

SpottyThings

Documento di sviluppo dell'applicazione web.

Indice

Scopo del documento.....	4
User flow.....	5
Application Implementation and Documentation.....	6
Project structures.....	6
Project Dependencies.....	7
Modelli nel database.....	7
Modello <i>User</i>	7
Modello <i>Annuncio</i>	8
Modello <i>Profilo</i>	8
Modello <i>Transazione</i>	9
Modello <i>Recensione</i>	9
Project APIs.....	10
Estrazione delle risorse dal class diagram.....	10
Diagramma delle risorse.....	12
Resource Diagram 1.....	12
Resource Diagram 2.....	12
Resource Diagram 3.....	13
Resource Diagram 4.....	13
Resource Diagram 5.....	14
Resource Diagram 6.....	14
Resource Diagram completo.....	15
Sviluppo API.....	16
API per il modello <i>Utente</i>	16
Salvataggio di un nuovo utente.....	16
Cancellazione di un utente.....	16
Ricerca di tutti gli utenti.....	16
Ricerca dei dati di un preciso utente.....	16
Aggiornamento dati utente.....	17
Aggiornamento email utente.....	17
Aggiornamento password utente.....	17
API per l'autenticazione.....	18
Login di un utente.....	18
Logout di un utente.....	18
Ripristino password.....	18
API per gestire l'invio di una mail.....	18
Invio di una mail.....	18
API per il modello <i>Annuncio</i>	19
Salvataggio di un nuovo annuncio.....	19
Ricerca di tutti gli annunci.....	19
Ricerca delle informazioni di un annuncio.....	19
Ricerca di tutti gli annunci di un preciso inserzionista.....	19
Ricerca di tutti gli annunci che soddisfano determinati filtri.....	19
Ricerca degli annunci secondo una keyword.....	20
Ordinamento di un elenco di annunci secondo un preciso parametro.....	20
Eliminazione di un annuncio.....	20
Aggiornamento delle informazioni di un annuncio.....	20
API per il modello <i>Recensione</i>	21
Salvataggio di una nuova recensione.....	21
Eliminazione di una recensione.....	21

Ricerca di tutte le recensioni.....	21
Ricerca delle recensioni fatte su un preciso utente.....	21
Ricerca delle recensioni fatte da un preciso utente.....	22
Ricerca della recensione relativa ad una transazione.....	22
API per il modello <i>Transazione</i>	23
Salvataggio di una nuova transazione.....	23
Eliminazione di una transazione.....	23
Ricerca di tutte le transazioni.....	23
Ricerca di tutte le transazioni di un utente in qualità di 'acquirente'.....	23
Ricerca di tutte le transazioni di un utente in qualità di 'venditore'.....	24
Ricerca di tutte le transazioni associate ad un preciso prodotto.....	24
API per il modello <i>Profilo</i>	25
Salvataggio di un nuovo profilo.....	25
Ricerca di tutti i profili.....	25
Ricerca delle informazioni di un preciso profilo.....	25
Ricerca dei profili con il rating più alto.....	25
Eliminazione di un profilo.....	25
Aggiunta di un elemento alla wishlist.....	26
Rimozione di un elemento alla wishlist.....	26
Aggiornamento descrizione profilo.....	26
Aggiornamento del rating del profilo.....	26
Aggiornamento statistiche annunci online.....	27
Aggiornamento statistiche di vendita.....	27
Aggiornamento statistiche di acquisti.....	27
API documentation.....	28

Scopo del documento

Il seguente documento riporta tutte le informazioni necessarie per descrivere lo sviluppo di una parte, molto completa, dell'applicazione web *SpottyThings*.

Nel primo capitolo viene riportato lo user flow, ovvero una descrizione tramite diagramma di tutte le azioni che si possono eseguire sulla parte implementata di *SpottyThings*, descrivendo le varie richieste effettuabili a front-end in ogni pagina e le varie risposte possibili.

Successivamente rappresentiamo una struttura del codice realizzato, descrivendo le dipendenze installate, i modelli realizzati e le API implementate.

Una attenta descrizione delle API implementate viene fatta con il diagramma delle risorse e il diagramma di estrazione delle risorse, in cui si individuano le risorse estratte a partire dal diagramma delle classi del documento D3-T25.

Nel capitolo quattro si spiega ciò che si è fatto con *Swagger* per la documentazione delle API.

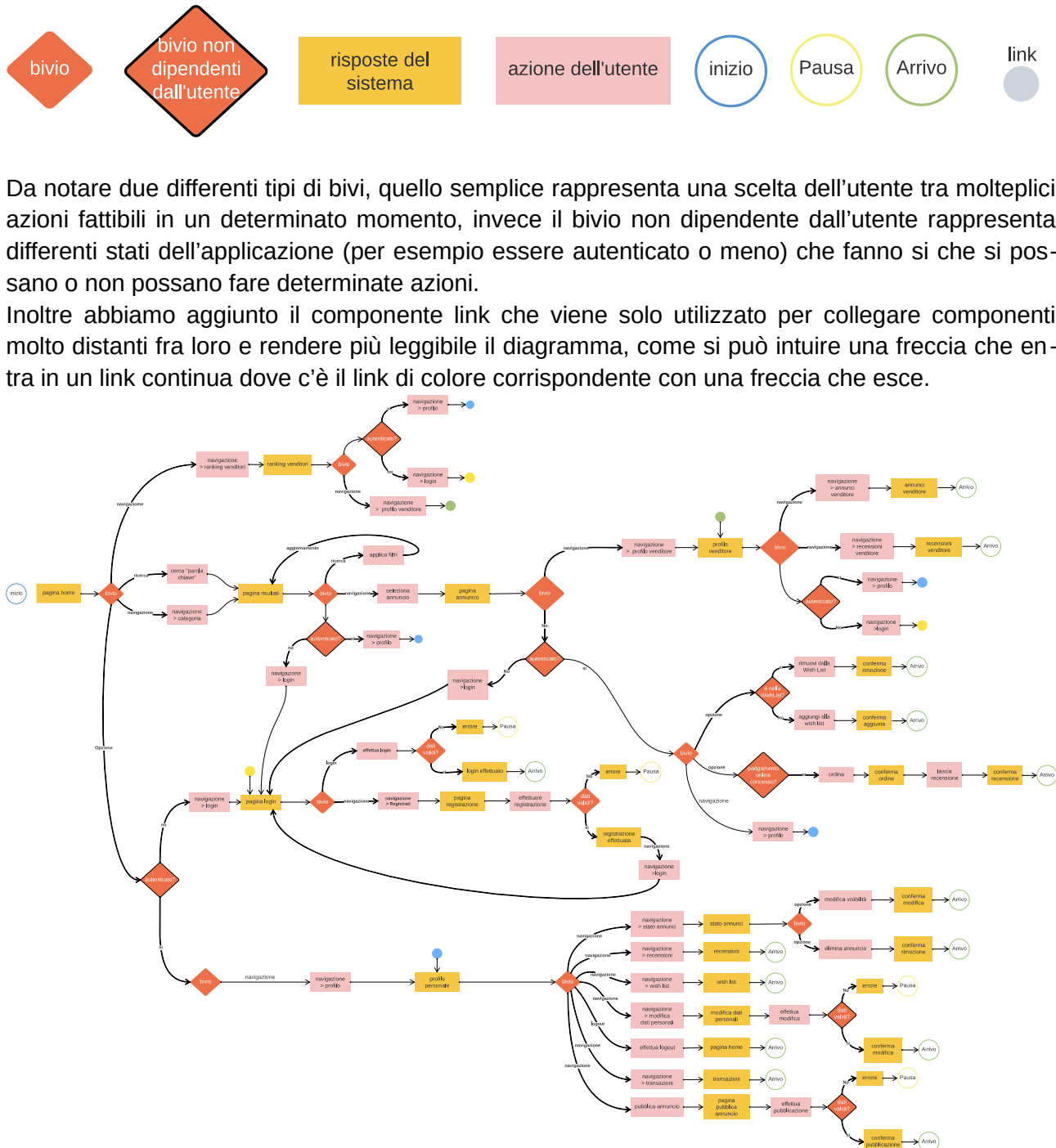
Successivamente viene fornita una breve descrizione per le pagine implementate e una descrizione del repository di GitHub con le istruzioni per effettuare il deployment.

Per finire mostriamo i vari casi di test realizzati per verificare il corretto funzionamento delle API.

User flow

In questa sezione viene riportato lo user-flow dell'applicazione, il quale descrive ciò che è possibile fare nell'implementazione descritta nel dettaglio in questo documento.

Qui sotto è riportata una didascalia dei vari componenti utilizzati nello user-flow.



Application Implementation and Documentation

L'applicazione *SpottyThings* è stata sviluppata utilizzando *NodeJS* e *VueJS* per la parte di front-end. A livello di back-end per la memorizzazione dei vari dati abbiamo utilizzato *MongoDB*.

Come si vede dallo user flow, abbiamo sviluppato tutte le parti riguardanti la registrazione, il login, il ripristino della password, la pubblicazione di un annuncio, la ricerca degli annunci mediante una parola chiave o dei filtri, la classifica degli utenti col rating migliore e il profilo personale. Abbiamo anche realizzato una "simulazione d'acquisto" di un oggetto, per simulare un possibile utilizzo completo da parte di un utente della nostra applicazione.

Infine abbiamo anche utilizzato le API del sistema esterno *NodeMailer* per gestire l'invio di email agli utenti (riguardante il ripristino della password).

Project structures

La struttura del progetto è rappresentata nella immagine sottostante. La directory principale *ProgettoT25* è divisa in due directory distinte: *FrontEnd* e *BackEnd*, in cui sono implementate le rispettive parti di front-end e back-end.

La cartella *BackEnd* è divisa in questo modo:

- cartella *node_modules* dove sono scaricate le dipendenze utilizzate nel sistema
- file *swagger.json* dove sono descritte approfonditamente e con opportuni esempi le API implementate nel sistema
- file *package.json* che rappresenta il file di configurazione generale del progetto
- file *.env* dove sono definite le variabili d'ambiente
- file *index.js* che rappresenta lo start point del progetto, il programma che deve essere eseguito per attivare la connessione verso il database *MongoDB* e attivare il server all'indirizzo *localhost:8080*
- cartella *src* dove viene definita tutta la business logic del sistema lato back-end.

