formazione implementato, analizzando le modifiche che sono state apportate in termini strutturali e di funzionalità.

Capitolo 1 - Tecnologie e architetture

In questo capitolo verranno introdotte in sequenza le principali tecnologie e architetture utilizzate per lo sviluppo del progetto di tesi. Come prima cosa verrà presentata una panoramica sugli applicativi monolitici e le loro caratteristiche, dopodiché verrà introdotta l'architettura a microservizi e successivamente i framework ASP.NET Core MVC e Angular2, utilizzati rispettivamente per lo sviluppo delle API lato server e lato client. Si parlerà anche di Continuous Integration e Autoscaling come meccanismi integrabili. Infine verranno presentati brevemente il broker di messaggi RabbitMQ e l'applicativo di versioning SVN.

1.1 – Panoramica sulle applicazioni monolitiche

L'architettura monolitica è l'approccio di sviluppo convenzionale e storicamente più utilizzato. La sua struttura prevede che l'applicazione sia complessivamente costruita come una singola unità: un'interfaccia utente e un'applicazione lato server costituita da vari moduli che operano con i dati forniti dall'utente e applicano modifiche al database che li raccoglie [1].



L'architettura monolitica si presta bene alla realizzazione di piccole applicazioni o comunque applicazioni poco soggette a cambiamenti. Se le dimensioni delle applicazioni sono ridotte questo rappresenta sicuramente un valido approccio dato che lo sviluppo è semplice e avviene utilizzando la medesima tecnologia per ognuna delle funzionalità. In più anche le fasi di test e distribuzione risultano semplici e immediate. Diverso discorso deve esser fatto per applicazioni

di maggiore dimensione e complessità o che necessitano di continue evoluzioni, fattori che rendono più complicate tutte le fasi di sviluppo dell'applicazione.

VANTAGGI	SVANTAGGI
Semplice da sviluppare una volta raccolti requisiti e scelti linguaggi e tecnologie.	Limitazioni in termini di grandezza e complessità
Semplice da testare	Un update richiede un deploy completo
Semplice effettuare il deploy in produzione	Un singolo bug o una modifica può scatenare effetti a catena su tutta l'applicazione
Semplice effettuare uno scaling orizzontale effettuando più copie su server.	Un cambio di tecnologia o framework è complesso in quanto stravolge l'intera applicazione.

Nella tabella vengono riassunti sommariamente i principali vantaggi e svantaggi dell'approccio architetturale monolitico. Per capire la complessità di un'applicazione di questo tipo basta immedesimarsi in un nuovo sviluppatore che entra nel team e che ha bisogno di imparare il funzionamento dell'intera applicazione, indipendentemente da quello che deve sviluppare. Inoltre il vantaggio di avere un semplice scaling orizzontale rappresenta allo stesso tempo anche un difetto, in quanto la replica dell'intera applicazione è il solo e unico metodo di distribuzione possibile: è palese che questo, all'aumentare di dimensioni e complessità, comporta un diretto aumento di costi e risorse necessari [2].

1.2 – Teoria dei microservizi

Le applicazioni che presentano una struttura monolitica costituiscono il punto di partenza del processo evolutivo delle architetture che poi giunge fino a quella a microservizi. Sebbene inizialmente tutte le applicazioni presentassero questa struttura, con il passare del tempo, ma soprattutto con l'accrescersi delle dimensioni delle applicazioni e delle esigenze di modularità, si è passati allo studio di architetture orientate ai servizi. Da queste sperimentazioni è emersa, come un trend o un pattern, l'architettura a microservizi che riscontra in Amazon, Netflix, Spotify, ecc.... alcuni dei casi di maggior successo. Questa architettura è quindi un pattern architetturale per lo sviluppo di una singola applicazione come un insieme di microservizi, cioè dei servizi piccoli e autonomi eseguiti come processi distinti e che lavorano insieme

comunicando mediante meccanismi leggeri, come lo scambio di messaggi [3]. Questo pattern può essere definito application architecture in quanto è applicabile alle singole applicazioni, e per questo si differenzia dalle architetture orientate a servizi – SOA, Service Oriented Architecture – che vengono definite enterprise architecture, cioè applicate a tutte le applicazioni e i sistemi di un'organizzazione.

Le applicazioni monolitiche come venivano intese convenzionalmente, mediante questo pattern, vengono quindi suddivise in microservizi, ognuno rappresentativo di una singola funzionalità. L'intera applicazione in produzione sarà costituita da molti microservizi, in alcuni casi anche centinaia o migliaia, quindi bisogna osservare che ogni microservizio deve essere in grado, anche indirettamente, di coordinarsi con gli altri microservizi e non deve ovviamente compromettere il funzionamento di altri microservizi né del sistema nel suo complesso. A questo proposito è importante anche il fattore dell'integrità dei dati: è importante, soprattutto se si sceglie di utilizzare uno o più database condivisi tra i microservizi, la coordinazione tra i diversi microservizi che agiscono sulla stessa base dati. Questa eventualità non dovrebbe però verificarsi in un sistema a microservizi ben progettato, in quanto ognuno dei microservizi dovrà agire solo su una sottoparte del database che utilizza o, nella migliore delle ipotesi, in un database proprio. Nell'applicativo sviluppato in questa tesi si è scelto di adottare un database condiviso, con il microservizio sviluppato che reperisce dati da tabelle utilizzate anche da altre API, ma che è destinato ad effettuare operazioni di modifica su tabelle utilizzate solo da esso.

In alcuni casi si può inoltre prevedere una cache interna al microservizio contenente solo i dati necessari all'applicazione, per poter velocizzare i tempi di risposta per i messaggi che non cambiano o per fornire una risposta come alternativa ad un guasto.

La caratteristica fondamentale dei microservizi, però, è quella di operare completamente in modalità stateless, cioè senza stato o contesto. Sostanzialmente il microservizio deve essere totalmente 'agnostico' riguardo all'entità che lo contatta, quindi i dati generati in una sessione non vengono mai memorizzati per poi essere riutilizzati successivamente. Per spiegare meglio questa caratteristica ci si può riferire al meccanismo di autenticazione adottato nell'applicazione, che verrà spiegato meglio nel capitolo successivo: in un sistema stateful, cioè con stato, quando viene effettuato il login il sistema memorizza automaticamente lo stato di 'loggato' ed ogni pagina visualizzata successivamente conoscerà questa informazione e verrà memorizzata in sessione. Quindi ogni richiesta successiva al login è influenzata da questa informazione memorizzata. Nei sistemi stateless come quello sviluppato, invece, non si tiene

traccia di nulla che riguardi la sessione ed ogni richiesta deve essere effettuata in maniera indipendente ogni volta. Tornando al caso dell'autenticazione, al momento del login il sistema fornisce un token che viene mantenuto dall'entità richiedente e che deve essere sottoposto (e convalidato) ogni volta ed in ogni richiesta che viene effettuata, visto che il sistema non sa se l'utente è loggato o meno.

I requisiti fondamentali per attuare l'architettura a microservizi sono sostanzialmente tre:

- INDIPENDENCY: ogni microservizio deve essere indipendente dagli altri e distinguibile indipendentemente.
- LOOSE COUPLING: i microservizi di un sistema devono essere scarsamente accoppiati in modo che ogni cambiamento su di essi non influisca a catena sugli altri [4].
- BUSINESS GOAL: ogni unità deve essere la più piccola possibile e in grado di fornire uno specifico obiettivo aziendale.

Ogni microservizio deve quindi avere la propria logica di business e proprie parti interessate comunicando con software di terze parti o, per esigenze specifiche, con gli altri microservizi del sistema.

La tabella sottostante riassume quali sono i pro e i contro dell'architettura a microservizi:

VANTAGGI	
TIME-TO-MARKET PIU' RAPIDO	I cicli di sviluppo sono abbreviati, quindi aggiornamenti e deployment avvengono in maniera più agile
SCALABILITA'	I microservizi possono essere distribuiti su più server e infrastrutture, in base alla domanda e alle esigenze aziendali.
RESILIENZA	Vista l'indipendenza di ogni microservizio, i guasti o blocchi di una singola unità non hanno impatto sull'intero sistema aziendale.
ACCESSIBILITÀ	È molto più semplice comprendere, aggiornare e migliorare i componenti accelerando lo sviluppo.

MAGGIORE APERTURA	È possibile scegliere il linguaggio e la tecnologia ottimale per la funzione da creare.
SVANTAGGI	
COMUNICAZIONE	L'eterogeneità di linguaggi potrebbe comportare una gestione complessa dell'applicazione.
COMPETENZE TECNICHE	Servono competenze specifiche per la gestione di architetture, monitorando costantemente gli aggiornamenti.
TEST	Servono meccanismi molto solidi, un errore che si rivela in un punto dell'applicazione può ripercuotersi su più livelli di distanza.
GESTIONE DELLE VERSIONI	L'aggiornamento ad una nuova versione potrebbe compromettere la retro-compatibilità. Mantenere più versioni può essere un'alternativa ma complica gestione e manutenzione.

Un'altra caratteristica fondamentale dell'architettura a microservizi è che si presta bene ad essere integrata con tecniche di rilascio continuo e scalatura automatica mediante i meccanismi di Continuous Integration e Autoscaling, descritti di seguito.

1.3 – Continuous Integration

Per continuous integration si intende la pratica che si applica in contesti di sviluppo software attraverso un sistema di versioning che consente di automatizzare il deployment automatico di ogni modifica effettuata sull'applicazione, effettuando allineamenti frequenti tra gli ambienti di lavoro degli sviluppatori e l'ambiente condiviso. Questa tecnica si basa su dei solidi princìpi introdotti dall'informatico britannico Martin Fowler [5]:

- Mantenimento di un repository del codice sorgente, fondamentale per l'automatizzazione di build e test.
- Rendere la build automatica, senza bisogno di intervento umano.

- Rendere la build auto-testante, includendo test automatizzati nel processo di compilazione.
- Eseguire commit giornalieri alla mainline, in modo da avere massima sincronizzazione con il codice scritto dagli altri.
- Ogni commit deve far partire una build, in modo da evidenziare subito eventuali errori e mantenere la mainline in uno stato sano.
- Correzione immediata di eventuali errori in fase di build o test in modo da evitare che si creino errori a catena.
- Fare in modo che la build sia veloce, in modo da non costringere gli sviluppatori ad aspettare per verificarne l'esito o effettuare altri commit.
- Esecuzione dei test in un clone dell'ambiente di produzione, evitando la generazione di bug inspiegabili dopo il rilascio in produzione con conseguenti danni.
- Rendere facile l'accesso all'ultimo eseguibile a tutti coloro coinvolti nel progetto.
- Rendere accessibili i risultati dell'ultimo build, in modo da stabilire quali sono i moduli che hanno bisogno di maggiore attenzione.
- Automatizzazione dei rilasci attraverso script che consentono di distribuire facilmente l'applicazione in qualsiasi ambiente.

Tutte queste "best practices" sono mirate a minimizzare le possibilità del cosiddetto "integration hell" [6], cioè l'insieme delle problematiche dovute all'integrazione tra le versioni individuali degli sviluppatori. Con un approccio ad integrazione manuale è molto difficile ipotizzare la durata di questa fase, mentre con l'integrazione continua il processo diventa fluido e in ogni momento è possibile sapere cosa non funziona.

1.4 – Autoscaling

Le tecniche di auto-scaling sono utilizzate nell'ambito del cloud computing e consistono in metodi che permettono la distribuzione automatica delle risorse in base al carico di lavoro delle diverse applicazioni aumentando o diminuendo il numero di istanze, come una sorta di meccanismo di load balancing. Questo meccanismo si contrappone al tradizionale meccanismo di scaling che prevede la gestione manuale di queste situazioni in cui viene deciso a priori il numero di istanze, mentre duplicazioni successive devono avvenire ad opera di un utente addetto. L'utilizzo di tecniche di autoscaling è utile in primis quando il carico delle applicazioni aumenta in maniera tale da rendere necessario un incremento di risorse, ma serve anche per

poter ridurre i costi quando la necessità di risorse è inferiore. Per attuare un buon meccanismo di autoscaling è bene applicarlo secondo il criterio di scaling più attinente alle caratteristiche dell'applicazione tra i seguenti:

- Utilizzo medio della CPU
- Carico HTTP, basato sull'utilizzo o sulle richieste al secondo.
- Metriche di monitoraggio Stackdriver

Per ogni criterio devono essere stabilite le opportune metriche per poter attuare al meglio l'operazione di scaling. Il meccanismo di raccoglie continuamente informazioni sull'utilizzo in base alla politica adottata e confronta l'utilizzo effettivo con l'utilizzo desiderato determinando se le risorse devono essere aumentate o ridotte [7].

1.5 – Microsoft .NET

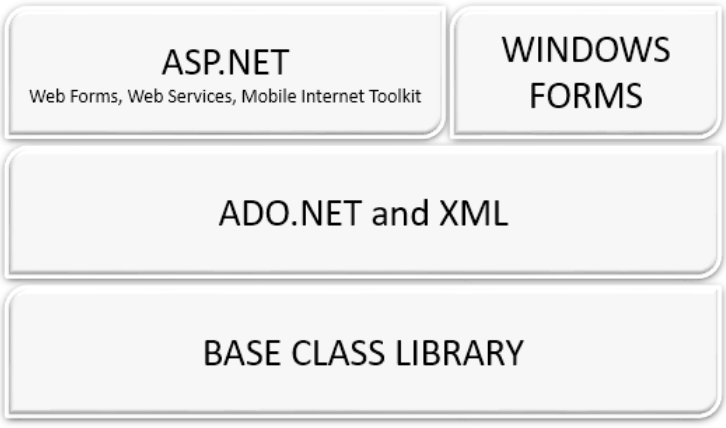
Microsoft .NET è una piattaforma di sviluppo general purpose lanciata da Microsoft nel 2002 inizialmente per lo sviluppo di applicazioni desktop e server, evolvendo poi verso le architetture client/server, internet ed intranet. La sua caratteristica peculiare è di essere indipendente dalla versione di Windows su cui è installata e di includere molte funzionalità progettate espressamente per integrarsi in ambiente internet garantendo massima sicurezza e integrità dei dati [8] [9]. Nella tabella sottostante vengono riassunte le principali caratteristiche della piattaforma .NET:

INTEROPERABILITA'	Consente la compatibilità permettendo di accedere a funzionalità implementate in applicazioni che vengono eseguite all'esterno dell'ambiente .NET
AMBIENTE DI ESECUZIONE COMUNE	Tutti i programmi vengono eseguiti sotto la supervisione del CLR (Common Language Runtime), la macchina virtuale del framework .NET.
INDIPENDENZA DAL LINGUAGGIO	Con l'introduzione del CTS (Common Type System), un sistema di tipi comuni, vengono definiti tutti i diversi tipi e costrutti supportati dal CLR e come essi interagiscono tra loro.

