Università degli Studi di Milano Bicocca

Scuola di Scienze

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di laurea in Informatica

URBAN STORIES SHARING: IL BACK-END

Relatore: *Prof. Micucci Daniela*

Co-relatore: *Dott. Ginelli Davide*

Relazione della prova finale di:

Francesco Michele Ribaudo

Matricola 807847

Anno Accademico 2017-2018

Indice

Introduzione

In un mondo sempre più urbanizzato, le città sono un punto di incontro per la società contemporanea. Con il termine walkability si definisce quanto è "amichevole" camminare in una certa area. La walkability porta benifici in molti settori come quello sanitario, urbano ed economico. I fattori che influenzano questa unità di misura includono la presenza o meno di marciapiedi, la qualità degli stessi, nonché le condizioni del traffico e delle strade, modelli di utilizzo del suolo, accessibilità agli edifici e sicurezza. Inoltre, il concetto di walkability, è molto importante per ottenere un design urbano sostenibile.

Lo scopo del progetto **Urban Stories Sharing** è la realizzazione di un sistema che permette all'utente di narrare le proprie esperienze urbane descrivendo storie attraverso note, immagini, audio e video. Le annotazioni geolocalizzate verranno salvate su un repostory centralizzato così da poter essere condivise fra utenti. Questo progetto si focalizza, invece, sulla realizzazione di un Web service per la raccolta e la distribuzione delle storie urbane, supporterà quindi due delle operazioni **CRUD** base, quali GET e POST.

Con il termine back-end, ci si riferisce ad applicativi software con i quali gli utenti interagiscono indirettamente, solitamente attraverso l'utilizzo di applicazioni front-end.
Il progetto Urban Story Sharing nasce da un'idea del Dipartimento di Informatica, Sistemistica e Comunicazione dell'Università degli studi di Milano-Bicocca, che ha come obiettivo quello di migliorare la walkability delle aree urbane, attraverso un'applicazione mobile che permette ai cittadini di raccogliere dati relativi ai centri urbani da loro frequentati. Grazie alla raccolta di questi dati, si suppone, sia possibile migliorare l'urbanizzazione e quindi lo sviluppo delle città del futuro.// In questa tesi l'obiettivo è quello di sviluppare un web service per poter, appunto, raccogliere i dati da diversi tipi di dispositivi mobili servendosi del framework Laravel 5.

Nei prossimi capitoli verrà presentato il progetto in ogni sua parte, dalla fase di progettazione a quella di implementazione. In particolare, nel capitolo 1 si analizzeranno le tecnologie utilizzate per la realizzazione del servizio web. Nel capitolo 2 verrà illustrata la fase di progettazione. Nel capitolo 3 verranno esposte

le scelte implementative del progetto. Infine, nel capitolo 4 saranno presentati i risultati ottenuti e le conclusioni.

Capitolo 1

Tecnologie utilizzate

Questo capitolo si concentrerà solo sulle funzionalità offerte dal framework utilizzate durante la realizzazione dei servizi web per la raccoltà dei dati.

1.1 Laravel

Laravel è un framework per lo sviluppo di applicazioni web che cerca di facilitare il processo di sviluppo, semplificando le attività ripetitive utilizzate nella maggior parte delle applicazioni Web di oggi. Dal momento che riesce a fare tutti i compiti essenziali che vanno dalla gestione del web e gestione del database alla generazione di codice HTML, Laravel è chiamato **full stack framework**. Questo ambiente di sviluppo web integrato è pensato per offrire un miglioramento nel flusso di lavoro dello sviluppatore.

A differenza di altri ambienti di sviluppo, Laravel necessita solo di qualche modifica al codice di configurazione PHP ed è pronto per l'uso. Inoltre, l'utilizzo di pochi file di configurazione consente alle applicazioni web Laravel di avere un struttura del codice simile che le rende molto caratteristiche e facilmente identificabili. D'altro canto questo potrebbe essere visto come un vincolo su come lo sviluppatore intende organizzare la propria appliczione. Tuttavia, questi vincoli rendono molto più facile e veloce la creazione di applicazioni Web. Inoltre offre una macchina virtuale, **Laravel Homestead**, Vagrant pre-confezionata che fornisce un ambiente di sviluppo senza che sia necessario installare PHP, un server Web e qualsiasi altro software server sul computer locale. Le virtual machine di Vagrant sono completamente usa e getta quindi se qualcosa va storto, la si puo distruggere e ricreare in pochi minuti.

In questo progetto le funzionalità utilizzate di Laravel sono l'Eloquent ORM, i model, le route, i controller e i database.

1.1.1 ORM: Object-Relational Mapping

L'ORM è una tecnica di programmazione che aiuta a convertire i dati tra sistemi incompatibili. A questo scopo, Laravel, fornisce **Eloquent ORM** che consente di lavorare con gli oggetti e le tabelle del database utilizzando una sintassi semplice ed intuitiva. Ogni tabella del database ha un **modello** corrispondente utilizzato per l'interazione con quella tabella.

1.1.2 Model

Un Model è lo strumento con cui lo sviluppatore può manipolare i dati. Può essere considerato uno strato collocato tra i dati e l'applicazione.

1.1.3 Route

Le route permettono di definire gli instradamenti delle richieste HTTP. La loro funzione base è quella di definire come va gestita una certa richiesta HTTP. Avendo utilizzato in questo progetto delle classi Controller, le route si limiteranno a instradare le varie richieste verso l'opportuno controller.