Nella cartella *src* troviamo:

- il file *server.js* che importa i vari moduli che vengono utilizzati a back-end e definisce l'endpoint delle API
- la cartella *auxiliaries* che contiene le definizioni e il codice di tutte le funzioni ausiliarie implementate da noi e utilizzate nelle varie API
- la cartella *models* che contiene la definizione di tutti i modelli che saranno memorizzati nel database
- la cartella *controllers* che contiene il vero e proprio codice di tutte le API del sistema: sono definite le varie funzioni (specificate nel diagramma delle risorse sotto riportato) con cui si esprime la logica delle varie API e in cui si fanno le chiamate e gli accessi al database
- la cartella *routes* in cui sono organizzate le varie API divise per modello e vengono specificati i corrispondenti endpoint.

Mentre nella cartella *FrontEnd*:

**[.gitignore
package.json]**

.env
testing]

Project Dependencies

I moduli Node utilizzati e aggiunti al file *package.json*, nel campo *dependencies*, sono:

- *cors*: modulo per permettere alla web app di supportare il *Cross-Origin Resource Sharing* protocol
- *dotenv*: permette di utilizzare le variabili d'ambiente definite nel file *.env*
- *express*: framework che fornisce molte funzionalità per le web application come *Spotty-Things*, tra cui molte funzioni per creare e gestire le API
- *jsonwebtoken*: modulo per creare e gestire un token d'accesso
- *mongoose*: fornisce le varie funzioni per interagire con *MongoDB*
- *multer*: per gestire il body nelle varie API
- *nodemailer*: utilizzato per gestire l'invio di mail agli utenti
- *swagger-ui-express*: tool usato per documentare e testare la API che abbiamo progettato
- *jest*: modulo usato per il testing delle API e delle funzioni nel back-end
- *supertest*: modulo usato per chiamare le API in fase di testing.

Nel campo *scripts* abbiamo anche aggiunto lo script *"start"*: *"node index.js"* in modo tale da poter avviare il server semplicemente col comando *npm start* e lo script *"test"*: *"jest --coverage"* per eseguire il testing semplicemente col comando *npm test*.

Modelli nel database

Per la gestione dei dati utili nell'applicazione abbiamo definito diversi modelli di dati partendo dalle classi sviluppate nel class diagram del documento *D3-T25*. Le risorse necessarie da gestire nel nostro sistema hanno portato alla definizione di cinque modelli, per ognuno dei quali è stata definita una precisa collezione nel database.

Modello User

Per memorizzare i dati degli utenti che si iscrivono al nostro sito web abbiamo creato il modello *User*. In questa immagine mostriamo lo schema del dato.

[screen schema]

Si pone necessaria la definizione degli attributi sensibili *nome*, *cognome*, *datadinascita* e *indirizzo* perché l'RNF 17 definito nel *D1-T25* richiede esplicitamente di verificare l'identità degli utenti. In questo modo riusciamo a raccogliere dati sensibile degli utenti che confermano la loro reale identità.

L'attributo *metodiPagamento* è opzionale e prevede di associare l'indirizzo mail a cui è collegato un account *PayPal* dell'utente per procedere agli acquisti online direttamente sul sito.

Nel campo *_id* automaticamente creato da *MongoDB* viene salvato lo *username* dell'utente in quanto unico e richiesto per ogni utente che si registra al sistema.

Un esempio di elemento nella collezione nel database è il seguente:

[esempio dati]

Modello Annuncio

Per memorizzare i dati degli annunci che vengono postati su *SpottyThings* abbiamo creato il modello *Annuncio*. In questa immagine mostriamo lo schema del dato.

[screen schema]

L'attributo *modalitaTransazione* specifica se il preciso annuncio risulta postato in *Affitto* o in *Vendita*, mentre l'attributo *pagamentoOnline* specifica se è abilitato o meno il pagamento online per il suddetto annuncio. Nel caso si abilitato è resa obbligatoria la definizione di un indirizzo mail collegato ad un conto *PayPal* in cui verrà accreditato l'importo.

Nel caso di un annuncio in *Vendita* sarà presente l'attributo *prezzo*, mentre per gli annunci in *Affitto* ci saranno tre campi al suo posto: *prezzoAffittoAlGiorno*, *prezzoAffittoSettimanale* e *prezzoAffittoAllOra*. I campi che non vengono usati sono impostati su *undefined* oppure *null*.

Infine *visibile* è l'attributo che regola l'effettiva visibilità pubblica di un annuncio. Dopo che un annuncio viene acquistato, questo viene impostato su *false* (nel caso di un annuncio in *Affitto*, nel caso fosse in *Vendita* l'annuncio viene direttamente rimosso dal sito).

Nel campo *_id* automaticamente creato da *MongoDB* viene salvato il titolo dell'annuncio in quanto unico e richiesto per ogni annuncio che viene postato nel sistema.

Un esempio di elemento nella collezione nel database è il seguente:

[esempio dati]

Modello Profilo

Per memorizzare le attività svolte dai vari utenti nel nostro sistema abbiamo creato il modello *Profilo*. In questa immagine mostriamo lo schema del dato.

[screen schema]

Nello schema si riprende la classe *Profilo* del documento *D3-T25* e si salvano chiaramente tutte le statistiche relative alle vendite e agli acquisti dell'utente sul sito. I vari attributi vengono modificati ogni volta che si eseguono azioni rilevanti, ovvero:

- *rating*, *recensioniFatte* e *recensioniRicevute* dopo ogni recensione fatta oppure ricevuta
- *guadagnoDagliAffitti*, *guadagnoDalleVendite*, *transazioniCompletate* e *prodottiVenduti* ogni volta che un prodotto dell'utente viene acquistato/affittato
- *soldiSpesi* ogni volta che l'utente acquista online un prodotto
- *annunciOnlineVendita* e *annunciOnlineAffitto* ogni volta che l'utente posta un nuovo annuncio, modifica la visibilità di un annuncio oppure un suo annuncio viene acquistato/affittato.

Nel campo *_id* automaticamente creato da *MongoDB* viene salvato lo username dell'utente in quanto ogni profilo è identificato univocamente e obbligatoriamente da un utente, quindi dal profilo si può risalire all'utente proprietario.

Un esempio di elemento nella collezione nel database è il seguente:

[esempio dati]

Modello *Transazione*

Per memorizzare tutte le transazioni eseguite nel sistema (acquisto online di prodotti) abbiamo creato il modello *Transazione*. In questa immagine mostriamo lo schema del dato.

[screen schema]

L'attributo *metodoTransazione* specifica se la transazione è avvenuta online sul sito web oppure se è avvenuta di persona tra i due utenti e viene registrata manualmente, mentre l'attributo *tipologiaTransazione* specifica se la transazione avvenuta è un *Affitto* oppure una *Vendita*. Il *costo* rappresenta l'importo che è stato incassato direttamente dal venditore.

Un esempio di elemento nella collezione nel database è il seguente:

[esempio dati]

Nel campo *_id* automaticamente creato da *MongoDB* viene salvata una stringa generata dalla concatenazione dell'iniziale del venditore, dell'iniziale dell'acquirente, dell'iniziale del titolo dell'annuncio e della data in ms in cui è stata creata la transazione. In questo modo viene assegnata ad ogni transazione un identificativo univoco.

Modello *Recensione*

Per memorizzare tutte le recensioni eseguite nel sistema (effettuate da un compratore dopo un acquisto) abbiamo creato il modello *Recensione*. In questa immagine mostriamo lo schema del dato.

[screen schema]

Siccome la recensione è associata ad una precisa transazione effettuata l'attributo *transazioneRecensita* permette di risalire quale transazione è stata recensita. Le recensioni vengono raccolte dopo che si effettua un ordine online per valutare il comportamento del venditore durante la transazione.

Nel campo *_id* automaticamente creato da *MongoDB* viene salvato l'id della transazione recensita, in quanto ogni recensione fa riferimento ad un'unica transazione nel sistema.

Un esempio di elemento nella collezione nel database è il seguente:

[esempio dati]

Project APIs

In questa parte del documento vengono descritte le varie API implementate a partire dal diagramma delle classi del documento D3-T25. Useremo un diagramma per rappresentare l'estrazione delle risorse a partire dal class diagram e uno per rappresentare le risorse sviluppate.

Estrazione delle risorse dal class diagram

Questo diagramma mostra come abbiamo estratto le varie risorse sviluppate nel sistema a partire dal class diagram.

Inizialmente le prime risorse che abbiamo individuato sono stati i 5 modelli che abbiamo descritto nel capitolo **Modelli nel database**.

A partire dalle classi presenti abbiamo individuato i “tipi di dato” che dovevano essere memorizzati nel database in *MongoDB* preservandone gli attributi fondamentali. Infatti gli attributi specificati nel class diagram sono stati riportati anche nel diagramma delle risorse.

Per quanto riguarda la risorsa *Profilo* abbiamo omesso gli attributi *storicoTransazioni* e *recensioni-Ricevute*: in quanto array di altri oggetti memorizzati nel database avremmo avuto una risorsa troppo onerosa in termini di memoria occupata e velocità nella ricerca per cui gli abbiamo omessi. Tuttavia questa omissione non porta alcun svantaggio in quanto abbiamo definito dei metodi (nella risorsa *Annuncio* e *Recensione*) per ottenere i due array sopra descritti in modo molto semplice e veloce.

Successivamente abbiamo trasformato alcuni metodi delle classi in ulteriori risorse del nostro sistema.

Le risorse a cui sono collegati i modelli sopra descritti sono nient'altro che le API che abbiamo sviluppato e coincidono con alcuni metodi delle classi. Non tutti i metodi sono chiaramente diventati API in quanto alcuni sono semplicemente delle piccole funzioni ausiliarie e di supporto alle API.

Di queste risorse viene specificato il metodo (se si tratta di GET, POST, PATCH o DELETE a seconda del loro compito) e i parametri che richiedono per essere eseguiti (si è specificato body nel caso in cui servissero più di tre parametri in input per dire che quella data risorsa richiede tutti, o comunque molti, attributi del modello).

Infine viene specificato anche se l'effetto di quella risorsa ha rilevanza nel front-end oppure nel back-end: per le varie risorse i tipo POST, PATCH e DELETE l'effetto è chiaramente sul back-end perché il loro compito è di salvare, modificare o eliminare una risorsa nel database, non fornendo informazioni in front-end oltre che un messaggio di conferma. Le risorse di tipo GET hanno invece un chiaro effetto sul front-end perché interrogano il database per chiedere alcune risorse e poi le mostrano nel front-end.