BASE CLASS LIBRARY	Libreria accessibile da tutti i linguaggi della piattaforma .NET che fornisce classi che implementano funzionalità di base riguardanti la gestione dei file, interazione con i database e molto altro.
SICUREZZA	.NET fornisce un modello comune di sicurezza per tutte le applicazioni al fine di affrontare vulnerabilità sfruttate per scrivere codice dannoso.
PORTABILITA'	Il framework .NET è progettato per essere completamente indipendente dalla piattaforma su cui viene eseguito e per permettere alle applicazioni scritte in ambiente .NET di girare su qualsiasi piattaforma in cui esso sia implementato.

Il framework si compone di due elementi principali: CLR e Class Library. Il Class Library è sostanzialmente una vasta collezione di classi, gerarchica ed estensibile, che offre sia funzionalità di base che funzionalità per specifiche tipologie di applicazioni. [9] Le funzionalità della Class Library possono essere suddivise in tre categorie, non definite rigidamente quindi alcune funzionalità possono rientrare in più di una classe:

- Utility features: includono varie classi di Collezioni come liste, stack, code, dizionari, ecc. e anche classi per manipolazioni più varie.
- Wrappers around OS functionality: includono le classi per l'utilizzo del file system, le classi per gestire le funzionalità di rete e quelle per gestire l'I/O delle applicazioni.
- Frameworks: ci sono ari framework disponibili per sviluppare determinate applicazioni. Ad esempio ASP.NET viene utilizzato per sviluppare applicazioni web.



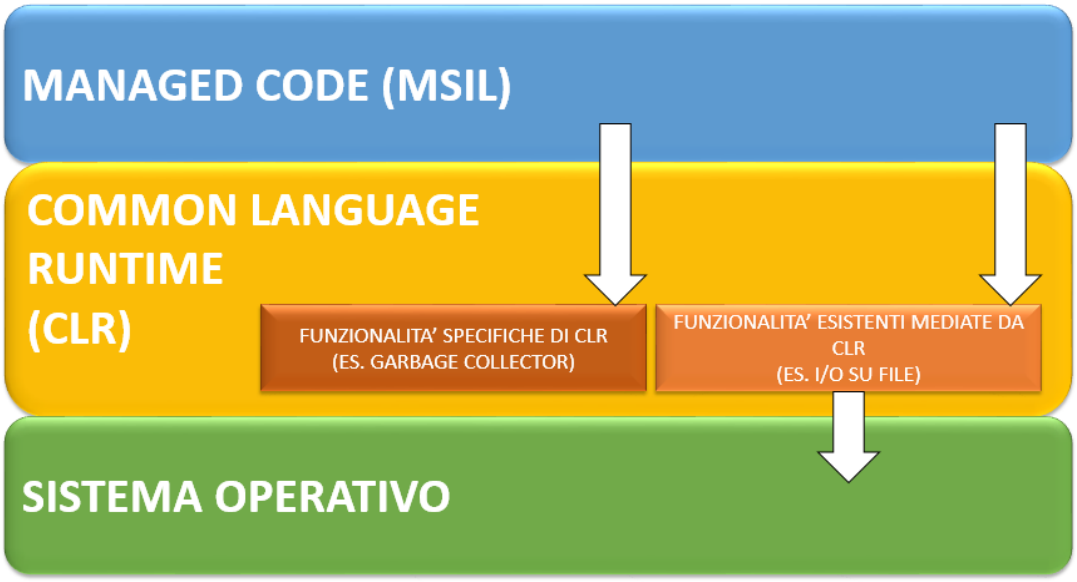
Il CLR – Common Language Runtime, invece è un livello del framework che gestisce l'esecuzione delle applicazioni .NET e le comunicazioni di queste con il S.O. (che non avvengono direttamente ma solo attraverso il CLR stesso) [10]. Questo componente fornisce anche numerosi servizi come sicurezza, gestione delle dipendenze e gestione della memoria oltre a semplificare la stesura del codice, la distribuzione dell'applicazione e migliorare l'affidabilità della stessa. Il CLR è a sua volta formato da cinque componenti principali:

- CTS – Common Type System: sistema di tipi unificato e inter-linguaggio.
- CLS – Common Language Specification: standard a cui qualsiasi linguaggio .NET deve aderire; prevede un sottoinsieme minimo del CTS utile per garantire l'interoperabilità fra linguaggi differenti.
- CIL – Common Intermediate Language: detto MSIL nell'implementazione Microsoft, è un linguaggio indipendente dalla CPU che può essere efficientemente tradotto nel linguaggio macchina di una data CPU.
- JIT – Just In Time Compiler: non tutto il codice CIL di un programma viene sempre eseguito, perciò solo la parte necessaria viene compilata un istante prima della sua esecuzione e memorizzato per successive esecuzioni.
- VES – Virtual Execution System: una macchina virtuale che rappresenta l'ambiente di esecuzione.

Le applicazioni che vengono eseguite vengono dette Managed Applications se queste vengono direttamente eseguite dal CLR, altrimenti vengono dette Unmanaged Applications. Nonostante questo anche le applicazioni di tipo Unmanaged possono "ospitare" al loro interno il .NET framework e quindi chiedere al CLR di eseguire componenti managed. Le Managed Applications sono scritte in MSIL che il CLR è in grado di eseguire offrendo vari servizi [9].

I vantaggi che porta con se il CLR sono i seguenti:

- Ambiente object-oriented: qualsiasi entità è un oggetto, con classi ed ereditarietà pienamente supportati (anche tra linguaggi diversi).
- Riduzioni di comuni errori di programmazione: linguaggi fortemente tipizzati, gestione delle eccezioni e prevenzione di memory leak grazie al garbage collector.
- Indipendenza dal sistema operativo: l'efficienza non viene persa grazie al JIT che ottimizza il codice per la specifica piattaforma.
- Piattaforma multi-linguaggio: i componenti di un'applicazione possono essere scritti con linguaggi diversi.



1.6 – Entity Framework e ADO.NET

Entity Framework è un mapper relazionale a oggetti che consente agli sviluppatori di .NET di usare un database con gli oggetti .NET, supportando lo sviluppo di applicazioni orientate ai dati. Entity Framework permette di lavorare con i dati sotto forma di oggetti senza doversi preoccupare delle tabelle e delle colonne del database su cui sono archiviati i dati. In questo modo gli sviluppatori lavorano ad un livello di astrazione più alto quando si occupano dei dati e possono creare e mantenere applicazioni con meno codice rispetto alle applicazioni tradizionali [11].

Entity framework è basato su ADO.NET, un set di classi che espongono servizi di accesso ai dati per i programmatori, che fornisce un set completo per la creazione di applicazioni distribuite e abilitate alla condivisione dei dati. ADO.NET è parte integrante del .NET Framework e consente l'accesso ai dati relazionali, ai dati XML e ai dati dell'applicazione [12].

1.7 – ASP.NET

ASP.NET è un framework open source per applicazioni web, sottoinsieme del .NET Framework e basato sul CLR. È appositamente progettato per funzionare con http e consente la creazione di pagine Web dinamiche, applicazioni Web, siti Web e servizi Web in quanto fornisce una buona integrazione con HTML, CSS e Javascript [13].

Questo framework è molto popolare tra gli sviluppatori in quanto porta con se svariati benefici:

- È un'estensione del .NET Framework.
- È più veloce degli altri framework disponibili sul mercato.
- Permette di scrivere il codice back-end per l'accesso ai dati e qualsiasi altra logica in C#.
- Permette la creazione di pagine dinamiche grazie a Razor, ma può essere integrato anche con framework quali Angular e React.
- È possibile creare ed eseguire app in diversi sistemi operativi.

1.8 – ASP.NET Core MVC

ASP.NET Core MVC è un framework di presentazione leggero, open source e altamente testabile ottimizzato per l'uso con ASP.NET Core, che viene utilizzato per creare siti Web dinamici in quanto fornisce uno sviluppo molto rapido. Costituisce un'alternativa al modello ASP.NET Web Forms utilizzato per applicazioni web [14]. Come si può intuire il nome deriva dal fatto che questo framework adotta il noto pattern MVC (Model-View-Controller) [15], perciò ogni applicazione web sviluppata con questo framework sarà divisa in tre componenti principali:

- Model: fornisce i metodi per accedere ai dati utili all'applicazione.
- View: visualizza i dati contenuti nel model e si occupa dell'interazione con utenti e agenti.
- Controller: riceve i comandi dell'utente (generalmente attraverso la View) e li attua modificando lo stato degli altri due componenti.

Questo comporta tra l'altro la tradizionale separazione tra logica applicativa, a carico di Model e Controller, e l'interfaccia utente, a carico della View.

ASP.NET Core MVC offre una serie di potenti funzionalità, tra cui:

- ROUTING: basandosi sul routing di ASP.NET Core è possibile eseguire mapping di URL e definire criteri di denominazione dell'URL dell'applicazione che funzionano perfettamente per l'ottimizzazione dei motori di ricerca e la generazione di collegamenti, indipendentemente dall'organizzazione dei file nel server web [16].
- ASSOCIAZIONE DI MODELLI: i dati delle richieste client vengono convertiti in oggetti che il controller è in grado di gestire, perciò non è necessario che la logica del

controller risolva i dati della richiesta in ingresso visto che li avrà a disposizione come parametri associati ai propri metodi [17].

- CONVALIDA DEL MODELLO: con l'ausilio di attributi di annotazione vengono associate al modello delle direttive di convalida dei dati.
- API WEB: questo framework è un ottimo supporto per lo sviluppo di Web API e servizi destinati ad un'ampia gamma di client, browser e dispositivi mobili. In più include il supporto per la negoziazione del contenuto http con il supporto predefinito per la formattazione dei dati come JSON o XML, offrendo anche la possibilità di definire formattatori personalizzati.

1.9 – Angular2

Angular2 è un framework open source per lo sviluppo di applicazioni web. È successore di AngularJS, ma le due versioni non sono compatibili in quanto Angular2 è stato completamente riscritto rispetto al predecessore. Anche i linguaggi di programmazione utilizzati sono diversi: AngularJS utilizzava JavaScript, mentre Angular2 utilizza TypeScript.

Le applicazioni sviluppate in Angular2 vengono eseguite direttamente dal web browser dopo essere state scaricate dal web server. Questo permette di risparmiare l'invio della pagina al server ogni volta che c'è una richiesta di azione da parte dell'utente. Il codice generato è compatibile con tutti i principali web browser moderni. Il framework Angular2 rappresenta uno strumento facile e veloce per sviluppare applicazioni che girano su qualunque piattaforma, inclusi smartphone e tablet, grazie alla combinazione con BootStrap che permette la alle applicazioni stesse di essere *responsive*[18] [19].

Ogni applicazione Angular2 è un insieme di **componenti** che interagiscono tra di loro e che possono essere combinati creando componenti organizzati gerarchicamente. Il componente principale è il root-component che costituisce il punto di ingresso dell'applicazione. Per la condivisione e il riutilizzo delle funzionalità tra componenti vengono messi a disposizione i **servizi**, che vengono iniettati all'interno dei componenti che ne hanno bisogno. Un altro elemento alla base di Angular2 sono i **moduli**, ossia contenitori di funzionalità che consentono di organizzare il codice di un'applicazione [18] [19]. Ogni modulo è definito da una serie di meta-informazioni che definiscono le dichiarazioni e le importazioni dei moduli esterni che vengono utilizzati dal modulo corrente. Tra le funzionalità degne di nota di Angular2 troviamo il Data Binding e il Routing. Il Data Binding è il processo di associazione tra fonte dei dati e

destinazione HTML, sostanzialmente utilizzato per mostrare dinamicamente il valore di un componente o di una sua proprietà. Angular2 permette di dividere l'applicazione in diverse View attraverso cui navigare mediante il concetto di routing. Questa funzionalità permette l'indirizzamento verso i diversi componenti in base alla URL digitata o mediante i link cliccati [18] [19].

1.10 – RabbitMQ

RabbitMQ è un gestore di code, detto anche message broker, cioè un software nel quale vengono definite delle code a cui si connettono le applicazioni per trasferire messaggi. Utilizza AMQP, un protocollo a livello applicativo per il message oriented middleware che garantisce funzionalità di messaggistica, accodamento, routing (sia punto-punto che publish-subscribe), affidabilità e sicurezza [20]. RabbitMQ rappresenta un intermediario tra i servizi, in questo caso le API Web, che necessitano di scambiarsi messaggi senza appesantire il carico di lavoro del server. Inoltre permette anche di distribuire un messaggio a più applicazioni e anche di bilanciare i carichi tra le applicazioni che sono in ascolto [21] I messaggi che arrivano al servizio non sono direttamente pubblicati nelle code dei destinatari ma vengono inizialmente presi in carico da un exchange. Questo componente esegue l'effettivo instradamento dei messaggi a diverse code compiendo scelte basate sul tipo di comunicazione e sul valore di alcuni attributi dei messaggi stessi, come la chiave di routing. A seconda del tipo di comunicazione può essere richiesto, per esempio, che le code in cui deve essere inserito il messaggio debbano avere una determinata chiave di routing. Le code che non soddisfano i requisiti non otterranno alcun messaggio, mentre se un messaggio non può essere inserito in nessuna coda viene semplicemente scartato. I messaggi rimangono in coda fino a quando non vengono gestiti da un'applicazione consumatore [21] [22].