1.1.4 Controller

Come anticipato nella sezione 1.1, Laravel semplifica molte operazioni. I controller fanno parte di queste semplificazioni permettendo di raggruppare ed organizzare tutta la logica di gestione delle richieste HTTP. I controller possono raggruppare la logica di gestione delle richieste correlate in una singola classe.

1.1.5 Database

L'interazione con i database offerta dal framework può essere strutturata in 3 parti:

- **Eloquent ORM**

  L'ORM Eloquent fornito con Laravel include una semplice implementazione PHP ActiveRecord che consente allo sviluppatore di eseguire query di database con una sintassi PHP invece di scrivere codice SQL, i metodi vengono semplicemente concatenati. Ogni tabella nel database possiede un Modello corrispondente attraverso il quale lo sviluppatore interagisce con detta tabella.

- **Schema builder**

  La classe Laravel Schema fornisce un database in grado di funzionare con una moltitudine di DBMS per gestire tutto il lavoro relativo al database come

la creazione o l'eliminazione di tabelle o l'aggiunta di campi a una tabella esistente. Funziona con una moltitudine di sistemi di database supportati da Laravel e MySQL.

- **Migrations**

  Le migrazioni possono essere considerate come un controllo di versione per il nostro database, infatti, consentono di modificare lo schema del database, descrivere e registrare tutte quelle modifiche specifiche in un file di migrazione. Ogni migrazione viene solitamente associata a un generatore di schemi per gestire il tutto senza sforzo. Una migrazione può anche essere ripristinata o riportata ad una versione precedente.

- **Seeders**

  La classe Seeder consente di inserire i dati nelle nostre tabelle. Questa funzione è molto utile poiché lo sviluppatore può inserire dati fittizzi nelle tabelle del database ogni volta che desidera testare l'applicazione web.

1.2 Composer

Un'altra caratteristica che distingue Laravel dagli altri framework è che è un framework Composer ready. In effetti, Laravel è esso stesso una miscela di diversi componenti Composer, ciò aggiunge un'interoperabilità al framework. Composer è uno strumento di gestione delle dipendenze per PHP. Essenzialmente, il ruolo principale di Composer nel framework di Laravel è quello di gestire la dipendenza delle dipendenze del nostro progetto. Ad esempio, se una delle librerie che stiamo utilizzando nel nostro progetto dipende da altre tre librerie che devono essere aggiornate, non è necessario trovare e aggiornare manualmente alcun file. È possibile aggiornare tutte e quattro le librerie tramite un singolo comando. Un altro vantaggio nell'utilizzo di Composer è che genera e gestisce un file di caricamento automatico, **autoload**, nella radice della directory **vendor/**, che conterrà tutte le dipendenze del progetto. In tal modo, dal lato dello sviluppatore non è necessario ricordare tutti i percorsi delle dipendenze e includere ciascuno di essi in ogni file del progetto, deve solo includere il file **autoload** fornito da Composer.

1.3 Artisan

Uno sviluppatore dovrebbe solitamente interagire con il framework di Laravel usando un'utilità a riga di comando che crea e gestisce l'ambiente del progetto. Laravel ha uno strumento da riga di comando incorporato chiamato **Artisan**. Questo strumento ci consente di eseguire la maggior parte delle operazioni di programmazione

ripetitive che la maggior parte degli sviluppatori evita di eseguire manualmente. Artisan può essere utilizzato per creare un codice scheletro, lo schema del database e le migrazioni associate che possono essere molto utili per gestire il sistema o ripararlo in caso di errori. Si possono creare anche i seeds di database che ci consentiranno di riempire con alcuni dati le tabelle. Può anche essere impiegato per generare subito i file di base dei modelli e dei controller tramite la riga di comando e gestire tali risorse e le rispettive configurazioni. Artisan ci permette persino di creare dei comandi personalizzati, per poter eseguire qualsiasi operazione possa essere necessaria al nostro progetto.

Capitolo 2

Progettazione

In questo capitolo verrà analizzata la parte progettuale del lavoro svolto focalizzandosi sui due pilastri principali, quali lo schema relazionale nella sezione 2.1 e le API nella sezione 2.2.

2.1 Schema Relazionale

2.1.1 Entità

Nella progettazione del repository centralizzato, per prima cosa, sono state definite le entità. Un'entità rappresenta una classe di oggetti che hanno proprietà comuni ed esistenza autonoma. Un'occorrenza di queste è una istanza della classe rappresentata dall'entità, dunque l'oggetto stesso. In uno schema, ogni entità ha un nome che la identifica univocamente, e viene rappresentata graficamente tramite un rettangolo con il nome dell'entità al suo interno. Dai requisiti iniziali bisogna quindi specificare le entità utili allo sviluppo del nostro progetto e i loro relativi attributi:

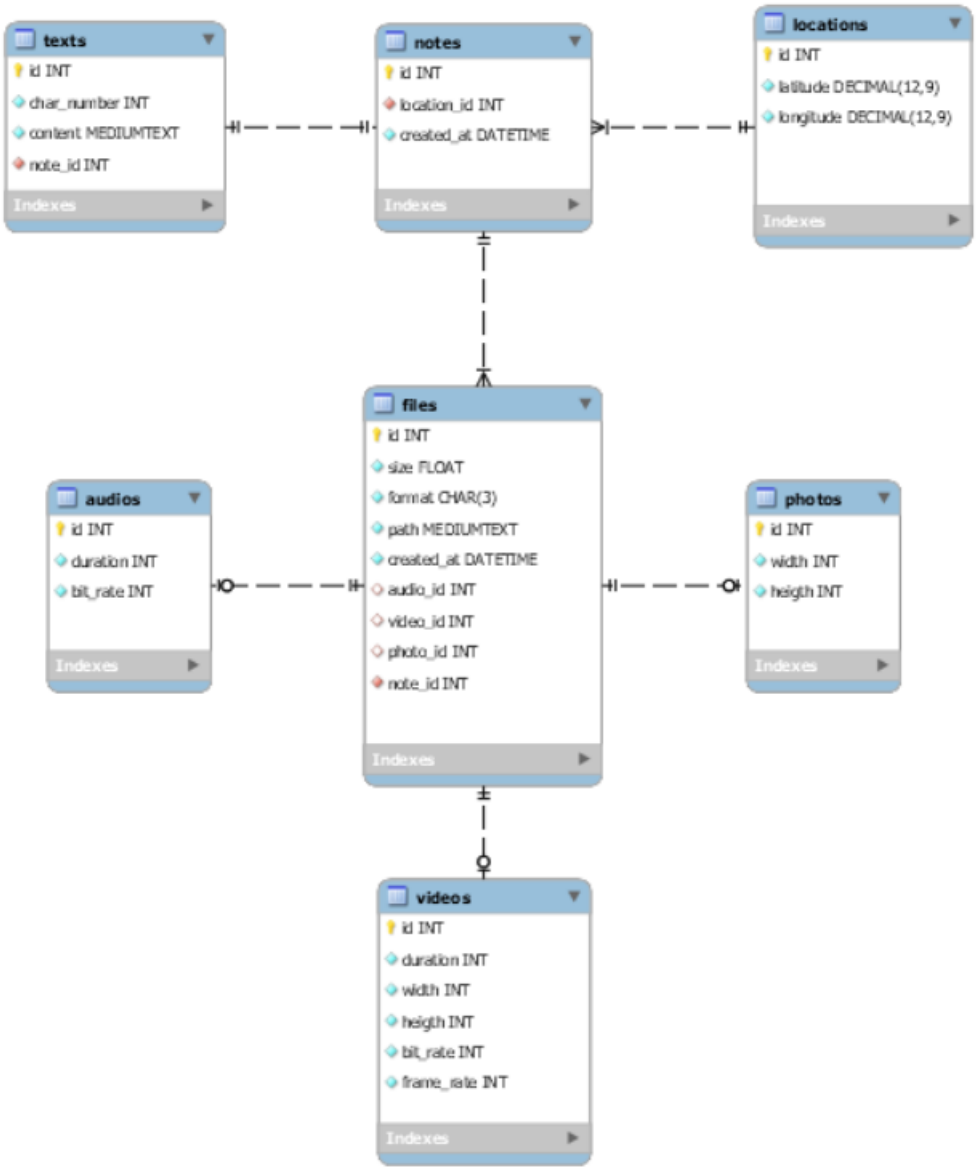

Figura 2.1: schema relazionale

- **Photos**

  L'entità Photos rappresenta, appunto, le foto che vengono memorizzate e richieste nel database. Gli attributi associati sono:

  - **id** identificatore numerico univoco.
  - **width** rappresenta la larghezza delle foto.

  - **height** rappresenta l'altezza delle foto.

- **Audios**

  La tabella Audios rappresenta tutti i file di tipo audio.

  Attributi:

  - **id** identificatore numerico univoco.
  - **duration** rappresenta la durata in secondi.
  - **bit_rate** rappresenta la qualità audio del file caricato.

- **Videos**

  L'entità Videos rappresenta i file video della base di dati. Attributi:

  - **id** identificatore numerico univoco.
  - **duration** rappresenta la durata in secondi.
  - **width** rappresenta la larghezza delle foto.
  - **height** rappresenta l'altezza delle foto.
  - **bit_rate** rappresenta la qualità audio del file caricato.
  - **frame_rate** rappresenta la qualità video del file.

- **Files**

  L'entità Files rappresenta una generalizzazione delle precedenti 3 entità. Gli attributi associati a questa tabella sono quindi tutti quei campi che hanno in comune i file multimediali(foto, audio e video):

  - **id** ovvero un identificatore numerico univoco.
  - **size** il quale indica la dimensione del file in byte.
  - **format** questo campo indica il formato del file che si vuole caricare o recuperare dal database.
  - **path** identifica l'indirizzo in cui viene memorizzato, localmente sul database, il file in questione.
  - **created_at** questo attributo definisce il giorno e l'ora di creazione del file.
  - **note_id** questa è la **chiave esterna** che associa l'entità **Files** e l'entità **Note**

Infine ci sono gli ultimi tre attributi, chiavi esterne, **photo_id, audio_id e video_id** , che contengono l'informazione sul tipo di file, ovvero se sono foto, audio o video, mettendo quindi in relazione la tabella files con la corrispondente tabella. Ovviamente queste tre chiavi posso assumere il valore **null** poichè, ad esempio, un file non può essere contemporaneamente una foto e un video, o un audio e una foto.

- **Notes**

  La tabella Notes rappresenta le storie condivise dagli utenti con il sistema. Una nota può contenere più tipi di file.

  Attributi:

  - **id** identificatore numerico univoco.
  - **location_id** chiave esterna che mettere in relazione l'entità Notes con l'entità Locations.

- **Texts**

  Questa entità rappresenta le note di tipo testo condivise dagli utenti.