Nella pagina seguente alleghiamo il diagramma che illustra quanto descritto, mentre una specifica più attenta delle API sarà fatta nel **diagramma delle risorse**.

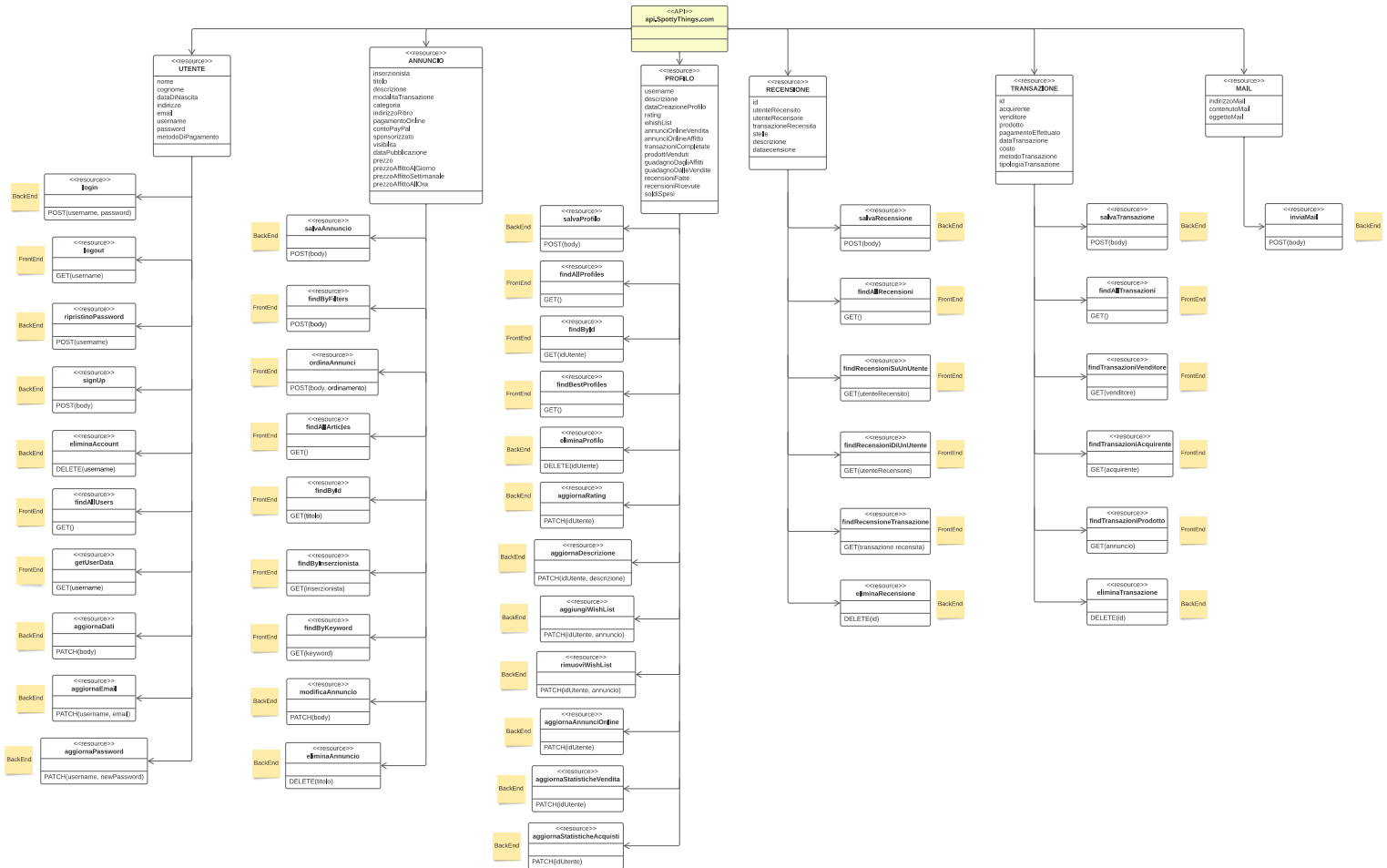


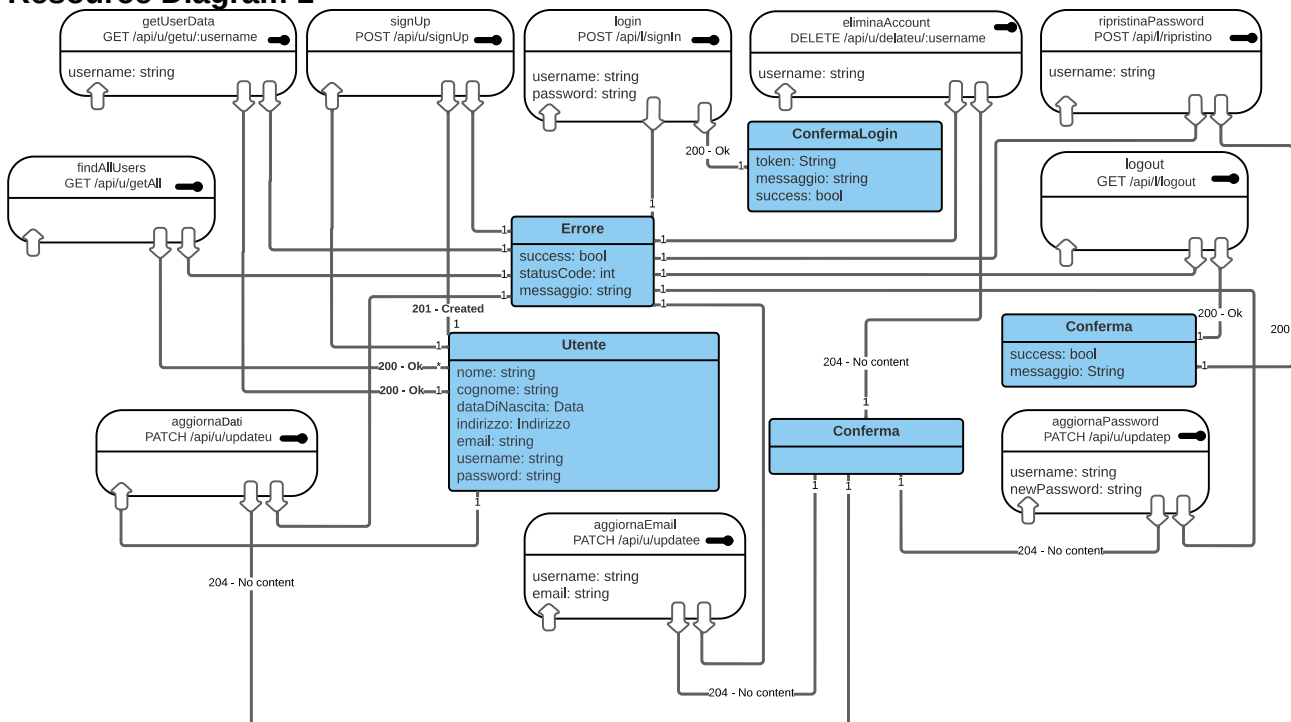
Diagramma delle risorse

Nel seguente diagramma delle risorse rappresentiamo le varie API sviluppate nel progetto. Per permettere una visione più modulare e chiara, abbiamo diviso il diagramma delle risorse per modello.

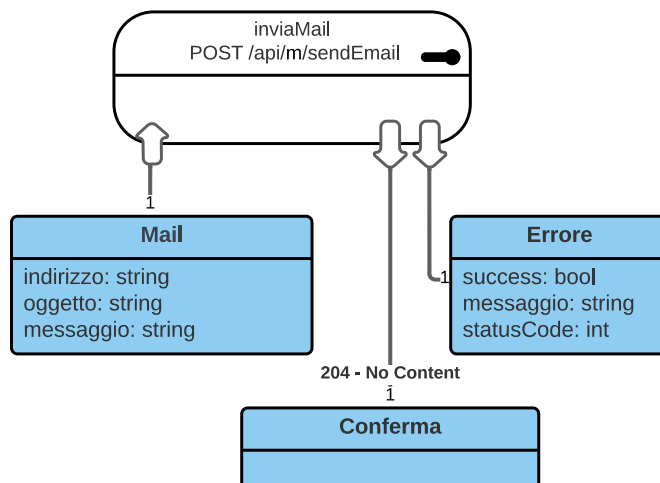
In tutti i diagrammi sono stati specificati gli input e gli output delle varie API. Le varie API possono ritornare errori di varia natura, quindi anziché creare un output per ogni possibile errore abbiamo creato un output comune per tutti gli errori. Infatti in ogni API il messaggio di errore è standardizzato: il campo *success* (impostato a *false*) e il campo *message*, che spiega cosa è andato storto o non è possibile fare, oltre che il codice di stato che identifica l'errore corrispondente.

Per le API che ritornano un codice *204 – No content* abbiamo associato un messaggio di conferma vuoto, in quanto appunto non ritornano nulla se non il codice 204 che conferma l'esecuzione dell'API con successo.