Nel caso specifico di questo progetto il servizio viene utilizzato per invio di messaggi che serviranno poi ad un terzo servizio per l'invio di e-mail. Si adotta una comunicazione diretta basata sul pattern Produttore-Consumatore, o Publish-Subscribe. Questo tipo di comunicazione prevede che vengano create delle code dalle applicazioni Consumatori che si mettono in ascolto per ricevere il messaggio, e l'applicazione Produttore non farà altro che pubblicare i messaggi nelle code. Assume fondamentale importanza il valore della chiave di routing delle code: infatti solo le code che avranno una chiave di routing che fa match con quella del messaggio potranno ricevere il messaggio.

1.11 – Versioning - SVN

Apache Subversion – SVN – è un software open source di versionamento e di controllo versione utilizzato per mantenere le versioni di file e progetti nella versione corrente e in tutta la storia dei file stessi [23] [24]. Nello specifico per effettuare versioning nell'applicazione è stato utilizzato TortoiseSVN, uno dei client grafici di SVN programmato per funzionare come estensione di Microsoft Windows [25] [26]. Subversion, come tutti i programmi di versioning, si basa su due entità principali: repository e copia di lavoro. Subversion utilizza un database centrale che contiene tutti i file controllati dalla versione con la loro cronologia completa. Il database è indicato come **repository** e normalmente si trova su un server che esegue Subversion fornendo poi il contenuto ai client su richiesta. La copia di lavoro indica cosa ha a disposizione il singolo sviluppatore sul PC locale. È possibile scaricare l'ultima versione del repository e lavorarci localmente senza influire su nessun altro. Al termine della modifica è possibile effettuare un commit sul repository per renderla effettiva [23] [24]. Un file può quindi trovarsi in uno dei quattro seguenti stati:

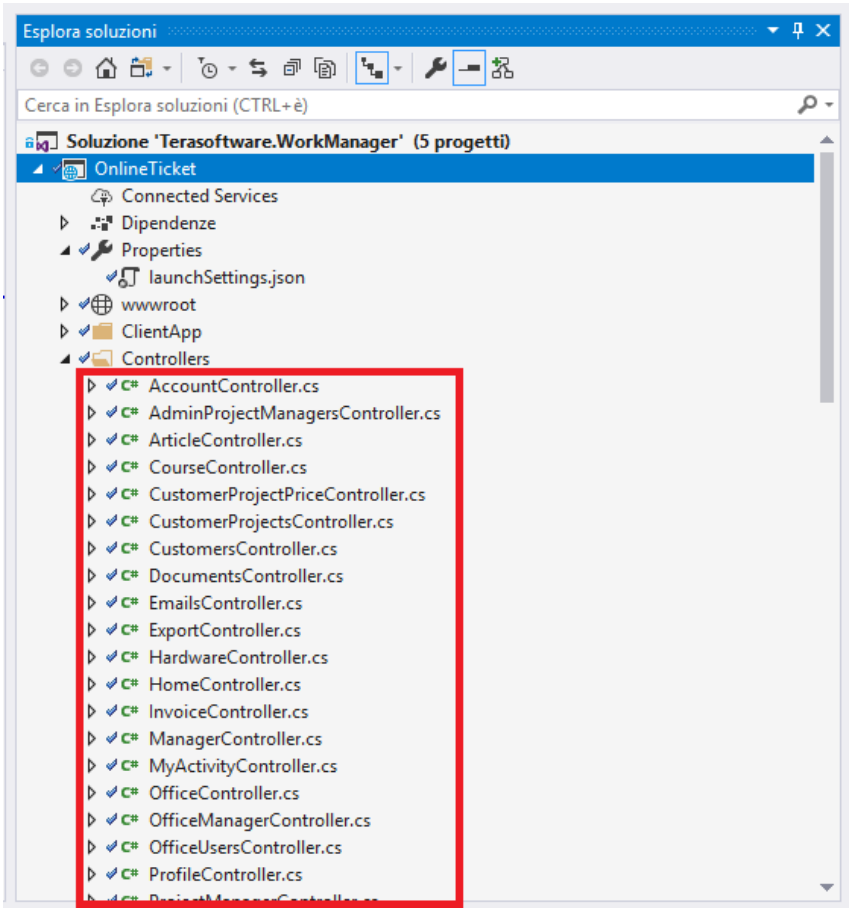
- Invariato e attuale: il file rimane invariato nella directory di lavoro e nessuna modifica a quel file è stata salvata nel repository dalla sua revisione funzionante.
- Modificato localmente e corrente: Il file è stato modificato nella directory di lavoro e nessuna modifica a quel file è stata memorizzata nel repository dall'ultimo aggiornamento.
- Invariato e obsoleto: Il file non è stato modificato nella directory di lavoro, ma è stato modificato nel repository. Il file deve essere aggiornato all'ultima revisione pubblica.
- Modificato localmente e non aggiornato: il file è stato modificato sia nella directory di lavoro che nel repository e dovrebbe essere aggiornato prima. Se Subversion non riesce a completare automaticamente l'unione in modo plausibile, lascia all'utente la risoluzione del conflitto.

Capitolo 2 – Analisi dell'applicazione

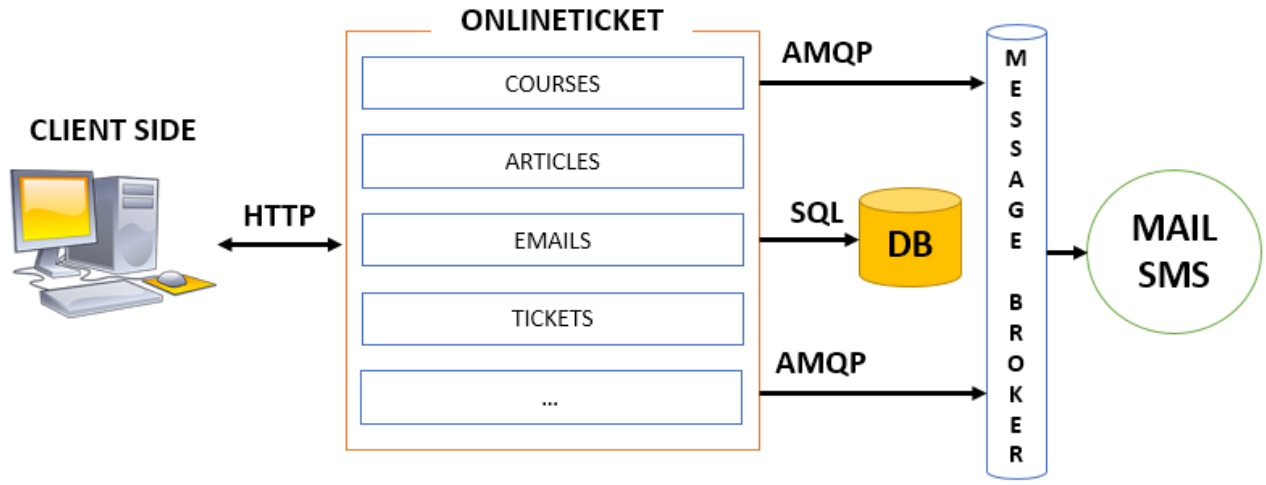
Lo scopo di questo capitolo è quello di presentare e analizzare l'applicazione WorkManager di FastCode nel suo stato iniziale, parlando della sua struttura e delle sue funzionalità con qualche dettaglio implementativo incentrandosi specificatamente nella parte dei corsi.

2.1 - WorkManager

L'applicazione WorkManager di FastCode è l'applicazione gestionale dell'azienda utilizzata per la gestione del personale, della loro formazione, dei progetti ecc…, cioè tutto quello che concerne l'attività aziendale. L'applicazione si presenta con la tradizionale struttura monolitica. Questo significa che lo stesso applicativo contiene tutti i servizi e viene utilizzato per la gestione di tutte le operazioni e funzioni necessarie. L'immagine sottostante è esplicativa di questo concetto: si può osservare come l'applicazione contenga al suo interno molteplici controllers che corrispondono alle diverse WebApi che compongono l'applicazione monolitica.

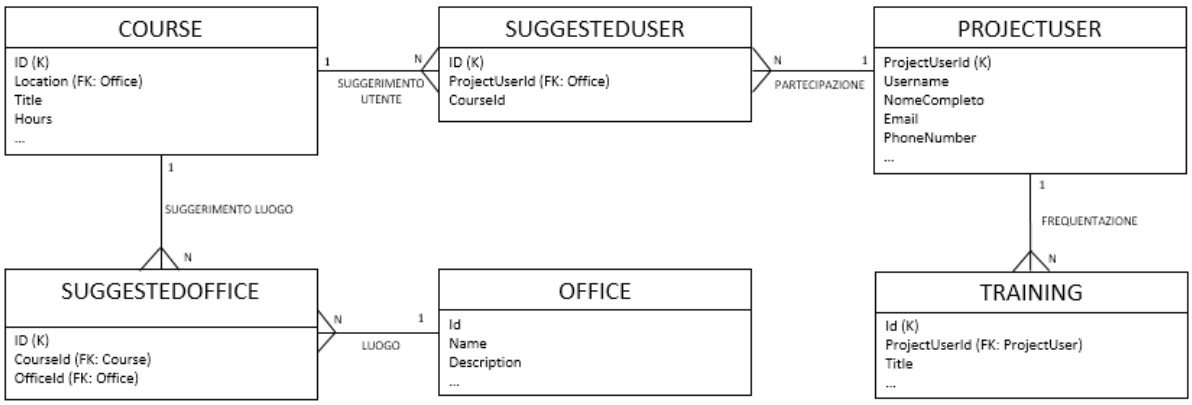


La figura seguente invece rappresenta uno schema del funzionamento del sistema, che di seguito verrà descritto nel dettaglio.



Database

Per quanto riguarda il database, si parla di un classico database relazionale, gestito mediante l'applicativo Microsoft SQL Server Management Studio 2014, che viene interrogato dall'applicazione mediante delle query SQL. La parte su cui ci si è focalizzati è stata solo quella relativa ai corsi, inizialmente così strutturata:



Le tabelle di maggiore interesse sono quelle su cui l'applicazione agisce direttamente inserendo e modificando dati cioè Course, SuggestedUser e SuggestedOffice:

Course: Id, Title, SuppyingAuthority, Cost, Location, StartDate, EndDate, Hours, Skills, Note, Status, CourseRelation, UserAdded, DateAdded, UserModified, DateModified, Active, AcceptedByAdmin

SuggestedUser: Id, CourseId, ProjectUserId

SuggestedOffice: Id, CourseId, OfficeId

Delle tabelle Office e ProjectUser non analizzeremo la struttura interna in quanto l'API dei corsi le utilizza soltanto in lettura, basti sapere che rappresentano rispettivamente gli uffici dell'azienda e gli utenti registrati nel sistema. La tabella Course rappresenta l'entità corso con tutte le informazioni relative quali costo, luogo di svolgimento, data di inizio e fine e altre. La tabella SuggestedUser rappresenta gli utenti iscritti ad un corso associando CourseId e ProjectUserId senza memorizzare altre informazioni. Stessa cosa per SuggestedOffice che rappresenta tutti gli uffici in cui il corso potrebbe svolgersi, anche qui associando CourseId e OfficeId. Questo consente di avere più di un utente e più di un ufficio associati ad un corso. La tabella Training, invece, costituisce sostanzialmente una ripetizione della tabella Course.

Lato client

Lato client troviamo un API sviluppata in Angular2 che gestisce e rappresenta il front-end dell'intera applicazione. Questa API è un agglomerato di componenti, ognuno delegato ad una precisa funzione e ognuno avente un servizio innestato nel costruttore del componente mediante il meccanismo di dependency injection. Ogni componente è associato ad un file HTML che rappresenta la sua interfaccia grafica, ed ad un file CSS, per definire lo stile. Quando l'utente interagisce con i controlli dell'interfaccia grafica questi generano un evento, che deve essere gestito dal componente associato. Se l'operazione da eseguire è interna all'applicazione client essa viene gestita interamente dal componente. Altrimenti se l'evento richiede che vengano reperiti o inviati dati al database, il componente richiama il servizio apposito. Il compito dei servizi è principalmente quello di inviare richieste alla WEB Api lato server al fine di reperire o inviare le informazioni, in base alle esigenze del componente. La comunicazione tra le due applicazioni avviene mediante protocollo HTTP: per le operazioni che richiedono il reperimento di dati da database verranno generate richieste di tipo GET, mentre per quelle che prevedono l'invio di dati al database le richieste generate saranno di tipo POST. In questo caso nel corpo della richiesta verrà inserito un oggetto contenente i dati da inviare, convertiti in formato JSON.

```
getCourse(id: number): Promise<CourseIndexModel> {
    let url = `${this.apiUrl}/GetCourse/${id}`;
    return this.http
            .get(url)
            .toPromise()
            .then(item => { return item.json() as CourseIndexModel; });
}
```

```
suggest(item: CourseViewModel): Promise<void> {
   let url = `${this.apiUrl}/Suggest`;
   return this.http
            .post(url, item)
            .toPromise()
            .then(item => { () => null; });
}
```

Questi esempi mostrano due funzioni del servizio Angular course.service che permettono di inviare due richieste per, rispettivamente, ottenere le informazioni relative ad un corso e suggerire un corso.