  Attributi:

  - **id** identificatore numerico univoco.
  - **char_number** numero di caratteri della nota testuale.
  - **content** contenuto della nota testo.
  - **note_id** chiave esterna che mette in relazione questa tabella con la tabella Notes.

- **Locations**

  Infine, la tabella Locations rappresenta la geolocalizzazione delle note, infatti, questa, è associata alla tabella Notes.

  Attributi:

  - **id** identificatore numerico univoco.
  - **latitude** rappresenta la latitudine.
  - **longitude** rappresenta la longitudine.

2.1.2 Associazioni

Le associazioni rappresentano un legame concettuali tra una, due o più entità. In particolare, le associazioni, risultano necessarie per la corretta costruzione e implementazione di un database. In particolare esistono tre tipi di associazioni, **uno a uno, uno a molti e molti a molti**.

Di seguito verranno illustrate le associazioni che compongono lo schema relazionale di questo progetto.

- **Files -> Photos, Audios, Videos**

  La relazione che lega l'entità Files alle entità Photos, Audios e Videos è una associazione di tipo **uno a uno**. Questo perchè concettualmente un file può essere di un solo tipo per volta. Una foto non può essere al contempo un audio o un video.

- **Notes -> Files**

  La relazione che lega queste due tabelle è di tipo **uno a molti** poichè una nota può contenere più files di diverso tipo.

- **Notes -> Texts**

  Una nota può avere solamente una nota testuale, quindi in questo caso si tratta di associazione **uno a uno**.

- **Locations -> Notes**

  In questo caso, può succedere che in uno stesso luogo, utenti diversi, condividano delle note, di conseguenza l'associazione che lega queste tabelle è di tipo **uno a molti**.

2.2 API

Dopo aver definito la struttura del database, sono state definite le API (**Application Programming Interface**). Un'API consente a due sistemi di comunicare tra loro. Le API, sostanzialmente, fungono da traduttore per due o più sistemi che interagiscono. Solitamente, ogni API ha documentazione e specifiche che determinano il modo in cui le informazioni possono essere trasferite. Le API possono utilizzare le richieste HTTP per ottenere informazioni da un'applicazione Web o un server web. Le API sono generalmente classificate come SOAP o REST ed entrambe sono utilizzate per accedere ai servizi Web. In questo progetto sono state definite delle API di tipo REST, che utilizzano gli URL per ricevere o inviare informazioni. REST utilizza quattro diversi verbi HTTP (GET, POST, PUT e DELETE) per eseguire attività.

2.2.1 Documentazione API

METODO	ENDPOINT	FUNZIONALITA'
Get	/notes	Lista di oggetti Note
Get	/notes/{id}	Recuperare una nota specifica
Get	/files/{id}	Recuperare un file specifico
Get	/texts/{id}	Recuperare una specifica nota testuale
Post	/texts	Salvare una nuova nota testuale
Get	/photos/{id}	Recuperare una specifica foto
Post	/photos	Salvare una nuova foto
Get	/audios/{id}	Recuperare uno specifico file audio
Post	/audios	Salvare un nuovo file audio
Get	/videos/{id}	Recuperare un nuovo file video
Post	/videos	Salvare un nuovo video

Tabella 2.1: Rappresentazione tabellare degli **Endpoints** definiti

2.2.2 Risorse

NOTE		
PARAMETRO	**TIPO**	**DESCRIZIONE**
id	int	Identificatore della nota
location	int	Chiave Esterna

Tabella 2.2: Descrizione risorsa **Note**

LOCATION		
PARAMETRO	**TIPO**	**DESCRIZIONE**
id	int	Indentificatore della geolocalizzazione
latitude	decimal	Rappresentazione della latitudine
longitude	decimal	Rappresentazione della longitudine

Tabella 2.3: Descrizione risorsa **Location**

FILE		
PARAMETRO	**TIPO**	**DESCRIZIONE**
id	int	Identificatore del file
size	float	Dimensione
format	char	Formato
path	mediumtext	Percorso di salvataggio
created_at	datetime	Data e ora di creazione
note_id	int	Relazione con tabella Notes
photo_id	int	Relazione con tabella Photos
audio_id	int	Relazione con tabella Audios
video_id	int	Relazione con tabella Videos

Tabella 2.4: Descrizione risorsa **File**

TEXT		
PARAMETRO	**TIPO**	**DESCRIZIONE**
id	int	Identificatore della nota testo
content	mediumtext	Contenuto del testo
char_number	int	Numero di caratteri
note_id	int	Relazione con tabella Notes

Tabella 2.5: Descrizione risorsa **Text**

PHOTO		
PARAMETRO	**TIPO**	**DESCRIZIONE**
id	int	Identificatore della foto
width	int	Larghezza della foto
height	int	Altezza della foto

Tabella 2.6: Descrizione risorsa **Photo**

AUDIO		
PARAMETRO	**TIPO**	**DESCRIZIONE**
id	int	Identificatore del file audio
duration	int	Durata dell'audio
bit_rate	int	Qualità dell'audio

Tabella 2.7: Descrizione risorsa **Audio**

VIDEO		
PARAMETRO	**TIPO**	**DESCRIZIONE**
id	int	Identificatore del file video
duration	int	Durata del video
width	int	Larghezza del video
height	int	Altezza del video
bit_rate	int	Qualità dell'audio
frame_rate	int	Qualità del video

Tabella 2.8: Descrizione risorsa **Video**

2.2.3 Endpoints

2.2.4 GET /notes

Questo endpoint permette di accedere ai dati del database restituendo tutte le note caricate dagli utenti. Questo endpoint è accessibile solo con una richiesta GET; inoltre, prevede la possibilità di passare dei parametri in query string.