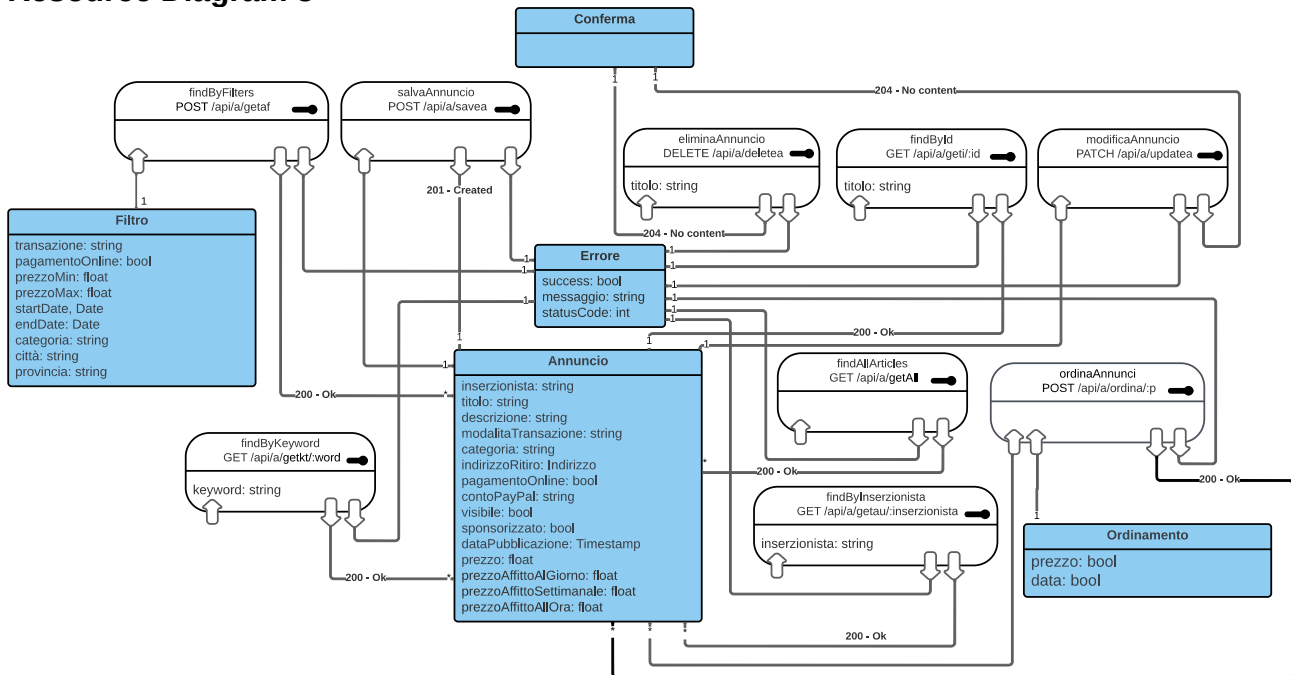
Resource Diagram 1



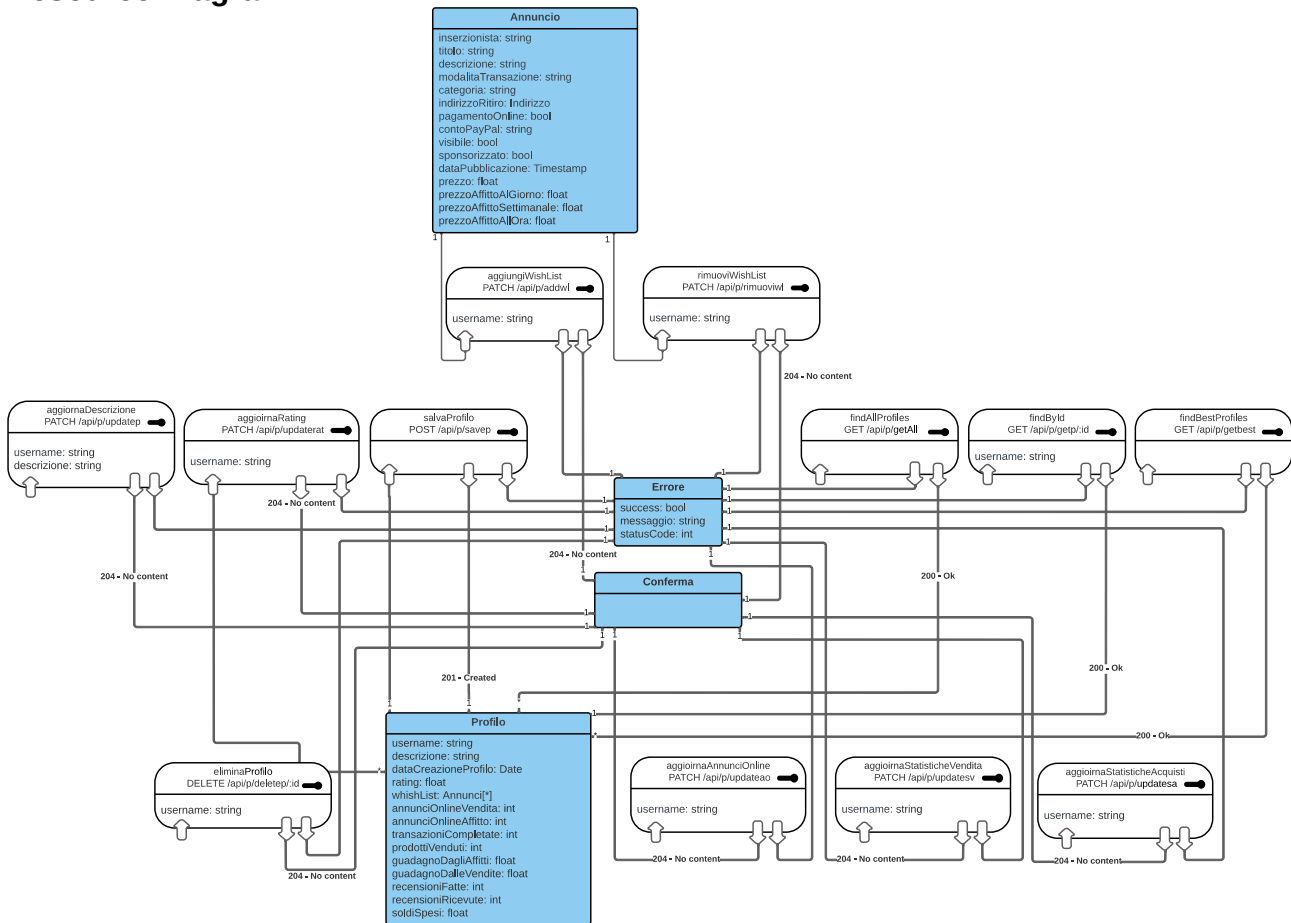
Resource Diagram 2



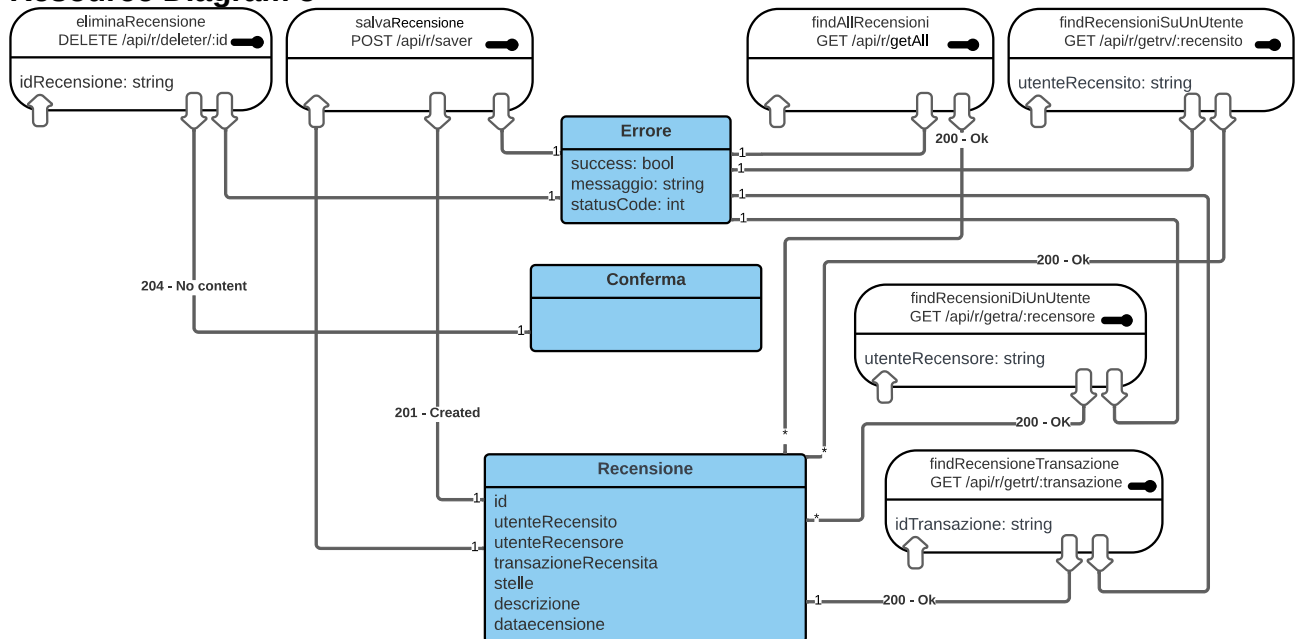
Resource Diagram 3



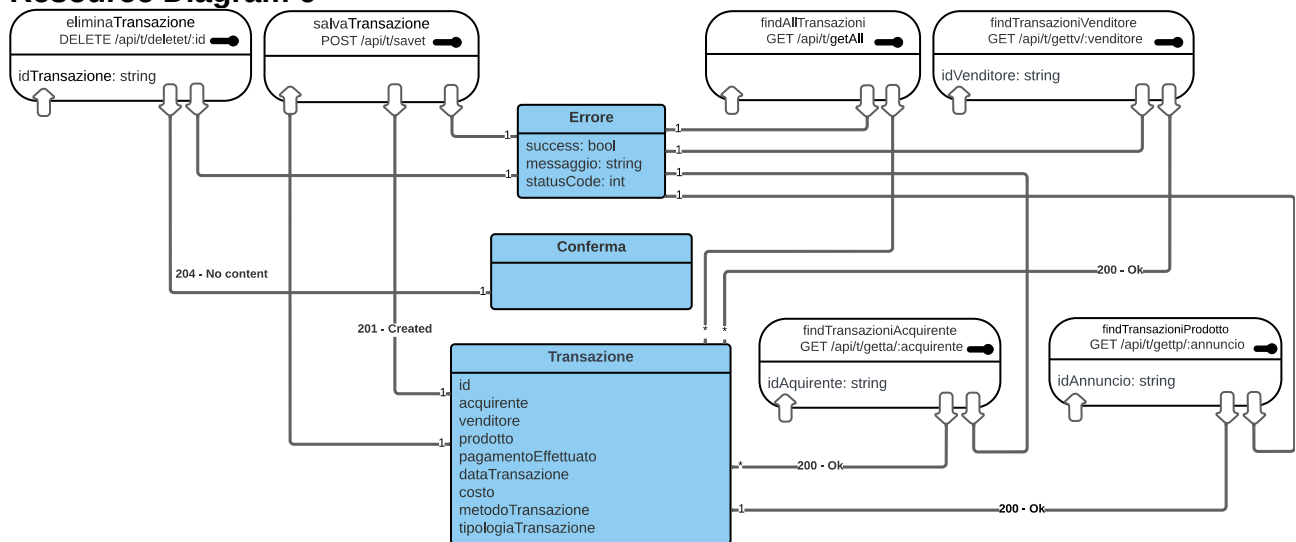
Resource Diagram 4



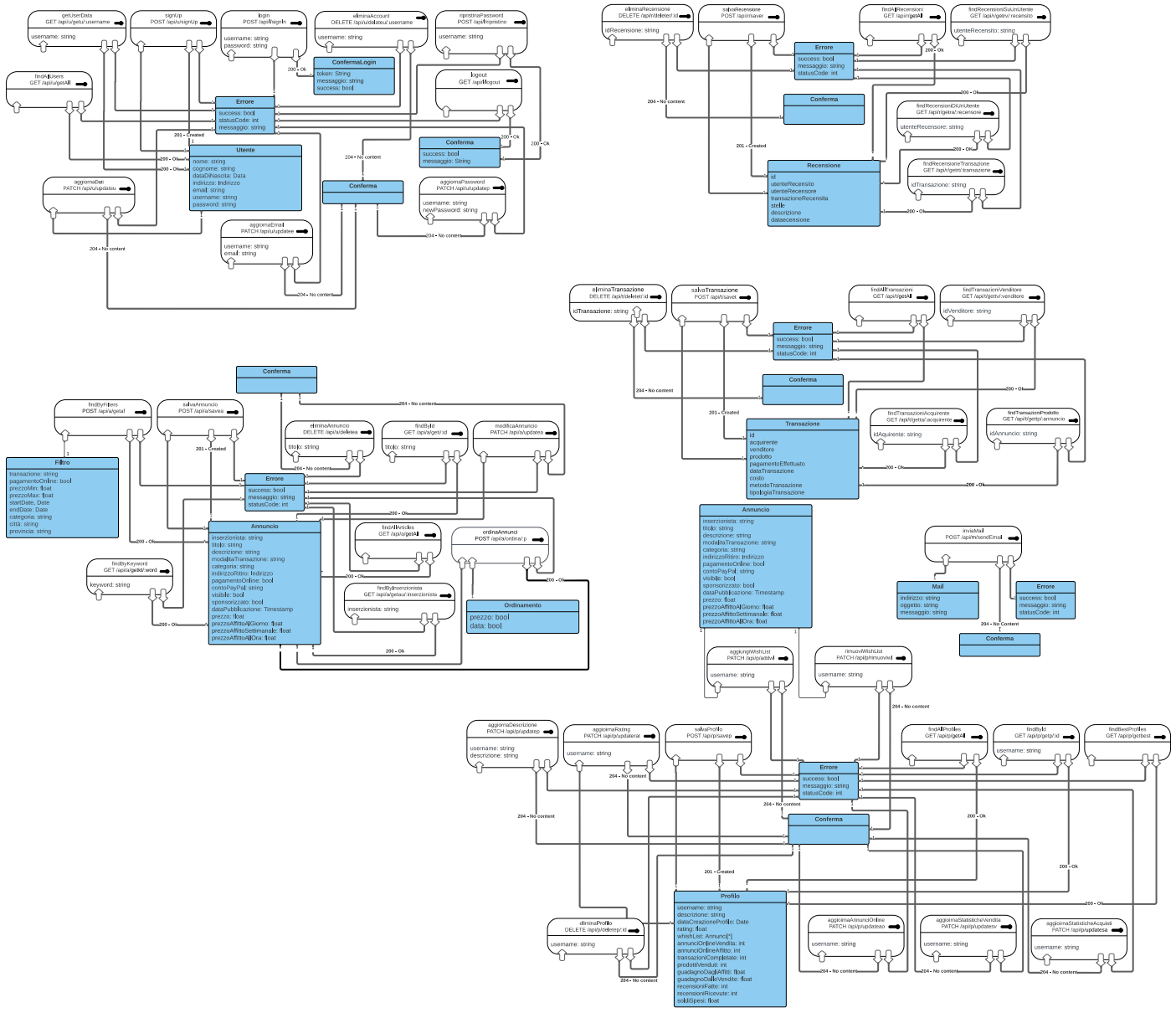
Resource Diagram 5



Resource Diagram 6



Resource Diagram completo



Sviluppo API

In seguito descriveremo il funzionamento delle varie API sviluppate nel progetto e le varie funzioni di *MongoDB* chiamate al loro interno.

Non alleghiamo il codice della API in quanto è molto lungo per ognuna di esse e può essere semplicemente visto nella repository (dalla cartella *BackEnd* si va in *src* e poi in *controllers*: qua si trovano tutte le API sviluppate divise in diversi file in base al modello di riferimento).

API per il modello *Utente*

Di seguito descriviamo le API per gestire le varie azioni eseguibili sulla risorsa *Utente*.

Salvataggio di un nuovo utente

L'API *signUp* ha lo scopo di creare una nuova istanza della risorsa *Utente* e salvarla nel database. Riceve in input un body con tutti i dati dell'utente che devono essere memorizzati: sono tutti richiesti tranne il campo *metodiPagamento* che può essere tralasciato.

Fornisce in output la risorsa appena creata se tutto è andato a buon fine oppure diversi messaggi di errori se l'username specificato o l'email specificata sono già occupati da un altro utente.

Si utilizza il metodo *findOne()* per verificare questi criteri di unicità di username e password e successivamente il metodo *save()* per salvare la risorsa nel database.

Cancellazione di un utente

L'API *eliminaAccount* ha lo scopo di eliminare un'istanza della risorsa *Utente* dal database.

Riceve come unico parametro di input lo username dell'utente che si intende eliminare. Se tutto è andato a buon fine non ritorna niente altrimenti se lo username specificato non esiste lo comunica con un messaggio di errore.

Utilizza il metodo *findOne()* per ricercare l'utente e il metodo *deleteOne()* per eliminare l'utente con lo username specificato.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato che intende eliminare definitivamente il proprio account.

Ricerca di tutti gli utenti

L'API *findAllUsers* ricerca nel database tutti gli utenti del sistema. È un API di supporto in quanto usata solo da noi sviluppatori per questioni di gestione del codice e non è mai realmente chiamata a front-end. Ritorna un elenco di tutti i possibili utenti oppure un messaggio di errore se non è presente nessun utente salvato.

Utilizza il metodo *find()* con il primo argomento *{}* per ottenere tutti gli utenti memorizzati.

Ricerca dei dati di un preciso utente

L'API *getUserData* ha lo scopo di accedere al database per fornire i dati di un utente il cui username viene specificato come unico parametro in input. Accede al database e ottiene la risorsa desiderata con il metodo *findOne()* con primo argomento *{username : req.params.username}* per ottenere l'utente che si desidera.

Se non trova nessun utente con lo username associato ritorna un errore 404.

Aggiornamento dati utente

L'API *aggiornaDati* ha lo scopo di modificare una precisa risorsa di tipo *Utente* memorizzata nel database. Riceve in input un body con lo username dell'utente da modificare e gli attributi che si intendono sovrascrivere.

Si utilizza il metodo *findOne()* con primo argomento `{username : req.params.username}` per ottenere l'utente che si desidera (se nessun utente possiede lo username specificato si ritorna un messaggio d'errore) e infine il metodo *updateOne()* specificando lo username dell'utente da modificare e gli attributi da modificare per effettuare la modifica.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato.

Aggiornamento email utente