L'API Angular relativa ai corsi è gestita sostanzialmente da quattro componenti:

- **home.component** rappresenta il componente della home page dell'intera applicazione dove vengono mostrate sommariamente le principali informazioni su corsi, articoli e richieste. Nello specifico riguardo ai corsi venivano mostrati solo gli ultimi 5 corsi inseriti. Tramite questo componente è possibile anche effettuare l'inserimento/suggerimento del corso mediante un controllo in cui vengono inserite tutte le informazioni necessarie.
- **course.component** è il componente che rappresenta la pagina in cui vengono mostrati tutti i corsi divisi in pagine. A fianco dei corsi non ancora approvati appare un pulsante mediante il quale è possibile approvarlo, qualora il ruolo dell'utente sia amministratore o office manager.
- **course-detail.component** è il componente che mostra i dettagli del singolo corso, senza permettere ulteriori azioni all'utente.
- **training.component** contiene informazioni e funzionalità accessibili solo all'amministratore e all'office manager. Qui vengono riassunti i corsi svolti dall'utente, che possono essere approvati dall'amministratore o office manager i quali possono anche aggiungere un corso.

Quest'ultimo componente non si appoggia al servizio dei corsi utilizzato dagli altri componenti – course.service – ma ha innestato un servizio diverso dedicato all'entità **training**, ossia training.service. Questo servizio non comunica con il Controller dell'API server con cui comunicano gli altri servizi della parte dei corsi, cioè CourseController, ma con il TrainingController che contiene il riferimento alla tabella del database Training che è collegata alle altre tabelle in questione solo perché contiene l'userID dell'utente, ma con funzionalità non ben specifiche. Come già accennato, questo fatto comporta una ridondanza e crea confusione,

in quanto si hanno due tabelle nel database e due controller nell'API server che svolgono i compiti che potrebbero essere svolti solamente dalla tabella Course e dal controller dei corsi.

Lato server

Lato server, come detto, le richieste vengono intercettate dalla relativa Web Api del monolite. Qui vengono sfruttate due importanti funzionalità di ASP.NET: il routing e l'associazione di modelli, già accennati nel capitolo precedente. Il routing permette di associare un metodo ad ogni specifico URL (o viceversa), quindi ogni volta che un indirizzo viene raggiunto tramite una richiesta, verrà eseguito il metodo associato. Contestualmente ogni controller avrà assegnato un URL che lo contraddistingue dagli altri della stessa API, solitamente un URL di tipo gerarchico. La funzionalità di associazione dei modelli si lega strettamente a questo in quanto permette di convertire dati e oggetti contenuti negli URL o nel corpo delle richieste in formato JSON, in oggetti facilmente gestibili dal controller. Grazie a questo meccanismo di comunicazione e alle funzionalità offerte sia da Angular che da ASP.NET l'implementazione è molto facile in quanto nelle singole API vengono creati degli oggetti mediante appositi modelli che poi vengono direttamente inseriti nelle richieste e nelle risposte. L'entità ricevente avrà a disposizione tutti i dati strutturati secondo il modello definito, dati che possono essere utilizzati direttamente per svolgere operazioni o essere mappati in ulteriori oggetti appositamente definiti. Naturalmente le due API, per quanto riguarda i dati trasmessi e ricevuti sotto forma di oggetti, avranno dei modelli di riferimento identici, in quanto l'oggetto trasmesso deve precisamente combaciare con il modello di oggetto atteso da chi lo riceve. Gli oggetti e i dati che giungono all'API Server vengono quindi utilizzati per il reperimento, l'aggiunta, la modifica o la cancellazione dei dati nel database. Questo lavoro viene effettuato direttamente dal controller che riceve i dati, effettua eventuali controlli, e interagisce direttamente con il database per attuare l'operazione specifica. Per riagganciarsi all'esempio precedente, nel codice riportato sotto è presente il metodo lato server che permette di reperire un determinato corso dal database.

```
[Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
[HttpGet("{id}")]
public async Task<IActionResult> GetCourse(int? id)
{
   if (!id.HasValue)
      return BadRequest();
   var dbCourse = await (from x in _context.Course
                  join pu in _context.ProjectUsers
                  on x.UserAdded
                  equals pu.UserId
                  where x.Id == id
                  select new CourseIndexModel
                     {
                        Id = x.Id,
                        Title = x.Title,
                        Skills = x.Skills,
                        Cost = x.Cost,
                        DateAdded = x.DateAdded,
                        StartDate = x.StartDate,
                        EndDate = x.EndDate,
                        Hours = x.Hours,
                        Location = x.Location,
                        Note = x.Note,
                        Status = x.Status,
                        StatusText = x.Status.ToString(),
                        SupplyingAuthority = x.SupplyingAuthority,
                        UserAdded = x.UserAdded,
                        SuggestedBy = pu.NomeCompleto
                     })
                  .FirstOrDefaultAsync();
   return Ok(dbCourse);
}
```

Mediante l'annotazione [HttpGet("{id}")] assegnata al metodo viene effettuato il routing: in particolare viene specificato che l'URL mediante il quale è possibile raggiungere questo metodo deve contenere un attributo chiamato id. Supponendo che il controller dei corsi abbia URL **'http://www.workmanager.it/courses'**, questo metodo sarà raggiungibile all'indirizzo **'http://www.workmanager.it/courses/id**, dove id è solitamente un intero o una stringa. Così avviene l'associazione dei modelli, con il tipo dell'oggetto che viene specificato poi nell'intestazione 'public async Task<IActionResult> GetCourse(int? id)' : in questo caso specifica che il metodo richiede un intero opzionale di nome *id*, utilizzabile nel metodo. All'interno del controller è presente l'oggetto context di tipo DBContext, un modello che esegue il mapping di entità e relazioni definite nel database. Agendo su questo oggetto è possibile reperire e modificare direttamente le informazioni contenute nel DB attraverso delle query Linq. In questo caso specifico viene cercato ed eventualmente restituito il corso avente id corrispondente a quello indicato nell'URL, corso che viene direttamente inserito nella risposta che viene ritornata. Il discorso fatto in questo esempio per la gestione dei corsi è replicato in maniera del tutto simile dagli altri controller dell'applicazione.

Gestione dell'autenticazione

Per quanto riguarda la gestione dell'autenticazione, l'applicazione utilizza il meccanismo dei bearer token. Al momento del login, se questo va a buon fine, l'applicazione rilascia un token ossia una stringa criptata che viene memorizzata nel browser e che resterà valida per una certa durata stabilita dal server al momento della creazione. L'applicazione lato client che detiene il token, dovrà includerlo in ogni richiesta affinché il server possa verificarne la validità e, in caso di risultato positivo, fornire il risultato richiesto. Lato client il token viene memorizzato nel Local Storage del browser e viene incorporato nelle richieste che l'API invia mediante un http interceptor, un componente che intercetta tutte le richieste http che partono dall'API, permettendo di manipolarle. Il meccanismo di autenticazione viene utilizzato inoltre per la gestione dei ruoli dei diversi utenti. Al momento del login l'API server oltre al token di autenticazione fornisce anche il ruolo dell'utente, qualora l'utente abbia un ruolo, che viene memorizzato nel Local Storage. Per ruolo si intende 'Administrator' o 'OfficeManager', se invece l'utente non ha un ruolo specifico viene memorizzato un campo vuoto.

Servizi utilizzati

L'applicazione, inoltre, si appoggia ad un message broker RabbitMQ per la gestione delle comunicazioni mediante email e sms implementato lato client in .NET e lato server in Erlang. Il servizio è formato da due code con tecnica di comunicazione diretta, secondo il noto pattern Publish/Subscribe. La prima – WORK_QUEUE – è quella principale in cui vengono inseriti tutti i messaggi che arrivano dall'API Server che vengono poi consumati dal servizio. I messaggi che giungono nelle code vengono poi mandati ad un ulteriore servizio che si occupa della creazione della mail/sms e del suo effettivo invio al destinatario. Nel caso in cui fallisca l'invio al secondo servizio, il messaggio che non è stato inviato viene aggiunto nella coda di retry – RETRY_QUEUE. Per i messaggi che finiscono in questa coda viene effettuato un ulteriore tentativo di invio ogni 10 minuti. Ad ogni tentativo fallito il messaggio viene nuovamente accodato nella coda di retry.

2.2 – Problematiche della versione monolitica

Le numerose problematiche dovute alla mancanza di funzionalità unita anche al fatto che molte di quelle già implementate non svolgono come dovrebbero il loro compito, hanno portato l'azienda a sentire necessità di un portale funzionante secondo le proprie esigenze. Unito a

questo, l'architettura monolitica inizialmente adottata pone, o potrebbe porre, all'azienda una serie di problemi legati alla gestione di una grande quantità di codice e al deployment, ma soprattutto potrebbe rendere difficile un cambio di tecnologia (qualora ce ne fosse bisogno) e addirittura complicare l'estendibilità delle funzioni dell'applicazione stessa. Ciò che ha spinto l'azienda a valutare una riprogettazione del sistema sono soprattutto la necessità di scalabilità e la totale assenza di resilienza da parte dell'applicazione. Soprattutto si vuole che eventuali modifiche a funzionalità non-core dell'applicazione influiscano su funzionalità core durante il processo di deploy dell'applicazione. Da questa serie di limitazioni nasce la necessità dell'azienda di guardare oltre all'approccio monolitico e considerare una riprogettazione dell'applicazione in un'ottica più modulare, in modo da ridurre appunto le problematiche che scaturiscono dalla gestione di un macro-applicativo e ottenere benefici non solo gestionali (scalabilità, estendibilità, eccetera), ma anche in termini di costi e risorse impiegate. La riprogettazione avviene considerando il superamento della struttura monolitica a favore dell'approccio a microservizi. Nel capitolo successivo si entrerà nel dettaglio riguardo il modo in cui è stata affrontata questa riprogettazione entrando nello specifico per quanto riguarda la parte di gestione dei corsi, ma anche in ottica più generale riguardo all'intera applicazione.

Capitolo 3 – Riprogettazione applicazione

In questo capitolo si parlerà di come è stata riprogettata l'applicazione ponendo poi il focus sulla parte effettivamente sviluppata, quella della gestione dei corsi e della formazione del personale, riportando esempi e dettagli implementativi al fine di delineare come il sistema è cambiato dalla sua forma iniziale.

3.1 - Il nuovo Workmanager

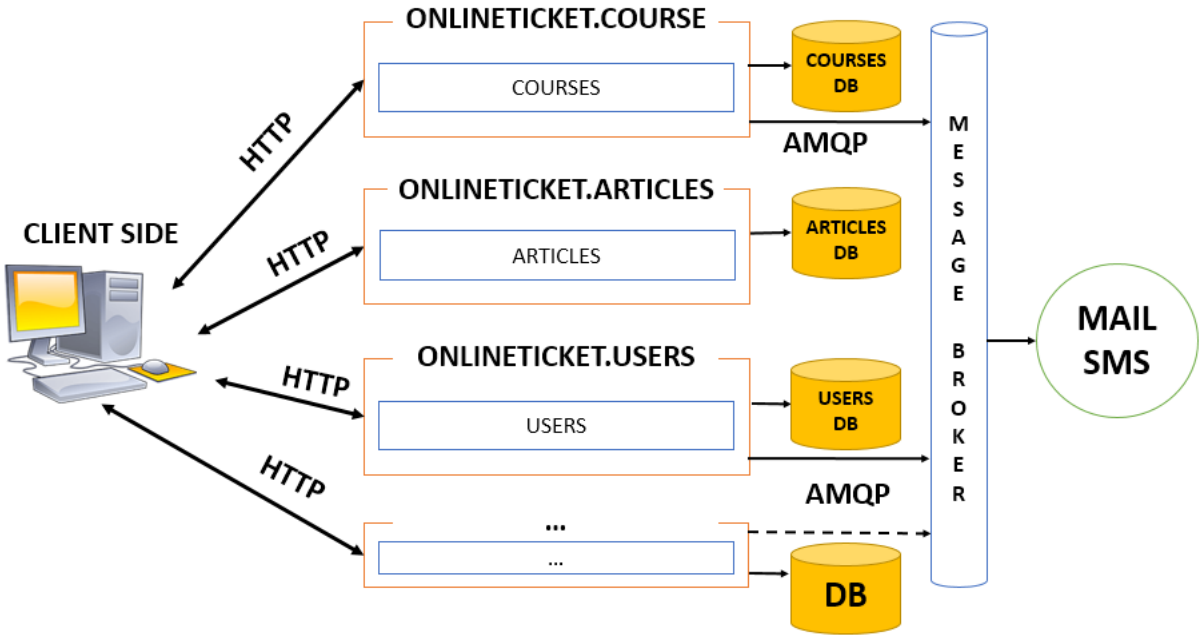
Come accennato nel finire del capitolo precedente, le esigenze dell'azienda hanno portato a valutare una riprogettazione del sistema passando dall'iniziale struttura monolitica ad un'architettura a microservizi. L'obiettivo della riprogettazione è quindi quello di far evolvere l'applicazione composta da N moduli, in un sistema composto da N microservizi indipendenti che rappresentano le differenti funzionalità del sistema. A livello applicativo, il monolite composto da numerosi controller verrà sostituito da altrettanti, o più, microservizi ognuno contenente un controller delegato ad una specifica funzione.

Questa riprogettazione influisce in positivo in tutto il ciclo di vita del software dallo sviluppo che avviene in tempi ridotti vista la minore mole di codice che compone ogni microservizio, alla manutenzione visto che è possibile isolare piccole porzioni da correggere nelle componenti non funzionanti.

A questo si aggiunge la possibilità di una maggiore scalabilità, processo che ora si riduce al singolo microservizio e quindi, portando un esempio, se il carico di lavoro relativo ai corsi aumenta è possibile scalare solo quel microservizio e non l'intera applicazione.

Anche la resilienza del sistema subisce un notevole miglioramento visto che, utilizzando questo approccio, ogni microservizio rappresenta un entità indipendente, quindi ogni modifica con conseguente deploy dell'applicazione riguardante un singolo microservizio non influisce sul funzionamento dei singoli altri microservizi ma soprattutto non influisce sull'intero sistema.

I benefici di questo fattore consistono quindi anche nel fatto che la modifica, o anche il malfunzionamento, di microservizi non-core non inficia in nessun modo sugli altri, in particolar modo sui microservizi che svolgono attività di tipo core.