PARAMETRO	RICHIESTO	VALORE	DESCRIZIONE
lat	opzionale	numerico	Latitudine associata alla posizione dell'utente a partire dalla quale si vogliono cercare le note.
long	opzionale	numerico	Longitudine associata alla posizione dell'utente a partire dalla quale si vogliono cercare le note.
max_distance	opzionale	numerico	Distanza massima entro il quale si vogliono cercare le note rispetto alla posizione dell'utente.
type	opzionale	text, photo, audio, video	Specifica il tipo di note da ricercare.

Tabella 2.9: Descrizione dei **parametri** in input

PARAMETRO	VALORE
notes	array di oggetti Note

Tabella 2.10: Struttura del **JSON** restituito

```
/notes
```

Listing 2.1: Esempio di chiamata

```
[
        {
                "id": 1,
                "location_id": 1,
                "created_at": "2018-10-04␣07:18:01"
        },
        {
                "id": 2,
                "location_id": 2,
                "created_at": "2018-10-04␣07:20:10"
        },
]
```

Listing 2.2: Esempio **JSON** restituito

```
/notes?lat=12.45&long=13.45&max_distance=1
```

Listing 2.3: Esempio di chiamata con **query string**

```
{
        "id": 4,
        "location_id": 4,
        "created_at": "2018-10-04␣07:21:50",
        "location": {
                "id": 4,
                "latitude": "12.45",
                "longitude": "13.45"
        }
},
{
        "id": 5,
        "location_id": 5,
        "created_at": "2018-10-04␣07:23:00",
        "location": {
                "id": 5,
                "latitude": "12.45",
                "longitude": "13.45"
        }
}
```

Listing 2.4: Esempio **JSON** restituito

2.2.5 GET /notes/{id}

Questo endpoint permette di accedere ai dati del database restituendo la nota specificata tramite parametro. Questo endpoint è accessibile solo con una richiesta GET.

PARAMETRO	RICHIESTO	VALORE	DESCRIZIONE
id	obbligatorio	intero	Identifica la nota a cui si vuole accedere

Tabella 2.11: Descrizione dei **parametri** in input

PARAMETRO	VALORE
note	oggetto di tipo Note

Tabella 2.12: Struttura del **JSON** restituito

```
/notes/1
```

Listing 2.5: Esempio di chiamata

```
{
        "id": 1,
        "location_id": 1,
        "created_at": "2018-10-04 07:18:01"
}
```

Listing 2.6: Esempio **JSON** restituito

2.2.6 GET /files/{id}

Questo endpoint permette di accedere ad uno specifico file passando come parametro l'id. Questo endpoint è accessibile solo con una richiesta **GET**.

PARAMETRO	RICHIESTO	VALORE	DESCRIZIONE
id	obbligatorio	intero	identifica il file a cui si vuole accedere

Tabella 2.13: Descrizione dei **parametri** in input

PARAMETRO	VALORE
file	oggetto di tipo File

Tabella 2.14: Struttura del **JSON** restituito

```
/files/1
```

Listing 2.7: Esempio di chiamata

```
{
        "id": 1,
        "size": 12345,
        "format": "mp3",
        "path": "audios/audio-1538637710.mp3",
        "note_id": 4,
        "photo_id": null,
        "audio_id": 1,
        "video_id": null,
        "created_at": "2018-10-04 07:21:50"
}
```

Listing 2.8: Esempio **JSON** restituito

2.2.7 POST /texts

Questo endpoint permette di salvare note testuali all'interno del database. Non necessita di parametri ed è accessibile con una richiesta HTTP di tipo **POST**.

PARAMETRO	VALORE
content	testo
latitude	numerico
longitude	numerico

Tabella 2.15: Struttura del **JSON** da inviare con una **POST**

```
/texts
```

Listing 2.9: Esempio di **POST**

```
{
        "content": "lorem ipsum, quia dolor sit",
        "latitude": 14.6543,
        "longitude": 21.65432
}
```

Listing 2.10: Esempio **JSON** da inviare

2.2.8 GET /texts/{id}

Questo endpoint permette di accedere ad una specifica nota testuale. E' accessibile solo con una richiesta **GET**.

PARAMETRO	RICHIESTO	VALORE	DESCRIZIONE
id	obbligatorio	intero	identifica la nota testuale a cui si vuole accedere

Tabella 2.16: Descrizione dei **parametri** in input

PARAMETRO	VALORE
text	oggetto di tipo Text

Tabella 2.17: Struttura del **JSON** restituito

```
/texts/4
```

Listing 2.11: Esempio di chiamata

```
{
        "id": 4,
        "content": "Testo Di prova TEST RESPONSE",
        "char_number": 28,
        "note_id": 7
}
```

Listing 2.12: Esempio **JSON** restituito

2.2.9 POST /photos

Questo endpoint permette di salvare delle foto all'interno del database. Non necessita di parametri ed è accessibile con una richiesta HTTP di tipo **POST**.

PARAMETRO	VALORE
size	numerico
format	alfanumerico
width	numerico
height	numerico
image_data	alfanumerico
latitude	numerico
longitude	numerico

Tabella 2.18: Struttura del **JSON** da inviare con una **POST**

```
/photos
```

Listing 2.13: Esempio di richiesta **POST**

```
{
        "size" : 4321,
        "format" : "png",
        "width" : 1024,
        "heigth" : 817,
        "image-data" : "iVBORw0KGgoAABBJRU5ErkJggg==",
        "latitude" : 45.5225169,
        "longitude" : 9.2142299

}
```

Listing 2.14: Esempio **JSON** da inviare

2.2.10 GET /photos/{id}

Questo endpoint permette di accedere ad una specifica foto presente nel database. E' accessibile solo con una richiesta **GET**.