L'API *aggiornaEmail* ha lo scopo di modificare l'indirizzo email di una precisa risorsa di tipo *Utente* memorizzata nel database. Riceve in input un body con solamente lo username dell'utente da modificare e il nuovo indirizzo email da sovrascrivere.

Si utilizza il metodo *findOne()* con primo argomento `{username : req.params.username}` per ottenere l'utente che si desidera (se nessun utente possiede lo username specificato si ritorna un messaggio d'errore) e infine il metodo *updateOne()* specificando lo username dell'utente da modificare e l'indirizzo email che sostituirà quello vecchio. Se l'email è già in uso da un altro utente si ritorna un messaggio d'errore.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato.

Aggiornamento password utente

L'API *aggiornaPassword* ha lo scopo di modificare la password di una precisa risorsa di tipo *Utente* memorizzata nel database. Riceve in input un body con solamente lo username dell'utente da modificare e la nuova password da sovrascrivere.

Si utilizza il metodo *findOne()* con primo argomento `{username : req.params.username}` per ottenere l'utente che si desidera (se nessun utente possiede lo username specificato si ritorna un messaggio d'errore) e infine il metodo *updateOne()* specificando lo username dell'utente da modificare e la password che sostituirà quella vecchia.

Non vengono fatti controlli se la nuova password soddisfi i requisiti di sicurezza richiesti dal RNF 1 specificato nel documento D1-T25 in quanto vengono già effettuati a front-end prima di chiamare l'API.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato.

API per l'autenticazione

Di seguito descriviamo le API per gestire l'autenticazione di un utente.

Login di un utente

L'API *login* ha lo scopo di autenticare un preciso utente che intende accedere al proprio account. Riceve in input lo username e la password di un utente e utilizza il metodo *findOne()* per verificare l'esistenza di un utente col dato username e successivamente, in caso di esito positivo del precedente controllo, verifica la password fornita con quella dell'utente ottenuto. In caso di utente non esistente o password errata ritorna un errore, altrimenti crea un token d'accesso della durata di 12 ore (grazie al metodo *sign()* di *jsonwebtoken*) che ritorna in output e servirà per chiamare le varie API che richiedono un token d'accesso per essere eseguite.

Logout di un utente

L'API *logout* ha lo scopo di permettere l'uscita dal proprio account a un preciso utente. Riceve in input il token d'accesso attualmente attivo per l'utente e fornisce in output un messaggio di conferma relativo alla corretta disattivazione del token.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato.

Ripristino password

L'API *ripristinoPassword* viene eseguita con lo scopo di fornire una nuova password all'utente il cui username viene specificato come unico parametro del body in input a questa funzione.

È diversa dall'API *aggiornaPassword* in quanto nel primo caso è l'utente in prima persona ha richiedere di sostituire una password con una nuova da lui fornita, mentre qua la password dell'utente viene ripristinata automaticamente con una stringa di 13 caratteri casuali conforme ai requisiti di sicurezza.

Si utilizza il metodo *findById()* per cercare l'utente con lo username specificato (in quanto lo username funge anche da *_id* dell'utente) e successivamente il metodo *updateOne()* per modificare la password. Per generare la password casualmente si utilizza il metodo *createRandomString()* definito da noi nella cartella *auxiliaries*.

API per gestire l'invio di una mail

Di seguito descriviamo l'API che utilizziamo per mandare automaticamente una mail ad un utente. Il suo funzionamento si basa sulle API esterne di *nodemailer*.