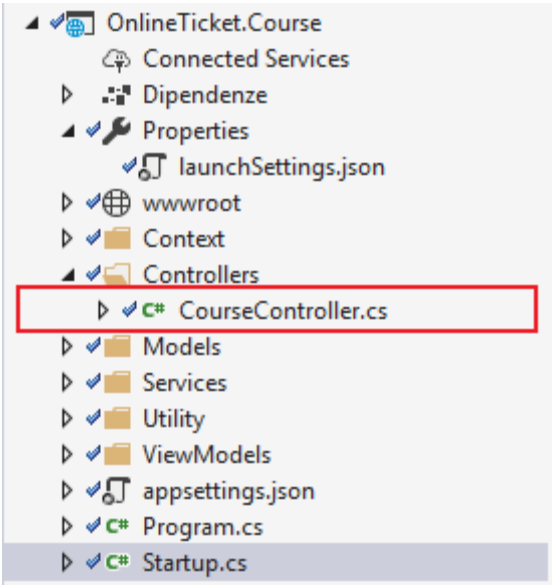
L'aspetto finale del sistema riprogettato dovrebbe essere come quello mostrato dalla figura, in cui sono rappresentati solo alcuni dei microservizi che comporranno il sistema. Le frecce che partono dai microservizi rappresentano i collegamenti presenti, ovvero i servizi che vengono utilizzati da quel microservizio. Non è detto che tutti i microservizi utilizzino tutti gli stessi servizi, ad eccezione del database. Per esempio il microservizio dei corsi e quello degli utenti utilizzeranno il Message Broker per l'invio di comunicazioni mediante email, mentre il microservizio degli articoli non implementerà questo collegamento. Per quanto riguarda la tipologia di database da adottare non esiste uniformità all'interno del progetto, in quanto ogni miscroservizio si interfaccerà con una tipologia di database più consona tenendo in considerazione il compito del microservizio stesso e le esigenze generali. Il microservizio sviluppato si basa momentaneamente su un database relazionale, ma microservizi che gestiscono altri compiti, come ad esempio i log, utilizzeranno database non relazionali.

Durante la riprogettazione, un punto critico da analizzare è stato proprio il database. Infatti l'applicazione iniziale utilizzava un database condiviso da ogni modulo che componeva l'applicazione. Riprogettando il sistema ci si è basati sulla teoria dei microservizi che vuole che ogni microservizio si appoggi ad un database proprio e indipendente dai database degli altri microservizi. Quanto definito in fase di progettazione non è stato possibile metterlo in atto durante lo sviluppo per due differenti motivi. Il primo è rappresentato dalle necessità dell'azienda, intenzionata a mantenere un database unificato e condiviso onde evitare di dover effettuare delle modifiche anche sugli altri applicativi che lo utilizzano. Il secondo consegue dal fatto che l'implementazione del sistema riprogettato è stata solo iniziata, con lo sviluppo

del microservizio per la gestione dei corsi e della formazione del personale, quindi la riprogettazione del database sarebbe dovuta avvenire in maniera parziale interessando solo la parte dei corsi. Se si pensa solo alla singola applicazione in esame questa cosa sarebbe stata possibile, ma avrebbe comportato la modifica anche di tutti i moduli dell'applicazione che si appoggiano sulla porzione di database dei corsi. In seguito quindi alla direttive aziendali e alle eventuali problematiche la scelta è stata quella di mantenere un database condiviso, opportunamente modificato in modo da poter implementare le funzionalità richieste ed eliminare problematiche inizialmente presenti.

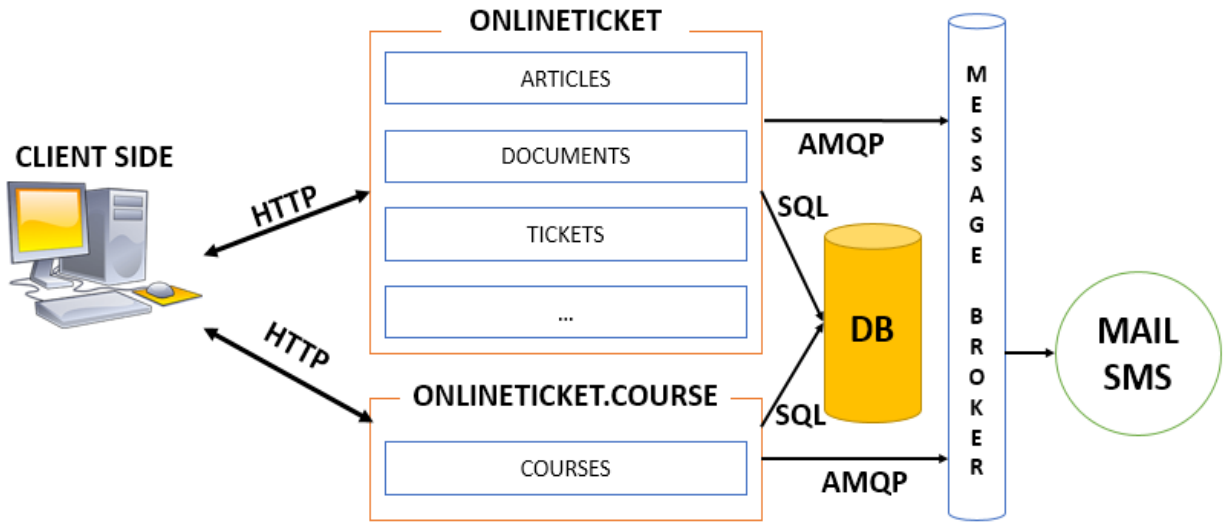
3.2 – Il microservizio per la gestione dei corsi

Come detto, nel contesto della riprogettazione del sistema è stato sviluppato il microservizio relativo alla gestione dei corsi e alla formazione del personale. Questo microservizio sostanzialmente prende il posto del modulo dei corsi facente parte dell'applicazione iniziale prima della riprogettazione, al quale sono state migliorate alcune funzionalità e aggiunte delle altre. L'immagine seguente, se confrontata con quella simile presente nel precedente capitolo, permette di capire meglio il concetto di microservizio. Prima avevamo un'applicazione composta da numerosi controller che svolgevano le varie funzionalità, mentre ora il microservizio creato contiene un solo controller al suo interno visto che si dovrà occupare soltanto della gestione dei corsi e della formazione.



Visto che non tutto il sistema è effettivamente implementato mediante l'architettura a microservizi, allo stato attuale l'applicazione presenta una struttura ibrida. Nello specifico il

modulo che gestiva i corsi nell'applicazione iniziale viene escluso e sostituito dal microservizio implementato, ma il resto dell'applicazione è rimasto ancora "monolitico" in attesa di essere modificata. L'immagine seguente raffigura la struttura del sistema dopo l'esclusione del modulo dei corsi e la creazione del relativo microservizio.



Le tecnologie e i protocolli utilizzati rimangono pressoché invariati; l'unico cambiamento che avviene riguarda appunto l'entità che svolge le funzionalità prima effettuate da una parte del sistema, ora svolte da un microservizio a sé.

3.3 - Estensione delle funzionalità

L'evoluzione dell'API dei corsi in microservizio non è il solo cambiamento che ha subito l'applicazione. Infatti, rispetto alla versione iniziale, sono state inserite molte funzionalità e migliorate quelle presenti anche facendo riferimento alle richieste e alle linee guida dell'azienda:

Le richieste principali prevedevano che il microservizio potesse permettere al personale interno, o all'azienda stessa, di proporre corsi e certificazioni da seguire. Gli utenti di ruolo amministratore o office manager devono essere in grado di approvare o meno i corsi proposti dal personale. In più si richiedeva una sezione personale per ogni utente riassuntiva della carriera aziendale, ovvero dei corsi di formazione frequentati.

Database

Come prima cosa non si può non parlare di come è stato modificato il database per la parte dei corsi, che subisce leggere variazioni rispetto a come era prima. Inizialmente l'iscrizione di un utente ad un corso veniva gestita mediante la tabella SuggestedUser, che però associava solo CourseId e ProjectUserId. Dopo l'analisi del problema iniziale si è giunti alla conclusione che sarebbero servite informazioni per poter sapere, ad esempio, se l'utente ha iniziato/finito il corso e quando. Questo viene effettuato mediante i campi DateStart e DateEnd, oltre al campo State che rappresenta se il partecipante è solo iscritto, se ha iniziato il corso o se l'ha finito. La tabella SuggestedUser diventa quindi la tabella CourseParticipant.

CourseParticipant: Id, CourseId, ProjectUserId, DateStart, DateEnd, State, NoteAdmin, NoteUser

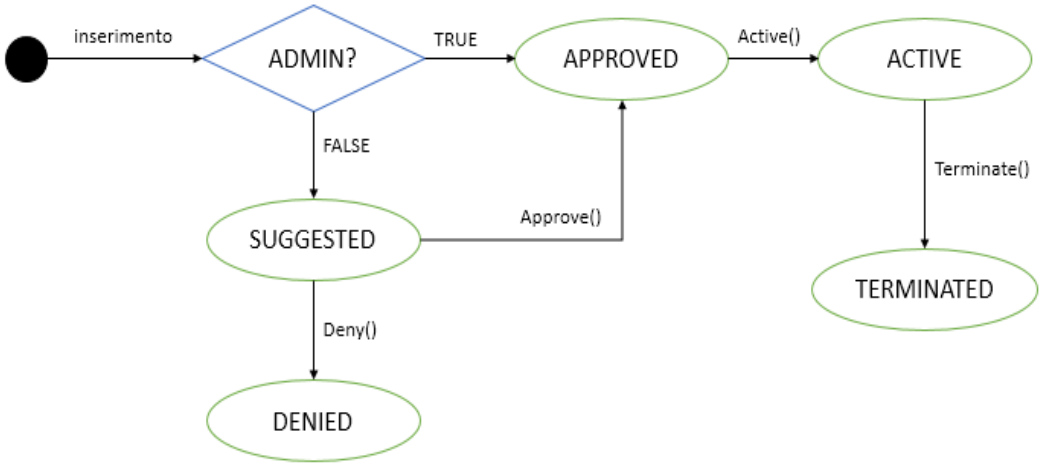
Inoltre anche la tabella corsi è stata modificata aggiungendo campi per tenere traccia di informazioni utili come ad esempio il tipo del corso e il numero massimo e minimo di partecipanti.

Course: Id, Title, SuppyingAuthority, Cost, Location, StartDate, EndDate, Hours, Skills, Note, Status, CourseRelation, UserAdded, DateAdded, UserModified, DateModified, Active, AcceptedByAdmin, TypeOfCourse, IsOpenCourse, MaxParticipants, MinParticipants

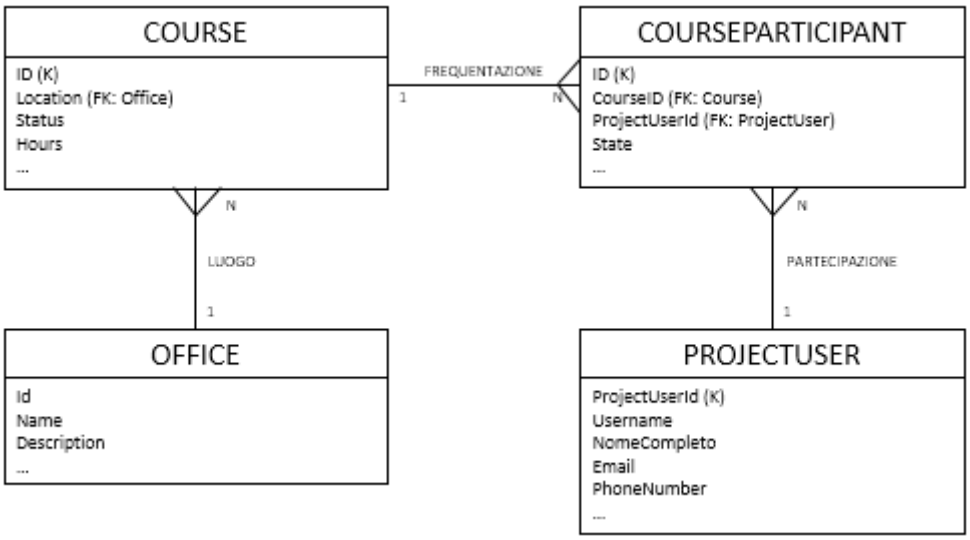
Inizialmente i campi AcceptedByAdmin, TypeOfCourse e IsOpenCourse potrebbero sembrare ridondanti e trarre in confusione, ma sono tutti legati alla logica di funzionamento del sistema e hanno dei ruoli ben definiti. AcceptedByAdmin è l'attributo che specifica se un corso suggerito da un user è stato approvato o meno dall'amministratore. Finché un corso non è approvato nessuno può iscriversi liberamente al corso. Naturalmente se è l'amministratore a proporre il corso esso sarà automaticamente approvato. TypeOfCourse specifica se il corso è stato suggerito dall'azienda (quindi dall'amministratore o dall'office manager) oppure da un semplice utente. IsOpenCourse invece specifica se l'iscrizione al corso è aperta o meno. Nel caso il corso sia chiuso nessun utente può iscriversi di sua spontanea volontà, ma sarà l'amministratore a doverlo inserire tra i partecipanti.

In seguito a queste modifiche a livello applicativo si è optato anche per la modifica all'attributo Status. Inizialmente questa informazione era pressoché inutile in quanto l'unico stato possibile era quello di corso suggerito. Ora gli stati sono cinque e rappresentano tutte le fasi di vita di un

corso: suggerito, approvato, attivo, terminato, negato e si susseguono come mostrato dal diagramma seguente.



Tornando alla struttura del database, sempre in seguito all'analisi iniziale si decide di mantenere un solo luogo per ogni corso e di memorizzarlo nel campo Location della tabella Course mediante il suo Id. Allo stato attuale delle cose si avrebbe una ridondanza di informazioni con la tabella SuggestedOffice, che viene quindi eliminata. L'aspetto finale del database presenta quindi la seguente struttura:



Lato client

Lato client l'API relativa ai corsi rimane gestita sostanzialmente dai "vecchi" 4 componenti:

- **home.component** viene modificato in modo da rappresentare informazioni sommarie in base al ruolo dell'utente. Se il ruolo è quello di amministratore o office manager

verranno mostrati gli ultimi corsi inseriti nel sistema e quelli da approvare. Se invece il ruolo è quello di utente normale verranno mostrati gli ultimi corsi da egli suggeriti e quelli a cui è iscritto.