PARAMETRO	RICHIESTO	VALORE	DESCRIZIONE
id	obbligatorio	intero	identifica la foto a cui si vuole accedere

Tabella 2.19: Descrizione dei **parametri** in input

PARAMETRO	VALORE
photo	oggetto photo

Tabella 2.20: Struttura del **JSON** restituito

```
/photos/4
```

Listing 2.15: Esempio di chiamata

```
{
        "id": 4,
        "width": 1920,
        "height": 1080,
        "image_data": "R0lGODlhPQBEAPAwO/AwH+0pCZbEhAAOw==",
        "file": {
                "id": 4,
                "size": 450,
                "format": "gif",
                "path": "photos/image-1538637823.gif",
                "note_id": 6,
                "photo_id": 1,
                "audio_id": null,
                "video_id": null,
                "created_at": "2018-10-04␣07:23:43"
        }
}
```

Listing 2.16: Esempio **JSON** restituito

2.2.11 POST /audios

Questo endpoint permette di salvare files audio all'interno del database. Non necessita di parametri ed è accessibile con una richiesta HTTP di tipo **POST**.

PARAMETRO	VALORE
size	numerico
format	alfanumerico
duration	numerico
bit_rate	numerico
audio_data	alfanumerico
latitude	numerico
longitude	numerico

Tabella 2.21: Struttura del **JSON** da inviare con una **POST**

```
/audios
```

Listing 2.17: Esempio di richiesta **POST**

```
{
        "size" : 4321,
        "format" : "mp3",
        "duration" : 120,
        "bit_rate" : 12345,
        "audio-data" : "SUQzAwAAA/zw3cGIAAgACAAIAAgACAAAA==",
        "latitude" : 45.5225169,
        "longitude" : 9.2142299
}
```

Listing 2.18: Esempio **JSON** da inviare

2.2.12 GET /audios/{id}

Questo endpoint permette di accedere ad uno specifico file audio presente nel database ed è accessibile solo con una richiesta **GET**.

PARAMETRO	RICHIESTO	VALORE	DESCRIZIONE
id	obbligatorio	intero	identifica il file audio a cui si vuole accedere

Tabella 2.22: Descrizione dei **parametri** in input

PARAMETRO	VALORE
audio	oggetto audio

Tabella 2.23: Struttura del **JSON** restituito

```
/audios/6
```

Listing 2.19: Esempio di richiesta **GET**

```
{
        "id": 6,
        "duration": 10,
        "bit_rate": 4321,
        "audio_data": "SUQzAwAAAs3/cGIAAgACAAIAAgACAAAA==",
        "file": {
                "id": 6,
                "size": 12345,
                "format": "mp3",
                "path": "audios/audio-1538637710.mp3",
                "note_id": 4,
                "photo_id": null,
                "audio_id": 1,
                "video_id": null,
                "created_at": "2018-10-04␣07:21:50"
        }
}
```

Listing 2.20: Esempio **JSON** restituito

2.2.13 POST /videos

Questo endpoint permette di salvare files video all'interno del database. Non necessita di parametri ed è accessibile con una richiesta HTTP di tipo **POST**.

PARAMETRO	VALORE
size	numerico
format	alfanumerico
duration	numerico
width	numerico
height	numerico
bit_rate	numerico
frame_rate	numerico
video_data	alfanumerico
latitude	numerico
longitude	numerico

Tabella 2.24: Struttura del **JSON** da inviare con una **POST**

```
/videos
```

Listing 2.21: Esempio di richiesta **POST**

```
{
        "size" : 4321,
        "format" : "mkw",
        "duration" : 120,
        "width" : 1080,
        "heigth" : 920,
        "bit_rate" : 12345,
        "frame_rate" : 54321,
        "video_data" : "iVBORw0KGgrkJggg==",
        "latitude" : 45.5225169,
        "longitude" : 9.2142299
}
```

Listing 2.22: Esempio **JSON** da inviare

2.2.14 GET /videos/{id}

Questo endpoint permette di accedere ad uno specifico file video presente nel database ed è accessibile solo con una richiesta **GET**.

PARAMETRO	RICHIESTO	VALORE	DESCRIZIONE
id	obbligatorio	intero	Identifica il file video a cui si vuole accedere.

Tabella 2.25: Descrizione dei **parametri** in input

PARAMETRO	VALORE
video	oggetto Video

Tabella 2.26: Struttura del **JSON** restituito

```
/videos/1
```

Listing 2.23: Esempio di richiesta **GET**

```
{
        "id": 1,
        "duration": 10,
        "width": 1920,
        "height": 1080,
        "bit_rate": 4321,
        "frame_rate": 1234
        "video_data": "ANdFI/UORIffOCHGhRMSxOBv+VSBIAICgAAAKw",
        "file": {
                "id": 2,
                "size": 12345,
                "format": "mp4",
                "path": "videos/video-1538637780.mp4",
                "note_id": 5,
                "photo_id": null,
                "audio_id": null,
                "video_id": 1,
                "created_at": "2018-10-04␣07:23:00"
        }
}
```

Listing 2.24: Esempio **JSON** restituito

Capitolo 3

Implementazione

In questo capitolo verrà presentato nel dettaglio l'implementazione del database e del web service.