Invio di una mail

L'API *inviaMail* ha lo scopo di inviare automaticamente un messaggio ad un indirizzo email specificato. Riceve in input l'indirizzo email del destinatario, l'oggetto della mail e il messaggio da mandare.

Si usano i metodi forniti dalle API di *nodemailer*: il metodo *createTransporter* permette di eseguire l'accesso all'account GMAIL del sistema *Spottythings*, con le credenziali specificate nel file *.env*, mentre il metodo *sendMail* permette di inviare una mail fornendo indirizzo destinatario e mittente, oggetto e messaggio da inviare, forniti nel body della richiesta.

API per il modello *Annuncio*

Di seguito descriviamo le API per gestire le varie azioni eseguibili sulla risorsa *Annuncio*.

Salvataggio di un nuovo annuncio

L'API *salvaAnnuncio* ha lo scopo di creare una nuova istanza della risorsa *Annuncio* e di salvarla nel database.

Riceve in input alla richiesta un body con i vari attributi della risorsa annuncio che viene creata.

Viene fornita in output la risorsa appena creata se tutto è andato a buon fine. In caso contrario si restituisce un messaggio d'errore (nel caso in cui la richiesta non sia ben formulata oppure si specifichi un titolo di un annuncio già presente nel database oppure lo username di un inserzionista non esistente).

Vengono chiamati i metodi *findOne()* sulle collezioni di *Annuncio* e *User* in MongoDB per verificare unicità del titolo e presenza dell'inserzionista e infine il metodo *save()* per salvare l'annuncio nel database. Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato.

Ricerca di tutti gli annunci

L'API *findAllArticles* ha lo scopo di ricercare ed elencare tutti gli annunci salvati nel database. È un API di supporto in quanto mai chiamata a front-end, ma usata solo da noi sviluppatori per gestione del codice e debugging.

Non c'è niente in input alla richiesta, ma viene fornito in output l'elenco di tutti gli annunci presenti oppure un messaggio di errore se nessun annuncio è salvato nel database.

Per ottenere l'elenco di tutti gli annunci salvati nel sistema si usa il metodo *find()* con primo argomento `{}`.

Ricerca delle informazioni di un annuncio

L'API *findById* ha lo scopo di ricercare un preciso annuncio nel database, dato il suo titolo passato come unico parametro in input alla richiesta. Viene fornita in output la precisa risorsa richiesta se presente, altrimenti si mostra un messaggio d'errore.

Viene utilizzato il metodo *findById()* con primo argomento *req.params.id* per andare a prendere dal database l'annuncio con l'id specificato.

Ricerca di tutti gli annunci di un preciso inserzionista

L'API *findByInserzionista* ha lo scopo di ricercare ed elencare tutti gli annunci effettuati da un preciso inserzionista. Riceve in input come unico parametro lo username dell'utente desiderato.

In output dalla richiesta si ottiene l'elenco di tutti gli annunci ricercati oppure un messaggio d'errore se nessun annuncio è associato all'utente cercato.

Viene usato il metodo *find()* con primo argomento `{inserzionista : req.params.inserzionista}` per ottenere l'elenco di tutti gli annunci desiderati.

Ricerca di tutti gli annunci che soddisfano determinati filtri

L'API *findByFilters* ha lo scopo di ricercare ed elencare tutti gli annunci che soddisfano un insieme di filtri. Quest'API riceve in input alla richiesta un body contenente tutti i valori dei filtri per cui si intende ricercare degli annunci. Viene fornito in output l'elenco degli annunci che soddisfano i filtri forniti oppure un messaggio d'errore se nessun annuncio è concorde con i filtri.

In questa funzione si usa il metodo *find()* con primo argomento *{}* per ottenere tutti gli annunci del sistema e poi la funzione *filterArray()* definita da noi nella cartella *auxiliaries* per filtrare l'elenco degli annunci ottenuti mantenendo solo quelli desiderati.

Ricerca degli annunci secondo una keyword

L'API *findByKeyword* ha lo scopo di ricercare ed elencare tutti gli annunci che soddisfano una precisa keyword.

Quest'API riceve in input una parola e fornisce in output l'elenco di tutti gli annunci che contengono la parola fornita in input all'interno del titolo (indipendentemente dalle maiuscole e dalle minuscole). Se nessun annuncio nel database contiene la keyword specificata nel titolo, si ritorna un messaggio d'errore.

Viene utilizzato il metodo *find()* con primo argomento *{}* per ottenere tutti gli annunci del sistema e poi la funzione *filterByTerm()* definita da noi nella cartella *auxiliaries* per filtrare l'elenco degli annunci ottenuti mantenendo solo quelli desiderati.

Ordinamento di un elenco di annunci secondo un preciso parametro

L'API *ordinaAnnunci* ha lo scopo di ordinare un elenco di annunci fornito in input secondo un preciso ordinamento fornito come parametro (che riguarda la data di pubblicazione oppure il prezzo).

Quest'API non effettua chiamate al database, ma usa delle funzioni definite da noi nella cartella *auxiliaries* per effettuare il sorting dell'elenco di annunci fornito in input.

In output viene ottenuto lo stesso elenco di annunci fornito ma ordinato secondo l'ordinamento specificato in input. Se quest'ultimo è mal formattato viene ritornato un errore.

Eliminazione di un annuncio

L'API *eliminaAnnuncio* ha lo scopo di eliminare un'istanza della risorsa *Annuncio* dal database.

Riceve come unico parametro di input il titolo dell'annuncio che si intende eliminare. Se tutto è andato a buon fine non ritorna niente altrimenti se il titolo specificato non è associato a nessun annuncio viene comunicato un messaggio di errore.

Si utilizza il metodo *findById()* per ricercare l'annuncio e il metodo *deleteOne()* per eliminare l'annuncio con il titolo specificato.

Per eseguire quest'API non è necessario fornire un token d'accesso che certifica che l'utente che la esegue è l'utente proprietario dell'annuncio, perché può essere eseguita anche dal sistema in sé: dopo che un annuncio è stato comprato online, viene chiamata questa API per rimuovere il suddetto annuncio dal sito web.

Aggiornamento delle informazioni di un annuncio

L'API *modificaAnnuncio* ha lo scopo di modificare una precisa risorsa *Annuncio*, il cui id è passato come parametro in input alla richiesta. Quest'API prende in input anche un body con i vari attributi che si intendono sovrascrivere per la risorsa nel database.

Se la modifica è andata a buon fine non si ritorna niente, altrimenti si fornisce un messaggio d'errore se la richiesta è mal formulata oppure se il titolo fornito non è associato a nessun annuncio.

Si utilizza il metodo *findOne()* per ricercare l'annuncio e il metodo *updateOne()* per modificare l'annuncio appena individuato, sovrascrivendo gli attributi specificati nel body.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato, proprietario dell'annuncio.

API per il modello *Recensione*

Di seguito descriviamo le API per gestire le varie azioni eseguibili sulla risorsa *Recensione*.

Salvataggio di una nuova recensione

L'API *salvaRecensione* ha lo scopo di creare una nuova istanza della risorsa *Recensione* e salvarla nel database.

Riceve in input un body con i vari attributi della recensione che dovranno essere salvati nella risorsa del database. L'attributo *dataRecensione* si riferisce al momento in cui la recensione viene creata quindi viene creato al momento prendendo la data di quando avviene il salvataggio nel database. Vengono utilizzati i metodi *findById()* per verificare l'unicità della recensione e della transazione e successivamente il metodo *save()* per salvare la recensione nel database.

Se tutto è andato a buon fine viene fornita in output la risorsa appena creata, altrimenti si torna un errore se è già stata salvata una recensione per una data transazione, se gli utenti non coincidono con la transazione o se la recensione è mal formulata.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato.

Eliminazione di una recensione

L'API *eliminaRecensione* ha lo scopo di eliminare una precisa risorsa di tipo *Recensione* dal database. Riceve in input un parametro che rappresenta l'id della recensione da cancellare.

Viene utilizzato il metodo *findById()* per cercare la precisione recensione da eliminare e, in caso positivo (altrimenti si ritorna un errore), si usa il metodo *deleteOne()* per eliminare la recensione appena individuata.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato, in quanto una recensione può essere eliminata solo dall'utente autore di essa.

Ricerca di tutte le recensioni

L'API *findAllRecensioni* ha lo scopo di ricercare ed elencare tutte le recensioni salvate nel database. Fornisce in output un elenco di recensioni oppure un errore se nessuna recensione è salvata nel database.

Si usa il metodo *find()* con primo argomento *{}* per ottenere dal database tutte le recensioni salvate.

È un API di supporto in quanto usata solo da noi sviluppatori per questioni di gestione del codice e non è mai realmente chiamata a front-end.

Ricerca delle recensioni fatte su un preciso utente

L'API *findRecensioniSuUnUtente* ha lo scopo di ricercare ed elencare tutte le risorse di tipo recensione che valutano un preciso utente, il cui username è fornito come unico parametro di input.

Se nessuna recensione è associata al dato utente viene tornato un errore, altrimenti l'elenco delle recensioni su quell'utente.

Si usa il metodo *find()* con primo argomento *{username : req.params.recensito}* per ottenere dal database l'elenco di tutte le recensioni di interesse.

Ricerca delle recensioni fatte da un preciso utente

L'API *findRecensioniDiUnUtente* ha lo scopo di ricercare ed elencare tutte le risorse di tipo recensione fatte da un preciso utente, il cui username è fornito come unico parametro di input.

Se nessuna recensione è stata fatta dal dato utente viene tornato un errore, altrimenti l'elenco delle recensioni di quell'utente.

Si usa il metodo *find()* con primo argomento *{username : req.params.recensore}* per ottenere dal database l'elenco di tutte le recensioni di interesse.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato, in quanto solo l'utente autore delle recensioni può vedere l'elenco di tutte le recensioni da lui fatte.

Ricerca della recensione relativa ad una transazione

L'API *findRecensioneTransazione* ha lo scopo di verificare se per una precisa transazione, il cui id è dato come unico parametro in input, ha associato o meno una transazione.

Si usa il metodo *find()* con primo argomento *{transazioneRecensita : req.params.transazione}* per ottenere dal database la recensione associata alla precisa transazione richiesta come parametro.

In caso in cui nessuna recensione sia associata alla transazione fornita viene tornato un errore.

API per il modello *Transazione*

Di seguito descriviamo le API per gestire le varie azioni eseguibili sulla risorsa *Transazione*.