- La funzionalità di inserimento/suggerimento del corso viene migliorata adattandola alle nuove specifiche e in modo da poter permettere l'inserimento dei nuovi campi del database. Anche per quanto riguarda questo controllo avvengono cambiamenti in base al ruolo dell'utente: infatti l'utente di ruolo amministratore può decidere se rendere chiuse le iscrizioni al corso e può inoltre aggiungere partecipanti. A questo proposito è stata aggiunta la possibilità di selezionare tutti gli utenti e di effettuare un filtraggio dei partecipanti in base al luogo scelto per il corso: scelta una zona quindi sarà possibile visualizzare solo gli utenti di quella zona.
- Anche **course.component** muta il suo comportamento a seconda del ruolo dell'utente. Per quanto riguarda l'amministratore amministratore o office manager vengono mostrati tutti i corsi, i corsi da approvare, quelli da attivare e quelli da terminare. Se si tratta di un normale utente verranno mostrati i corsi suggeriti e quelli a cui si può iscrivere. A prescindere dal ruolo vengono mostrati i corsi a cui l'utente è iscritto e quelli che sta svolgendo.
- **course.details.component** mostra i dettagli del singolo corso permettendo agli utenti di iscriversi e di disiscriversi, qualora possano farlo. All'amministratore è permesso inoltre di approvare il corso, aggiungere partecipanti, modificare il corso, farlo partire e terminare. La modifica del corso è permessa anche all'utente che l'ha inserito, ma solo prima che il corso venga fatto partire dall'amministratore.
- **training.component** viene completamente cambiato rispetto al suo aspetto iniziale. Ora infatti contiene per ogni utente un riassunto dei corsi che egli ha frequentato all'interno dell'azienda. Viene tolta la funzionalità di suggerimento/inserimento del corso già presente nella home page.

La modifica di quest'ultimo componente, di cui si erano evidenziate le limitazioni nello scorso capitolo, non comporta solo cambiamenti riguardo alle informazioni che vengono visualizzate. Infatti ora questo componente avrà innestato al suo interno non più il servizio che aveva in precedenza, ma quello relativo ai corsi come gli altri tre componenti appena descritti. Trattandosi dello stesso servizio, naturalmente comunicherà direttamente con il microservizio reperendo tutti i corsi che l'utente ha completato al pari degli altri componenti. Anche la tabella Training del database che veniva usata in precedenza non viene più utilizzata in quanto tutte le

operazioni del controller del microservizio si basano sulle quattro tabelle del database che compongono la parte dei corsi mostrata in precedenza.

La struttura delle richieste http che vengono effettuate mediante il servizio rimane sostanzialmente invariata, cambiano solo i componenti utilizzati per effettuare queste richieste. A partire dalla versione 5 di Angular infatti il modulo @angular/http viene deprecato in favore del nuovo modulo @angular/common/http. Le richieste ora avvengono mediante la classe HttpClient, che permette di ricevere in modo asincrono le risposte e mapparle nei vari oggetti del servizio. Questo cambiamento comporta che le nuove richieste non vengono più intercettate dall'http interceptor già presente, ma ne necessita uno nuovo. Perciò è stato necessario implementare un nuovo componente, **course-request.interceptor**, che intercetta tutte le richieste di tipo HttpRequest ma non permette di manipolarle direttamente, infatti per ogni richiesta è necessario fare un clone che è poi possibile modificare. Naturalmente le richieste non verranno più mandate all'URL del vecchio servizio, ma viene indicato ora l'URL del microservizio creato.

Lato server

Il microservizio prende il nome di OnlineTicket.Course e come struttura interna è molto simile al vecchio, se non per il fatto che è un solo servizio indipendente e con funzionalità estese. Solo per citarne alcune viene aggiunta la possibilità di inserire partecipanti ai corsi (sia al momento della creazione che successivamente) e di modificare il corso finché questo non è viene fatto partire. Per quanto riguarda l'autenticazione è logico che non può essere gestita autonomamente dal microservizio, in quanto tutti i token dovrebbero poi essere riconosciuti sia da OnlineTicket che da OnlineTicket.Course. Pur essendo indipendente, il microservizio fa comunque parte di un sistema quindi viene da se che l'unica soluzione è utilizzare lo stesso sistema utilizzato dalla parte monolitica del sistema. Questo è stato possibile fornendo al sistema di autenticazione di ASP.NET del microservizio le stesse credenziali (chiave di cifratura e Issuer) dell'applicazione: perciò ogni token che viene rilasciato in seguito al login sarà valido sia per le richieste ad OnlineTicket che per quelle ad OnlineTicket.Course. Oltre ad offrire molte più funzionalità, l'altra grande differenza tra la vecchia applicazione e il microservizio è sostanzialmente l'inserimento di entità intermedie tra il controller dell'API server e il database. Se prima l'accesso avveniva in maniera diretta dal controller, ora è delegato a delle classi dedicate, ognuna per ogni tabella del database mediante il modello di progettazione dependency injection offerto da ASP.NET [27]. Questo permette di utilizzare gli oggetti che comunicano con il

database come servizi all'interno del controller, soltanto dichiarando la dipendenza nelle file Startup del microservizio, dove vengono configurati tutti i servizi:

services.AddScoped<ICourseParticipantService, CourseParticipantService>();

Mediante questo mapping delle dipendenze, il servizio viene direttamente reperito mediante il costruttore del controller e sarà poi direttamente utilizzabile mettendo a disposizione tutti i metodi dichiarati nella sua interfaccia. Ad esempio CourseParticipantService conterà tutti i metodi per inserire, cancellare o modificare i partecipanti al corso agendo sulla tabella CourseParticipant del database; allo stesso modo CourseService gestisce i corsi e agirà sulla tabella Course. L'accesso al database da parte di questi servizi avviene in maniera molto simile a come avveniva nel vecchio servizio, cioè mediante degli oggetti DbContext che eseguono il mapping delle entità del database in oggetti nell'API. Il controller ha quindi a disposizione i servizi che permettono l'accesso a tutte le tabelle di interesse, perciò può effettuare in autonomia tutte le operazioni che necessitano per poter soddisfare le richieste http che arrivano.

```
1  [Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
2  [HttpGet("{id}")]
3  public IActionResult GetCourse(int id) {
4      var course = _courseService.GetById(id);
5      var IDs = new List<int>();
6      var participants = new List<Participant>();
7      _courseParticipantService.getAll()
8              .Where(x => x.IdCourse == course.Id)
9              .ToList().ForEach(cp => IDs.Add(cp.ProjectUserId));
10     IDs.ForEach(item => {
11             var user = _projectUserService.getById(item);
12             var participant = new Participant() {
13                 ProjectUserId = user.ProjectUserId,
14                 UserId = user.UserId,
15                 Email = user.Email,
16                 NomeCompleto = user.NomeCompleto,
17                 EmailCC = user.EmailCC,
18                 DefaultUserName = user.DefaultUserName,
19                 PhoneNumber = user.PhoneNumber,
20                 EmailPersonale = user.EmailPersonale,
21                 SkypeId = user.SkypeId,
22                 HangoutId = user.HangoutId
23             };
24             _courseParticipantService.getAll().ToList().ForEach(cp => {
25                 if (cp.IdCourse==course.Id && cp.ProjectUserId==user.ProjectUserId)
26                 {
27                     participant.NoteUser = cp.NoteUser;
28                     participant.NoteAdmin = cp.NoteAdmin;
29                     participant.Status = cp.State;
30                 }
31             });
32             participants.Add(participant);
33     });
34     string location = _locationService.GetById(course.Location).Description;
```

```
35      var courseToReturn = new DetailedCourse()
36      {
37          Participants = participants,
38          Location = location,
39          Id = course.Id,
40          Active = course.Active,
41          Cost = course.Cost,
42          DateAdded = course.DateAdded,
43          DateModified = course.DateModified,
44          EndDate = course.EndDate,
45          Hours = course.Hours,
46          Note = course.Note,
47          Skills = course.Skills,
49          StartDate = course.StartDate,
50          Status = course.Status,
51          SupplyingAuthority = course.SupplyingAuthority,
52          Title = course.Title,
53          UserAdded = course.UserAdded,
54          UserModified = course.UserModified,
55          CourseRelation = course.CourseRelation,
56          AcceptedByAdmin = course.AcceptedByAdmin,
57          TypeOfCourse = course.TypeOfCourse,
58          IsOpenCourse = course.IsOpenCourse,
59          MaxParticipants = course.MaxParticipants,
60          MinParticipants = course.MinParticipants,
61      };
62      return Ok(courseToReturn);
63  }
```

In questo esempio si può notare come il metodo raggiunto mediante richiesta http non acceda direttamente all'istanza **context** del database come succedeva invece in precedenza. Quello che avviene è una delegazione di questo compito ai servizi (ognuno delegato a gestire una certa entità del database) che vengono innestati all'interno, i quali mettono a disposizione vari metodi utili, ad esempio, per inserimenti o per reperire certe informazioni. Andando più nei particolari dell'esempio, abbiamo il metodo che permette di ottenere tutte le informazioni relative al corso con un certo id specificato, inclusa la location in forma di stringa e anche la lista dei partecipanti. Come prima cosa vengono reperite le informazioni sul corso (riga 4) accedendo al servizio **CourseService** che si occupa dei corsi. Dopodiché, mediante l'id del corso vengono richiesti tutti i partecipanti a quel corso al servizio **CourseParticipantService** (righe 7, 8, 9), cioè gli userId dei partecipanti, poi tramite questi identificatori vengono richiesti i loro dati utente al servizio **ProjectUserService** (riga 11). Infine tramite il servizio **LocationService** si ottiene il nome completo della Location dove si svolgerà il corso (riga 34). Il compito del servizio è altrettanto semplice dato che l'accesso al database è molto facilitato dall'utilizzo degli oggetti di tipo DbContext che mappano le tabelle del database in oggetti di tipo DbSet, quindi gestibili come qualsiasi altro tipo di collezione di C#.

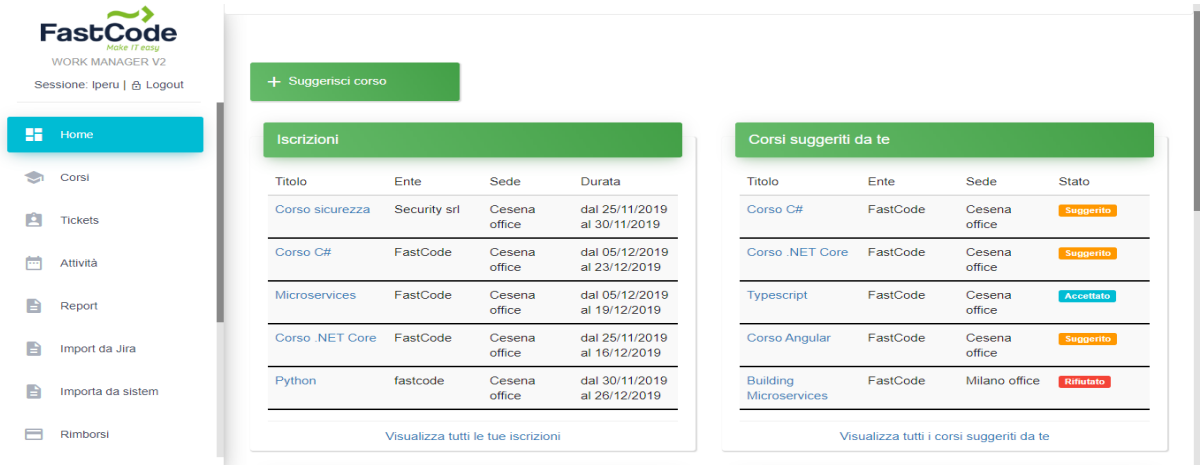
```
public ProjectUsers getById(int id)
   {
      return _context.ProjectUsers.Find(id);
   }
```

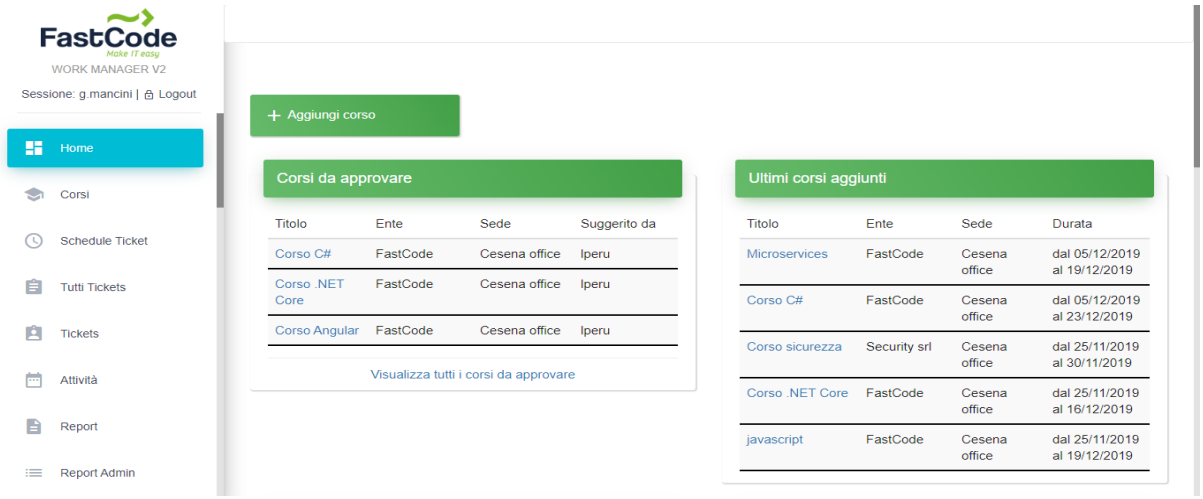
Per quanto riguarda il servizio e-mail utilizzato anche in precedenza, **ServiceBroker**, è stato rispristinato e reso utilizzabile anche dal microservizio semplicemente aggiungendolo tra le dipendenze del microservizio stesso. Lato controller viene semplicemente chiamato un ulteriore servizio adibito alle e-mail, **EmailService**, che costruisce un oggetto di tipo ServiceBroker e ne richiama la funzione Enqueue che permette di accodare un messaggio alla coda di RabbitMQ, dopodiché il messaggio viene inserito nel database nell'apposita tabella SentEmail. Come accennato già nel capitolo precedente la coda RabbitMQ a cui vengono inviati i messaggi è una semplice coda basata sul meccanismo publish/subscribe che riceve messaggi in coda e ad uno ad uno li consuma inviandoli ad un ulteriore servizio il quale è il vero responsabile della creazione e dell'invio dell'email.