3.1 Struttura dell'applicazione

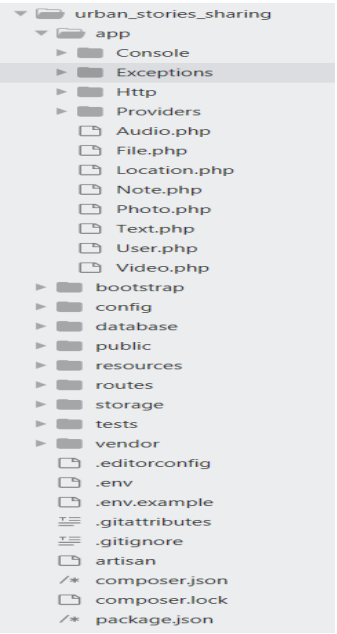

Figura 3.1: Struttura dell'applicazione

La struttura dell'applicazione lato back-end, come si può vedere in Figura 3.1, mantiene la struttura base di un'applicazione Laravel, strutturata in cartelle, ognuna

delle quali contiene dei files con uno specifico compito.

3.1.1 App directory

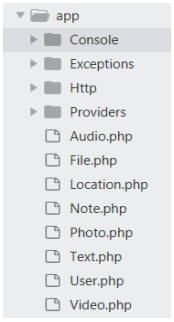

Figura 3.2: App directory

La cartella ***app*** contiene il codice principale dell'applicazione. Per impostazione predefinita del framework, questa directory è assegnata sotto il namespace di **App** e viene caricata automaticamente da Composer. Questa directory contiene tutti i modelli definiti per lo sviluppo del progetto oltre a varie sotto directory aggiuntive come Console e Http, che si possono considerare come un vero e proprio strato che fornisce un'API nel nucleo dell'applicazione. La directory Console contiene tutti i comandi Artisan personalizzati, mentre la directory Http contiene controller e middleware, ovvero tutta logica per gestire le richieste.

3.1.2 Database directory

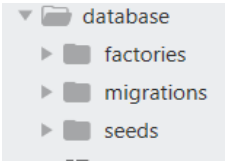

Figura 3.3: Database directory

La direcory ***Database*** contiene le migrazioni della base di dati e i file di ***seeds***, utili a popolare il modello.

3.1.3 Storage directory

La directory di archiviazione contiene l'archivio ***/app/directory*** pubblico, il quale, viene utilizzato per archiviare file generati dall'utente, ovvero le note contenenti i file multimediali, che devono essere accessibili al pubblico.

3.2 Implementazione database

Inizialmente, per quanto concerne la creazione del modello relazione, si era pensato di predisporre una entità autonoma per ogni tipo di file multimediale da caricare sul database, ma questo tipo di soluzione risultava ridondante poichè diverse entità presentavano gli stessi attributi, per questo motivo si è deciso di cambiare approccio e utilizzare un'entità genitore **files**, contenente le informazioni comuni, in relazione alle entità specializzanti per ogni tipo di file: **photos**, **audios** e **videos**.

```
public function up()
{
        Schema::create('files', function (Blueprint $table)
        {
                $table->increments('id');
                $table->float('size');
                $table->text('format');
                $table->mediumText('path')->nullable();
                $table->integer('note_id')->unsigned();
                $table->integer('photo_id')->unsigned()
                        ->unique()->nullable();
                $table->integer('audio_id')->unsigned()
                        ->unique()->nullable();
                $table->integer('video_id')->unsigned()
                        ->unique()->nullable();
                $table->foreign('note_id')->references('id')
                        ->on('notes');
                $table->foreign('photo_id')->references('id')
                        ->on('photos');
                $table->foreign('audio_id')->references('id')
                        ->on('audios');
                $table->foreign('video_id')->references('id')
                        ->on('videos');
                $table->timestamps();
        });
}
```

Listing 3.1: Esempio schema tabella files

```
public function up()
{
        Schema::create('videos', function (Blueprint $table)
        {
                $table->increments('id');
                $table->integer('duration');
                $table->integer('width');
                $table->integer('height');
                $table->integer('bit_rate');
                $table->integer('frame_rate');
                $table->timestamps();
        });
}
```

Listing 3.2: Esempio schema tabella videos

Con lo sviluppo del progetto si è deciso di aggiungere, al modello, una nuova entità ***notes*** messa in relazione con l'entità ***files***, ***texts*** e ***locations***, rappresentante la geolocalizzazione delle note, che inizialmente era in relazione con l'entità ***files***.

3.2.1 Codifica dei dati

La codifica dei dati è un punto critico del progetto, poichè una scelta implementativa non ottimale potrebbe appensantire molto la base di dati. Le due alternative possibili erano l'upload tramite ***multipart form-data*** oppure tramite la **codifica in base64**.

Dopo un'analisi delle ipotetiche dimensioni dei file salvate nel repository, si è scelto di riceve i dati codificati in **Base64**, poiché si presuppone che le note testuali, le immagini, i file audio e i file video caricati sul server siano di piccole dimensioni. Inoltre è stato scelto di non salvare la codifica dei dati nel database, per evitare, una volta raggiunti volumi importanti di dati, di appesantire la base di dati. Per permettere, nonostante ciò, al client, di ricevere il file richiesto con richieste HTTP di tipo ***GET***, il server si preoccupa di codificare i dati prima di essere inviati al client.

3.3 Implementazione web service

Questa sezione tratta dell'implementazione del web service sviluppato per lo scambio di informazioni con la parte client, in particolare si mostrerà come vengono salvati e recuperati i file.