Salvataggio di una nuova transazione

L'API *salvaTransazione* ha lo scopo di creare una nuova istanza della risorsa *Transazione* e salvarla nel database. Riceve in input un body con i vari dati da salvare di una transazione, tranne la data che viene creata al momento in cui la transazione viene salvata nel database. Anche l'id della transazione viene creato al momento grazie alla funzione *createId* definita da noi nella cartella *auxiliaries*.

Se tutto va a buon fine ritorna l'oggetto appena creato, altrimenti se uno degli utenti non esiste oppure l'annuncio per cui si fa la transazione non esiste oppure gli utenti e l'annuncio non concordano si ritorna un errore.

Viene usato il metodo *findOne()* per verificare gli utenti e l'annuncio e infine il metodo *save()* per salvare la nuova recensione nel database.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato, il quale comprando un prodotto online causa la creazione di una transazione.

Eliminazione di una transazione

L'API *eliminaTransazione* ha lo scopo di eliminare una precisa risorsa salvata nel database, identificata dal suo preciso id passato come unico parametro in input alla richiesta.

Viene usato il metodo *findById()* con primo argomento *req.params.id* per verificare l'effettiva presenza di una transazione con il dato id e successivamente, se presente (se assente si torna un errore) si usa il metodo *deleteOne()* per eliminare la transazione.

È un API di supporto in quanto usata solo da noi sviluppatori per questioni di gestione del codice e non è mai realmente chiamata a front-end.

Ricerca di tutte le transazioni

L'API *findAllTransazioni* ha lo scopo di ricercare ed elencare tutte le transazioni salvate nel database. Fornisce in output un elenco di transazioni oppure un errore se nessuna recensione è salvata nel database.

Si usa il metodo *find()* con primo argomento *{}* per ottenere dal database tutte le transazioni salvate.

È un API di supporto in quanto usata solo da noi sviluppatori per gestione del codice e non è mai realmente chiamata a front-end.

Ricerca di tutte le transazioni di un utente in qualità di 'acquirente'

L'API *findTransazioniAcquirente* ha lo scopo di ricercare ed elencare tutte le transazioni in cui un utente, il cui username è passato come unico parametro in input, appare come acquirente.

Fornisce in output l'elenco di transazioni desiderato oppure un errore se l'utente specificato non ha effettuato alcuna transazione.

Si usa il metodo *find()* con primo argomento *{acquirente : req.params.acquirente}* per ottenere tutte le transazioni desiderate.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato, il quale è l'unico che può vedere lo storico degli ordini effettuati nel sistema.

Ricerca di tutte le transazioni di un utente in qualità di 'venditore'

L'API *findTransazioniVenditori* ha lo scopo di ricercare ed elencare tutte le transazioni in cui un utente, il cui username è passato come unico parametro in input, appare come venditore del prodotto in questione.

Fornisce in output l'elenco di transazioni desiderato oppure un errore se l'utente specificato non ha effettuato alcuna transazione.

Si usa il metodo *find()* con primo argomento *{venditore : req.params.venditore}* per ottenere tutte le transazioni desiderate.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato, il quale è l'unico che può vedere tutti i prodotti che sono stati venduti o affittati da altri.

Ricerca di tutte le transazioni associate ad un preciso prodotto

L'API *findTransazioniProdotto* ha lo scopo di ricercare ed elencare tutte le transazioni associate ad un preciso prodotto, il cui titolo è passato come unico parametro in input. Si tratta di eventualmente un elenco di transazioni in quanto per un singolo prodotto in affitto possono esserci più transazioni in diverso tempo.

Fornisce in output l'elenco delle transazioni desiderate oppure un errore se nessuna è presente e si usa il metodo *find()* con primo argomento *{prodotto: req.params.annuncio}* per ottenere tutte le transazioni desiderate.

API per il modello *Profilo*

Di seguito descriviamo le API per gestire le varie azioni eseguibili sulla risorsa *Profilo*.

Salvataggio di un nuovo profilo

L'API *salvaProfilo* ha lo scopo di creare una nuova istanza del modello *Profilo* e salvarlo nel database.

La richiesta prende in input un body con il valore degli attributi da salvare e fornisce in output un messaggio di errore se la richiesta è mal formulata oppure se si vuole registrare un profilo per un utente non presente nel database oppure se il profilo per l'utente specificato esiste già. Se invece la creazione è andata a buon fine viene ritornata l'istanza appena creata.

Si usa il metodo *findOne()* per verificare l'unicità del profilo e l'esistenza dell'utente, mentre viene usato il metodo *save()* per salvare la risorsa appena creata.

Ricerca di tutti i profili

L'API *findAllProfiles* ricerca nel database tutti i profili salvati nel sistema. È un API di supporto in quanto usata solo da noi sviluppatori per questioni di gestione del codice e non è mai realmente chiamata a front-end. Ritorna un elenco di tutte le risorse *Profilo* presenti oppure un messaggio di errore se non è presente nessun profilo salvato.

Utilizza il metodo *find()* con primo argomento *{}* per ottenere tutti l'elenco desiderato.

Ricerca delle informazioni di un preciso profilo

L'API *findById* ricerca nel database una precisa risorsa *Profilo*, il cui id viene specificato come unico parametro in input nella richiesta.

In output viene fornita la precisa risorsa cercata oppure un messaggio d'errore nel caso in cui nessuna risorsa sia associata all'id specificato.

Si utilizza il metodo *findById()* per ricercare ed ottenere la risorsa desiderata.

Ricerca dei profili con il rating più alto

L'API *findBestProfiles* ricerca nel database tutti i profili e li ritorna ordinati dal rating più alto al rating più basso.

Fornisce in output l'elenco dei profili opportunamente ordinato oppure un errore se nessun profilo è salvato nel database.

Si utilizza il metodo *find()* con primo argomento *{}* per ottenere l'elenco dei profili, a cui si applica il metodo *sort()* che va ad ordinarli per rating decrescente.

Eliminazione di un profilo

L'API *eliminaProfilo* ha lo scopo di eliminare una precisa risorsa *Profilo* salvata nel database. Riceve come unico parametro in input l'id del profilo da eliminare. Se l'eliminazione va a buon fine non si ritorna nulla, altrimenti si mostra un messaggio d'errore (se il profilo con l'id specificato non è presente).

Si utilizza il metodo *findById()* per verificare l'esistenza del profilo e successivamente si eliminano tutti gli annunci fatti dall'utente, con il metodo *deleteMany()* con argomento *{inserzionista : req.params.id}* applicato sulla collezione *Annuncio*, e la precisa risorsa profilo con il metodo *deleteOne()* con parametro *{_id : req.params.id}*.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato che intende eliminare definitivamente il proprio profilo.

Aggiunta di un elemento alla wishlist

L'API *aggiungiWhishList* ha lo scopo di aggiungere un nuovo annuncio alla wishlist sul profilo di un utente. Riceve in input l'id del profilo e l'annuncio che si intende aggiungere.

Se l'operazione va a buon fine non si torna nulla, altrimenti un messaggio di errore per comunicare che non c'è nessun profilo associato all'id specificato oppure che l'annuncio che si intende aggiungere non esiste nel database oppure che l'annuncio è già presente nella wishlist.

Vengono utilizzati i metodi *findByld()* sulle collezioni *Annuncio* e *Profilo* per verificare l'esistenza dell'annuncio e del profilo e il metodo *updateOne()* per modificare la wishlist dell'utente richiedente. Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato.

Rimozione di un elemento alla wishlist

L'API *rimuoviWhishList* ha lo scopo di rimuovere un annuncio dalla wishlist sul profilo di un utente. Riceve in input l'id del profilo e il titolo dell'annuncio che si intende rimuovere.

Se l'operazione va a buon fine non si torna nulla, altrimenti un messaggio di errore per comunicare che non c'è nessun profilo associato all'id specificato oppure che l'annuncio che si intende rimuovere non esiste nel database oppure che l'annuncio è già non presente nella wishlist.

Vengono utilizzati i metodi *findByld()* sulle collezioni *Annuncio* e *Profilo* per verificare l'esistenza dell'annuncio e del profilo e il metodo *updateOne()* per modificare la wishlist dell'utente richiedente. Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato.

Aggiornamento descrizione profilo

L'API *aggiornaDescrizione* ha lo scopo di modificare la descrizione su un profilo di un utente. Riceve in input alla richiesta l'id del profilo da modificare e la nuova descrizione da sovrascrivere.

In caso di buona riuscita della operazione non ritorna nulla, altrimenti un messaggio d'errore comunica l'inesistenza del profilo specificato nel database oppure la mal formulazione della descrizione.

Si usa il metodo *findByld()* per ottenere il profilo da modificare e il metodo *updateOne()* per aggiornare la descrizione del dato profilo.

Quest'API può essere eseguita solo specificando un token d'accesso come parametro nell'header, in quanto rappresenta una richiesta eseguibile solo da un utente autenticato.

Aggiornamento del rating del profilo

L'API *aggiornaRating* ha lo scopo di aggiornare automaticamente i dati sulle recensioni e sul rating del profilo di un utente, il cui identificativo viene passato come unico parametro in input.

In output viene fornito un messaggio di errore se l'id specificato non è associato a nessun profilo.

Si usa il metodo *findByld()* per verificare l'esistenza di un profilo e il metodo *find()* sulla collezione *Recensione* (una volta con argomento *{utenteRecensito : req.body.id}* e una volta con argomento *{utenteRecensore : req.body.id}*) per ottenere l'elenco di tutte le recensioni che hanno a che fare con un particolare profilo e calcolare la media delle valutazioni, così da aggiornare i vari attributi mediante il metodo *updateOne()* sul profilo specificato.

Aggiornamento statistiche annunci online

L'API *aggiornaAnnunciOnline* ha lo scopo di aggiornare automaticamente i dati sugli annunci online di un particolare utente, il cui id è fornito come unico parametro in input alla richiesta.

In output viene fornito un messaggio di errore se l'id specificato non è associato a nessun profilo.

Viene usato il metodo *findById()* per verificare l'esistenza di un profilo associato all'id specificato e il metodo *find()* con primo argomento *{inserzionista : req.body.id}* per ottenere l'elenco di tutti gli annunci nel database associati al preciso profilo specificato.

Infine il metodo *updateOne()* modifica il valore degli attributi nel profilo relativi al conteggio degli annunci online.