3.4 – Screenshots dell'applicazione

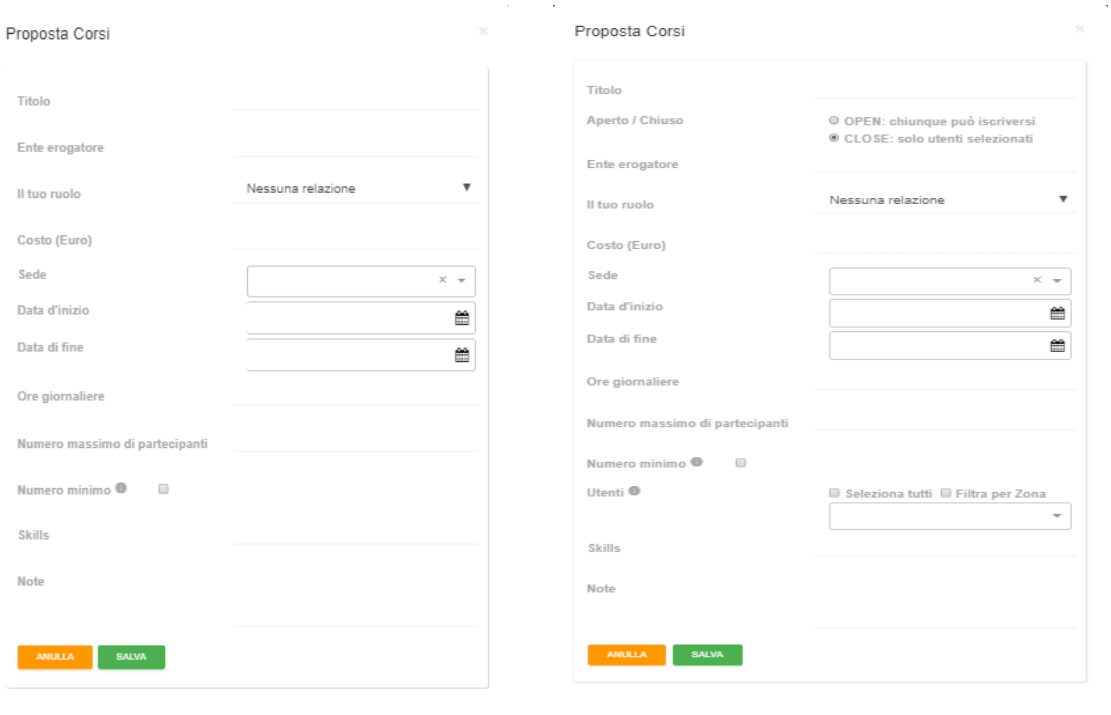
Di seguito vengono riportate alcune schermate dell'applicazione al fine di presentarne anche il funzionamento.

L'home page, come detto, è legata al componente home.component e subisce cambiamenti in base al ruolo dell'utente. Nel primo screenshot è raffigurato ciò che vede l'utente. La sezione di sinistra contiene gli ultimi 5 corsi a cui l'utente è iscritto, mentre quella di sinistra gli ultimi 5 corsi suggeriti dall'utente, ognuno con il relativo stato. Nell'immagine successiva invece è raffigurata la home page che si presenta all'amministratore/office manager. In questo caso la sezione di sinistra contiene gli ultimi 5 corsi che devono essere approvati (o rifiutati), mentre quella di destra contiene gli ultimi 5 corsi aggiunti.



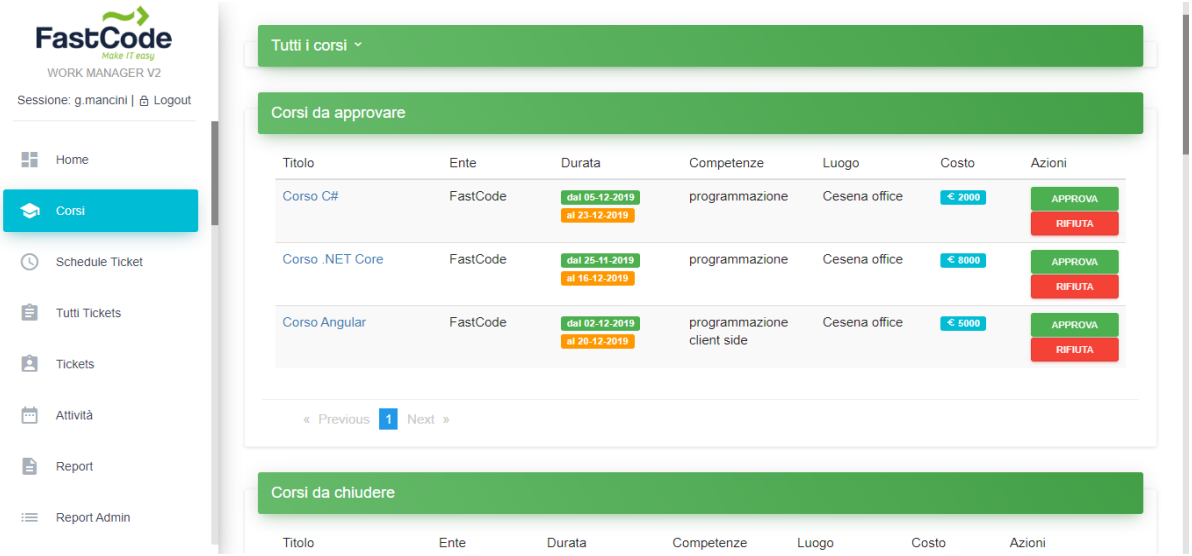


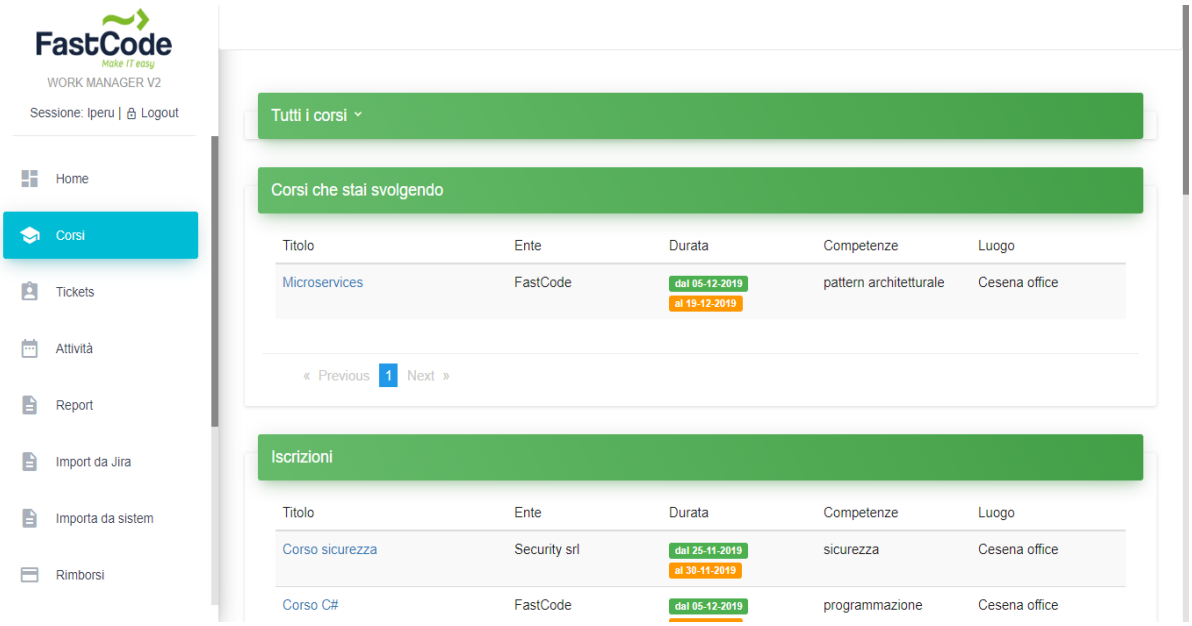
Anche per quanto riguarda la form di suggerimento/inserimento dei corsi, che si apre cliccando sul pulsante verde in alto, avvengono cambiamenti in base al ruolo dell'utente. Al contrario di un normale utente, l'amministratore ha la facoltà di effettuare più scelte relative al corso come ad esempio scegliere se l'iscrizione sia aperta o chiusa. Nel primo caso chiunque potrà iscriversi, mentre nel secondo i partecipanti sono scelti dall'amministratore mediante il menu a tendina opportuno.



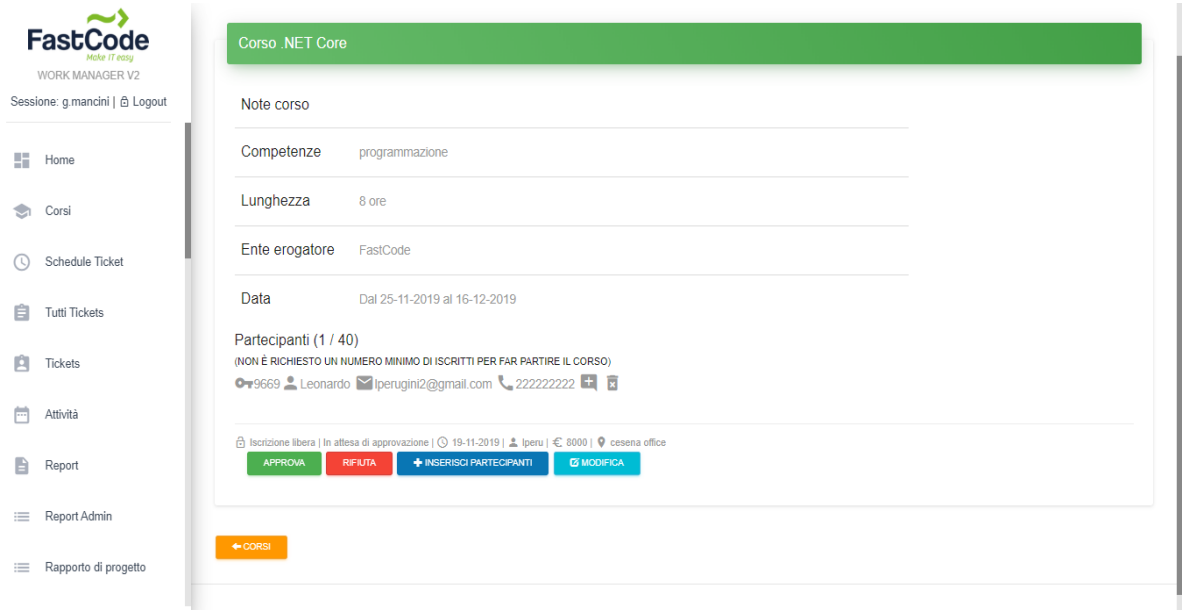
Il componente course.component è responsabile della visualizzazione della pagina contenente tutte le varie sezioni dei corsi. Anche qui nel caso in cui l'utente sia amministratore vengono mostrate informazioni differenti rispetto a quelle visualizzate da un utente normale, come spiegato in precedenza. Di seguito è mostrata questa distinzione: nel primo screenshot è

raffigurata la pagina vista dall'admin, mentre in quello successivo la pagina visualizzata da un normale utente.