3.3.1 Routes

```
Route::get('/notes', 'NoteController@index');
Route::get('/notes/{id}', 'NoteController@show');
Route::get('/texts/{id}', 'TextController@show');
Route::post('/texts', 'TextController@store');
Route::get('/photos/{id}', 'PhotoController@show');
Route::post('/photos', 'PhotoController@store');
Route::get('/audios/{id}', 'AudioController@show');
Route::post('/audios', 'AudioController@store');
Route::get('/files/{id}', 'FileController@show');
Route::get('/videos/{id}', 'VideoController@show');
Route::post('/videos', 'VideoController@store');
```

Listing 3.3: Definizione routes

Le routes definite vengono caricate dal **RouteServiceProvider** a cui è assegnato il gruppo middleware 'api' e si occupano di instradare le richieste sui vari endpoint verso il controller dedicato.

3.3.2 Controllers

Definite le routes, sono stati realizzati i vari controller. Per ogni tipologia di file caricabile è stato definito un controller che gestisce le varie operazioni di richiesta e salvataggio dati. Per ogni controller, quindi, sono stati definiti gli opportuni metodi che gestiscono le varie richieste HTTP; I nomi dei metodi definiti all'interno dei controller sono stati standardizzati in modo da permettere una più semplice comprensione.

Metodo Index

Il metodo ***index***, all'interno di un controller, risponde ad una richiesta di tipo ***GET*** e si occupa di recuperare i file richiesti dal database. Il metodo può ritornare una collezione di oggetti specifici, identificati dai parametri passati tramite query string oppure una collezione di tutte le note registrate nel database.

Metodo Store

l metodo ***store*** si occupa della gestione delle richieste HTTP di tipo ***POST***. Ovvero gestisce tutte le operazioni di salvataggio di ogni tipo di files.

Metodo Show

```
public function show($id)
{
        if(Note::find($id)){
                return Note::find($id);
        }else {
                return Response('Note not found', 404);
        }
}
```

Listing 3.4: Funzione Show del controller NoteController

Il un metodo ***show*** gestisce richieste HTTP di tipo ***GET*** e ritorna un oggetto singolo che corrisponde ad un file, identificato dal parametro ***id*** passato con la richiesta; il tipo dell'oggetto ritornato dipende dall'endpoint su cui si effettua la richiesta.

Query string

Quando viene effettuata una richiesta HTTP di tipo ***GET*** può essere necessario passare dei parametri in ***query string*** per specificare una certa tipologia di files da reperire. Per permettere questo tipo di operazioni è stata implementata la gestione, appunto, delle query string. La richiesta principale è quella di reperire delle note entro una certa distanza da una posizione data; La logica alla base di questa operazione è quella di ottenere tutti i parametri passati in query string, ovvero le coordinate geografiche dell'utente e la distanza massima entro la quale si vogliono recuperare i dati; dopodichè, vengono ciclate tutte le note all'interno del repository e, sfruttando il teorema di Pitagora, vengono selezionate solo le note richieste.

```
foreach ($notes as $key => $value) {

        if($value->location != null) {
                $lat2 = $value->location->latitude;
                $long2 = $value->location->longitude;
                $distance = sqrt(
                                pow($lat2-$lat1, 2) +
                                pow($long2-$long1, 2)
                                );

                if($distance > $max_distance) {
                        $notes->forget($key);
                }
        } else {
                $notes->forget($key);
        }
}
```

Listing 3.5: Gestione query string

Capitolo 4

Risultati ottenuti e conclusioni

In quest'ultimo capitolo vengono presentati i risultati ottenuti e i possibili miglioramenti per un possibile sviluppo futuro del progetto.

4.1 Risultati ottenuti

Il lavoro svolto durante questo progetto di stage ha permesso di ottenere un repository in grado di inviare e ricevere dati multimediali in formato JSON codificati in base64 da e verso un qualsiasi tipo di client, rispettando le direttive delle API sviluppate e descritte nel capitolo 3.

4.1.1 Ricezione dei dati

Dal punto di vista della ricezione dei dati, il sistema, è in grado di ricevere qualsiasi file multimediale sottoforma di oggetti JSON e salvarli logicamente nel database creato e fisicamente sull'ipotetico server che ospiterà il web service.

4.1.2 Invio dei dati

Per quanto riguarda l'invio dei dati, invece, tramite le query descritte nel Capitolo 3, il sistema può recuperare tutti i dati salvati, genericamente oppure in base al tipo di file. Inoltre, si possono reperire note in base alla loro geolocalizzazione o richiedere files entro una certa distanza dalla posizione dell'utente.

4.1.3 Sviluppi futuri

4.2 Conclusioni

Dall'analisi del risultato finale ottenuto viene compreso come il lavoro fin'ora prodotto sia funzionante ma migliorabile. Ad esempio, si potrebbe implementare la possibilità di gestire l'utenza, quindi aggiungere un servizio di autenticazione che permetta all'utente finale di avere uno storico dei file postati da esso o ancora di poter reperire informazioni riguardo le note postate dagli amici degli utenti. Questo prevede una riprogettazione del modello relazionale, poichè, allo stato attuale non è stato predisposto per questo tipo di funzioni.

Un altro aspetto migliorabile del web service è il lato della sicurezza dei dati, non gestito durante questo progetto. Molti altri aspetti sono migliorabili o modificabili in base al tipo di client frontend, ancora in fase di progettazione, che verrà sviluppato in futuro.