Aggiornamento statistiche di vendita

L'API *aggiornaStatisticheVendita* ha lo scopo di aggiornare automaticamente i dati e le statistiche relative alle vendite completate dall'utente nel sistema. Viene fornito in input alla richiesta l'id del profilo da modificare e in output un messaggio d'errore se nessun profilo è associato all'id specificato.

Viene usato il metodo *findById()* per verificare l'esistenza di un profilo associato all'id specificato e il metodo *find()* con primo argomento *{venditore : req.body.id}* per ottenere tutte le transazioni in cui l'utente appare come venditore. Dopo aver fatto operazione di aggregazioni e calcolo su quest'elenco si usa il metodo *updateOne()* per modificare il valore degli attributi nel profilo specificato relative alle statistiche di vendita dell'utente.

Aggiornamento statistiche di acquisti

L'API *aggiornaStatisticheAcquisti* ha lo scopo di aggiornare automaticamente i dati e le statistiche relative agli acquisti effettuati dall'utente nel sistema. Viene fornito in input alla richiesta l'id del profilo da modificare e in output un messaggio d'errore se nessun profilo è associato all'id specificato.

Viene usato il metodo *findById()* per verificare l'esistenza di un profilo associato all'id specificato e il metodo *find()* con primo argomento *{acquirente : req.body.id}* sulla collezione *Transazione* in *MongoDB* per ottenere tutte le transazioni in cui l'utente appare come acquirente. Dopo aver fatto operazione di aggregazioni e calcolo su quest'elenco si usa il metodo *updateOne()* per modificare il valore degli attributi nel profilo specificato relative alle statistiche degli ordini dell'utente.

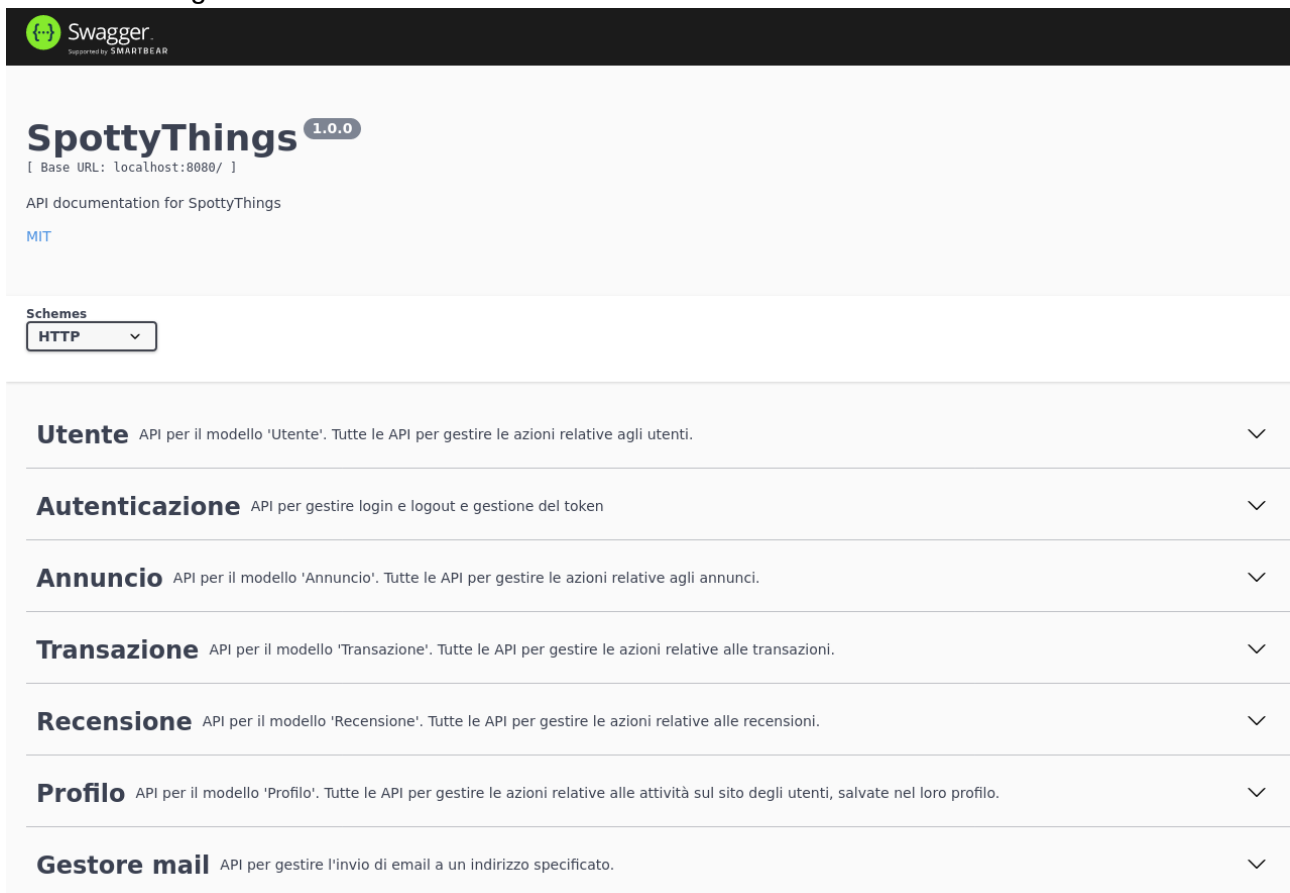
API documentation

Le API locali fornite dall'applicazione *SpottyThings* e descritte nella sezione precedente sono state documentate usando il modulo di NodeJS *Swagger UI Express*. In questo modo la documentazione relativa alle API è direttamente disponibile a chiunque veda il codice sorgente e le varie API sono facilmente testabili con *Swagger*.

L'endpoint da invocare per raggiungere la documentazione è:

<http://localhost:8080/api-docs>

Inizialmente le API vengono mostrate per modello per cui sono definite, quindi la rappresentazione iniziale è la seguente:



Swagger
Supported by SMARTBEAR

SpottyThings 1.0.0

[Base URL: localhost:8080/]

API documentation for SpottyThings

[MIT](#)

Schemes

HTTP

- Utente** API per il modello 'Utente'. Tutte le API per gestire le azioni relative agli utenti.
- Autenticazione** API per gestire login e logout e gestione del token
- Annuncio** API per il modello 'Annuncio'. Tutte le API per gestire le azioni relative agli annunci.
- Transazione** API per il modello 'Transazione'. Tutte le API per gestire le azioni relative alle transazioni.
- Recensione** API per il modello 'Recensione'. Tutte le API per gestire le azioni relative alle recensioni.
- Profilo** API per il modello 'Profilo'. Tutte le API per gestire le azioni relative alle attività sul sito degli utenti, salvate nel loro profilo.
- Gestore mail** API per gestire l'invio di email a un indirizzo specificato.

Premendo su uno sette tag vengono mostrate tutte le API per gestire quel particolare modello.

Le API possono essere di quattro tipi: POST, GET, PATCH oppure DELETE.

La POST è utilizzata quando è necessario inviare dei dati al sistema (ad esempio per creare una nuova risorsa), la GET quando è necessario ottenere dei dati dal sistema, la PATCH per modificare una precisa risorsa nel database e la DELETE per eliminare una risorsa.

Nella pagina seguente forniamo un esempio di rappresentazione con le API per il modello *Utente*.

Utente API per il modello 'Utente'. Tutte le API per gestire le azioni relative agli utenti.			^
POST	/api/u/signUp	Usata per salvare un nuovo utente nel sistema.	v
DELETE	/api/u/deleteu/{username}	Usata per eliminare un utente identificato da un preciso username.	v
GET	/api/u/getAll	Usata per ottenere tutti gli utenti registrati al sistema.	v
GET	/api/u/getu/{username}	Usata per ottenere una precisa risorsa utente associata al preciso username passato.	v
PATCH	/api/u/updateu/	Usata per aggiornare le informazioni di un utente, identificato da uno username che viene passato nel body.	v
PATCH	/api/u/updatee/	Usata per aggiornare l'indirizzo email di un utente, identificato da uno username che viene passato nel body.	v
PATCH	/api/u/updatep/	Usata per aggiornare la password di un utente, identificato da uno username che viene passato nel body.	v

Tutti i modelli sono organizzati in questo modo con le API ben descritte con il loro endpoint e la loro precisa funzionalità.

Tutte le API presentano una breve descrizione della loro funzionalità e successivamente forniscono il tipo di dati che si aspettano in input, associando anche un esempio valido di possibili valori in input.

Nella parte finale sono elencati tutti i possibili codici di ritorno che l'API può fornire con descrizione di cosa rappresentano.

Il seguente esempio mostra l'API *salvaUtente* del modello *Utente*.

Annuncio API per il modello 'Annuncio'. Tutte le API per gestire le azioni relative agli annunci.

POST /api/a/savea Usata per salvare un nuovo annuncio nel sistema.

Parameters Try it out

Name	Description
x-access-token <small>(header)</small>	token d'accesso <input type="text" value="x-access-token"/>
Dati annuncio object <small>(body)</small>	Le informazioni sul nuovo annuncio da creare <div>Example Value Model</div> <pre>{ "titolo": "Libro 'Promessi Sposi' edizione integrale", "insorcionista": "marioross17", "descrizione": "Libro in ottime condizioni de i 'Promessi Sposi'. Qualche piega in alcune pagine, ma niente di compromettente. Disponibilità della spedizione.", "via": "via Rosmini 15", "citta": "Rovereto", "provincia": "Trento", "modalitaTrasazione": "Vendita", "categoria": "hobby", "pagamentoOnline": true, "contoPayPal": "contomario@gmail.com", "sponsorizzato": false, "prezzo": 25.5, "prezzoAffittoAlGiorno": 5.5, "prezzoAffittoAllora": 7.5, "prezzoAffittoSettimanale": 11.5}</pre> <div>Parameter content type <input type="text" value="application/json"/></div>

Responses Response content type

Code	Description
201	CREATED. Annuncio creato correttamente.
400	BAD REQUEST. Errore nella formulazione della richiesta in seguito a incongruenze tra i dati (ad esempio pagamento online abilitato ma conto no).
401	UNAUTHORIZED. Token non fornito, impossibile autorizzare la richiesta.
403	FORBIDDEN. Token fornito non valido, impossibile autorizzare la richiesta. Si prega di riefettuare il login
404	NOT FOUND. Insercionista specificato non presente nel database.
409	CONFLICT. Errore in seguito al tentativo di creare un annuncio con un titolo già presente.
500	SERVER ERROR. Di varia natura.