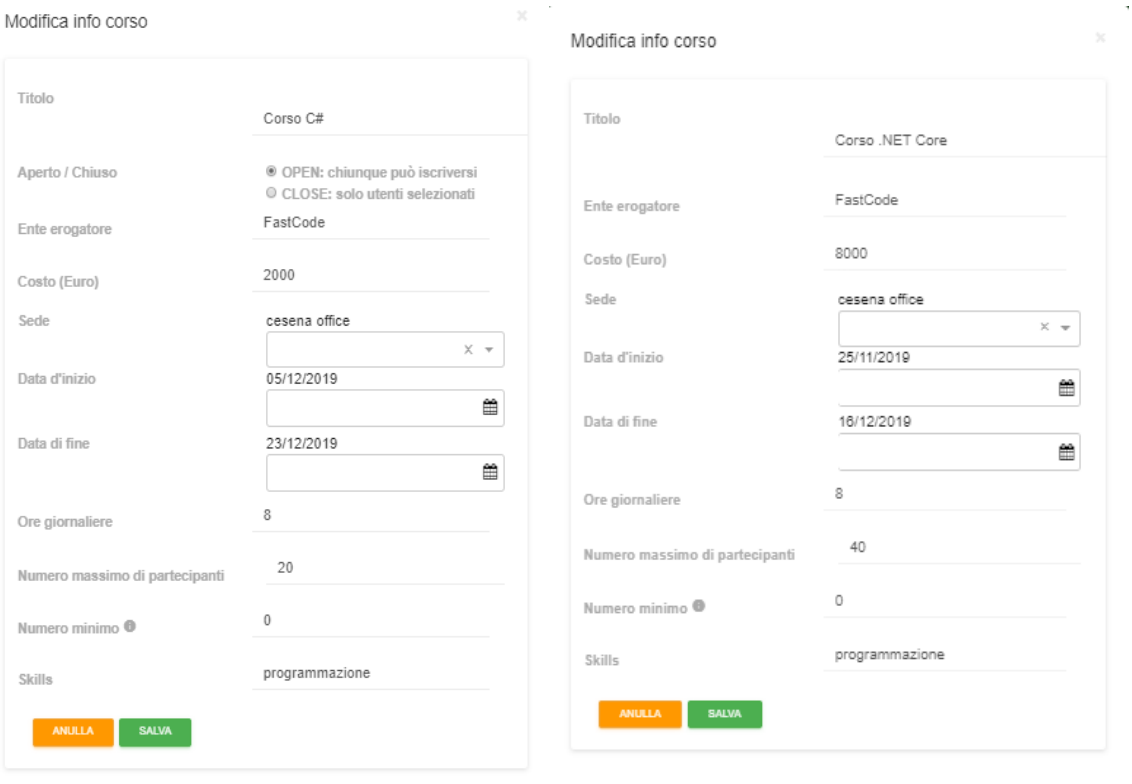




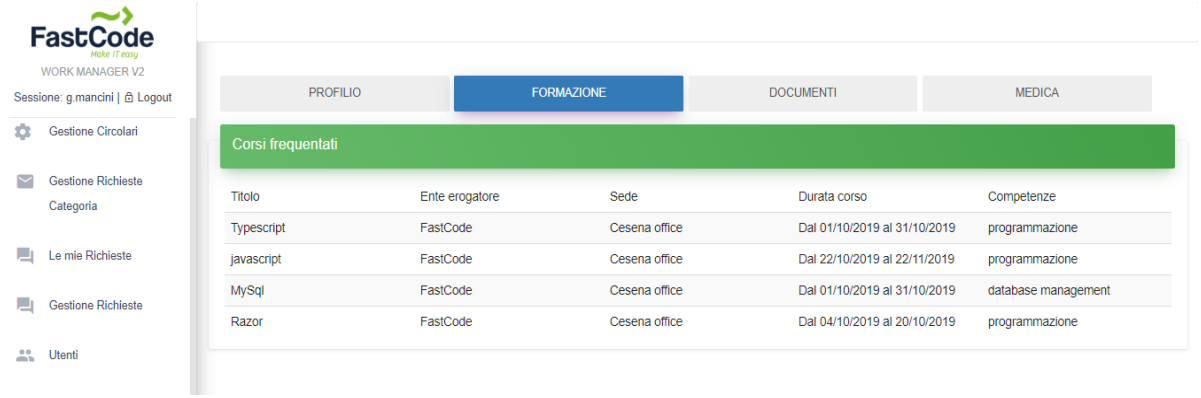
La pagina che mostra le informazioni dettagliate sul singolo corso è gestita dal componente course-detail.component. In questa pagina il ruolo dell'utente non influisce sulle informazioni che vengono mostrate, ma piuttosto sulle azioni che è possibile compiere. L'immagine sottostante rappresenta la pagina vista dall'amministratore per un corso proposto, ma non ancora approvato. È concessa la possibilità di approvare, rifiutare, modificare il corso o aggiungere partecipanti mediante gli appositi bottoni. In maniera simile dopo l'approvazione sarà possibile attivare e terminare il corso. Un normale utente, in base al tipo di corso e allo stato di avanzamento, potrà iscriversi o disiscriversi, attivare e terminare il corso.



Prima dell'approvazione del corso, sia l'admin che l'utente che ha inserito il corso, avranno la possibilità di effettuare delle modifiche. Anche qui il ruolo dell'utente è da tenere in considerazione in quanto l'amministratore potrà modificare più informazioni rispetto ad un normale utente. La form di modifica mostra le informazioni attuali del corso permettendone la modifica. All'amministratore è inoltre consentito di aggiungere partecipanti, mediante un'ulteriore form.



Il componente training.component gestisce la pagina di visualizzazione della carriera, dove sono elencati tutti i corsi svolti dall'utente durante il suo impiego nell'azienda.



Conclusioni

In questa tesi ho analizzato quanto svolto presso l'azienda FastCode riguardo alla riprogettazione dell'applicazione gestionale WorkManager utilizzata dall'azienda stessa, evolvendola da un approccio monolitico ad un'architettura a microservizi. L'idea della riprogettazione nasce dalla necessità di un'applicazione più modulare al fine di migliorare i processi di scaling del sistema. La modularità offerta dai microservizi inoltre permetterà all'azienda di poter estendere, modificare e migliorare ogni componente del sistema in maniera autonoma e senza inficiare sull'operato degli altri microservizi.

Nell'ambito di questa riprogettazione ho dato il via all'implementazione del sistema riprogettato sviluppando il microservizio per la gestione della formazione del personale aziendale. Momentaneamente infatti il sistema presenta ancora una struttura ibrida in quanto solo la gestione dei corsi di formazione avviene mediante un microservizio, mentre tutto il resto delle funzionalità (personale, rendicontazione, eccetera) rimane svolto dal "monolite".

Osservazioni

Lo sviluppo del microservizio è avvenuto senza particolari problemi. Questo è stato possibile soprattutto grazie ad una precedente esperienza di tirocinio curriculare in azienda proprio sul tema dei microservizi, cosa che ha reso necessario solo un breve ripasso riguardo al pattern architetturale e al linguaggio di programmazione utilizzato.

Diverso discorso per lo sviluppo dell'applicazione in Angular2. Trattandosi di un framework non conosciuto è stato necessario un periodo iniziale che mi ha permesso di prendere familiarità con l'ambiente e con il linguaggio utilizzato.

Le maggiori criticità sono state riscontrate però nel tentativo di comunicazione tra l'applicazione lato client e il microservizio. La soluzione è stata trovata nell'aggiornamento della versione di Angular e all'implementazione di un componente 'interceptor', che hanno permesso l'utilizzo di componenti aggiornati in modo da poter far arrivare richieste autenticate al microservizio.

Un'altra criticità, già descritta, ha riguardato il database. Sebbene la teoria voglia che ogni microservizio comunichi con un proprio database indipendente, le direttive assegnatemi

dall'azienda richiedevano la necessità di mantenere un database unico e condiviso per evitare che una modifica ingente del database richiedesse una difficile integrazione con le altre applicazioni che lo utilizzano.

Related Work

Il successo riscosso dal pattern architetturale a microservizi e i numerosi benefici correlati hanno portato all'ideazione di ulteriori architetture basate sugli stessi principi non solo per quanto riguarda il backend ma anche in ottica frontend, come l'approccio a Microfrontend.

Come accadeva per le applicazioni server-side prima dell'avvento dei microservices, le applicazioni lato client sono prevalentemente composte da un enorme blocco monolitico che svolge tutte le funzionalità. L'approccio a microfrontend prende il concetto alla base dei microservizi e lo applica al frontend suddividendo il blocco monolitico in sottodomini, ognuno corrispondente ad una singola funzionalità dell'intero dominio aziendale. In questo modo ogni microfrontend può essere sviluppato, innovato e distribuito in maniera totalmente indipendente dagli altri senza interessare l'intera applicazione. Siccome però nella maggior parte delle applicazioni frontend e backend interessano le stesse funzionalità dello stesso sottodominio, è prassi metterli assieme e farli gestire da un unico team. In questo modo ogni team è interfunzionale e sviluppa le sue funzionalità end-to-end, dal database all'interfaccia utente, lavorando in maniera autonoma su basi di codice ridotte e coerenti con la possibilità di effettuare modifiche o estensioni in modo più incrementale rispetto al passato [28] [29] [30].

Nel lavoro di questa tesi l'approccio in questione non è stato adottato in quanto le linee guida e le richieste fornitemi dell'azienda non lo richiedevano.

Future Work

Le possibilità di evoluzione dell'applicazione sviluppata potrebbero essere molteplici. In questa tesi sono state analizzate alcune tecniche che possono essere integrate alle applicazioni con architettura a microservizi per poterne migliorare ad esempio il deploy e lo scaling, anche se allo stato attuale il microservizio sviluppato non ne implementa alcuna.

Come prima cosa non si può non citare il fatto che l'implementazione del sistema riprogettato verrà portata a termine, ottenendo quindi un sistema con architettura completamente a

microservizi. Inoltre non si può escludere una ulteriore riprogettazione che includa il pattern architetturale a microfrontend sopra descritto.

Sicuramente si può pensare ad un futuro nel cloud per l'applicazione, ottenendo benefici in efficienza ed eliminando i costi relativi all'hardware. A questo si aggiunge l'opportunità di innovare prodotti e servizi tramite modelli scalabili e misurabili. Proprio riguardo alle tecniche di scalabilità dinamica offerte dal cloud computing entrano in gioco i meccanismi di autoscaling [7] precedentemente descritti. Nello specifico contesto di FastCode, avente molti uffici dislocati in Italia e all'estero, è immaginabile poter gestire i meccanismi di scalabilità in maniera completamente automatizzata. Per quanto riguarda il miglioramento del deploy di microservizi si può pensare che anche esso venga gestito in futuro in maniera automatica mediante meccanismi di Continuous Integration [5] già applicati dall'azienda per altri progetti.

Bibliografia

1. R. Gnatyk. (2018, October 03). Microservices vs Monolith: which architecture is the best choice for your business?
   Da https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/#.
2. J. Lewis, M. Fowler. (2014, March 25). Microservices, a definition of this new architectural term.
   Da https://www.martinfowler.com/articles/microservices.html.
3. A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, A. Sadovykh, Microservices. Science and Engineering. ISBN 978-3-030-31645-7, Springer (2020).
4. A. Acerbis. MokaByte 223 - Dicembre 2016.
   Da http://www.mokabyte.it/2016/12/microservizi-1/.
5. M. Fowler. (2006, May 01). Continuous Integration.
   Da https://martinfowler.com/articles/continuousIntegration.html.
6. Solutionsiq.com.
   Da https://www.solutionsiq.com/agile-glossary/integration-hell/.
7. Google Cloud Documentation.
   Da https://cloud.google.com/compute/docs/autoscaler/.
8. Wikipedia.com. Microsoft .NET.
   Da https://it.wikipedia.org/wiki/Microsoft_.NET.
9. A. Aggarwal – GeeksForGeeks. Introduction to .NET Framework.
   Da https://www.geeksforgeeks.org/introduction-to-net-framework/.
10. A. Aggarwal – GeeksForGeeks. Common Language Runtime (CLR) in C#.
    Da https://www.geeksforgeeks.org/common-language-runtime-clr-in-c-sharp/.
11. Microsoft Documentation. (2016, October 23). Entity Framework 6.
    Da https://docs.microsoft.com/it-it/ef/ef6/.
12. Microsoft Documentation. (2017, March 30). ADO.NET Overview.
    Da https://docs.microsoft.com/it-it/dotnet/framework/data/adonet/ado-net-overview.
13. D. Roth, R. Anderson, S. Luttin. (2019. April 07). Introduction to ASP.NET Core.
    Da https://docs.microsoft.com/it-it/aspnet/core/index?view=aspnetcore-3.0.
14. S. Smith. Overview of ASP.NET Core MVC.

Da https://docs.microsoft.com/it-it/aspnet/core/mvc/overview?view=aspnetcore-3.0.

15. Wikipedia.com. Model-view-controller.
    Da https://it.wikipedia.org/wiki/Model-view-controller.
16. R Nowak, S Smith, R. Anderson. (2019, September 24). Routing in ASP.NET Core.
    Da https://docs.microsoft.com/it-it/aspnet/core/fundamentals/routing?view=aspnetcore-3.0.
17. Microsoft Documentation. Model Binding in ASP.NET Core.
    Da https://docs.microsoft.com/it-it/aspnet/core/mvc/models/model-binding?view=aspnetcore-3.0.
18. TutorialsPoint.com. Angular2 Overview.
    Da https://www.tutorialspoint.com/angular2/angular2_overview.htm.
19. Angular.io. Architecture overview.
    Da https://angular.io/guide/architecture.
20. Wikipedia.com. Advanced Message Queuing Protocol.
    Da https://it.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol.
21. L. Johansson. (2015, May 18). RabbitMQ for beginners - What is RabbitMQ?
    Da https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html
22. L. Johansson (2015, September 03). RabbitMQ Exchanges, routing keys and bindings.
    Da https://www.cloudamqp.com/blog/2015-09-03-part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html.
23. Subversion.apache.org. Apache Subversion Features.
    Da https://subversion.apache.org/features.html.
24. Wikipedia.com. Apache Subversion.
    Da https://en.wikipedia.org/wiki/Apache_Subversion.
25. Tortoisesvn.net. About TortoiseSVN.
    Da https://tortoisesvn.net/about.html.
26. Wikipedia.com. TortoiseSVN.
    Da https://en.wikipedia.org/wiki/TortoiseSVN.
27. S. Namrouti, R. Anderson, S.Smith. (2019, February 24). Dependency injection into controllers in ASP.NET Core.
    Da https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/dependency-injection?view=aspnetcore-3.0.
28. C. Jackson. (2019, June 19). Micro Frontends.

Da https://martinfowler.com/articles/micro-frontends.html.

29. M. Geers. Micro Frontends - Extending the microservice idea to frontend development. Da https://micro-frontends.org/.
30. UXDX, L. Mezzalira. (2019, January 08). Micro Frontends Architecture. Da https://www.youtube.com/watch?reload=9&v=BuRB3djraeM.

Ringraziamenti

Desidero ringraziare innanzitutto il prof. Mario Bravetti per avermi assistito nelle fasi di realizzazione e stesura della tesi. Un dovuto ringraziamento va all'azienda FastCode Consulting Srl. di Cesena per avermi dato l'opportunità di svolgere questo progetto, in particolare a Gianni Mancini per la proposta e a Cesare Gregori per avermi assistito durante il percorso. Non posso non ringraziare il mio gruppo di amici, "La Banda". Grazie al mio amico Michele e al mio coinquilino Camillo, compagno di studio e di risate. Un grazie particolare va alla mia amica di infanzia Arianna su cui posso sempre contare. Infine il ringraziamento più importante va alla mia famiglia per il loro continuo e immancabile supporto.

Grazie.