Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

Analisi, progettazione e sviluppo del backend di un'applicazione web per la gestione di eventi

Tesi di laurea

Relatore
Prof.Davide Bresolin

Laureando
Alberto Lazari

Anno Accademico 2021-2022

Sommario

La tesi descrive il lavoro svolto durante il periodo di stage, della durata di circa trecento ore, presso la sede di Treviso di Moku S.r.l., il cui obiettivo era la reimplementazione del backend di una piattaforma di gestione di eventi, sfruttando gli strumenti tipicamente utilizzati nei progetti dell'azienda.

In particolare, i seguenti capitoli tratteranno del contesto lavorativo dell'azienda, dell'analisi svolta sullo stato della piattaforma ad inizio stage, della progettazione e successiva implementazione iniziale del nuovo backend, focalizzando l'attenzione sulle scelte stilistiche e architetturali perseguite.

Ringraziamenti

Voglio ringraziare il Prof. Davide Bresolin, per l'interesse, il supporto e l'aiuto fornito durante il periodo di stage e di stesura di questa tesi.

Ringrazio i miei genitori e tutti i miei famigliari per il supporto e l'affetto che mi hanno donato durante questi hanno di studio.

Ringrazio i colleghi di Moku che mi hanno accolto calorosamente tra loro durante la mia esperienza di stage, in particolare Riccardo e Nicolò, per avermi sempre fornito l'aiuto che cercavo durante il mio lavoro.

Ringrazio la Comunità Capi del Mestre 2, per avermi accompagnato fin qui, infondendo in me una maturità e una consapevolezza che mi hanno permesso di raggiungere questo traguardo.

Ringrazio i miei colleghi studenti, stagisti e gli amici dei gruppi di progetto, con cui ho condiviso le difficoltà e le soddisfazioni di quest'anno.

Infine ringrazio i miei amici, che mi sono sempre stati vicini e con cui ho condiviso esperienze indimenticabili.

Padova, Luglio 2022 Alberto Lazari

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

L'azienda

1.1 Descrizione generale

Descrizione dell'azienda: brevissima storia, divisione dei ruoli, spazi e luoghi di lavoro.

1.2 Modello di sviluppo

Modello agile, Scrum, organizzazione dei team.

Capitolo 2

Descrizione dello stage

2.1 Introduzione al progetto

Storia del progetto prima del mio arrivo, azienda che ha commissionato il progetto, descrizione dello scopo della piattaforma e del suo funzionamento, motivazioni alla base della scelta di riscrittura del backend.

2.2 Requisiti

Requisiti obbligatori, desiderabili e opzionali previsti.

2.3 Pianificazione

Divisione settimanale del lavoro dal piano di lavoro, incluse correzioni.

2.4 Tecnologie utilizzate

Descrizione della configurazione del framework Ruby on Rails utilizzata: librerie utilizzate, Postgres, AWS, API REST .

Capitolo 3

Analisi e refactor dei modelli

3.1 Introduzione

Spiegazione del lavoro svolto in questa fase.

3.2 Modifiche effettuate

Decisioni significative prese durante l'attività di refactor dei modelli.

3.3 Diagramma ER completo

Diagramma ER del nuovo backend.

Capitolo 4

Progettazione della API

4.1 Introduzione

Spiegazione del lavoro svolto in questa fase.

4.2 Notazione adottata

Spiegazione convenzioni adottate nella descrizione degli endpoint.

4.3 Descrizione delle funzionalità esposte

Descrizione degli endpoint esposti dalla API, in generale per ogni modello e nello specifico per le eccezioni.

4.3.1 Lista delle risorse

Route index, attributi mostrati per ogni modello implementato.

4.3.2 Dettagli di una risorsa

Route show, attributi mostrati per ogni modello implementato.

4.3.3 Creazione di una risorsa

Route create.

4.3.4 Modifica di una risorsa

Route update.

4.3.5 Eliminazione di una risorsa

Route delete.

4.3.6 Lista dei ruoli degli utenti

4.3.7 Lista delle risorse interne a una specifica risorsa

4.3.8 Creazione di una risorsa all'interno di un'altra risorsa

4.4 Gestione dei permessi

Permessi per le categorie di utenti per ogni controller.

4.4.1 Permessi richiesti

4.4.2 Parametri permessi

Capitolo 5

Codifica

5.1 Modelli

La codifica dei modelli passa per tre fasi successive:

1. creazione della entità del modello nel database e della classe, utilizzando le migrazioni;
2. l'associazione del modello con altri modelli o elementi di *storage*;
3. le validazioni sugli attributi e sulle associazioni dichiarate.

5.1.1 Migrazioni del database

Basandosi su quanto definito nella fase di progettazione dei modelli, descritta nel capitolo 3, questi sono stati generati utilizzando da linea di comando il generatore automatico `rails generate model` o, in versione ridotta, `rails g model`. Il comando accetta come argomenti:

- il nome del modello, al singolare e in *CamelCase*;
- gli attributi che deve avere il modello;
- per ogni attributo: il suo tipo, che rispecchia, ad alto livello, i tipi comunemente disponibili per le colonne nei DBMS SQL. Normalmente è uno dei seguenti tipi nativi delle migrazioni di Rails[1], agnostici rispetto all'implementazione del database:
  - `primary_key`,
  - `string`,
  - `text`,
  - `integer`,
  - `bigint`,
  - `float`,
  - `decimal`,

[1] *Documentazione ufficiale di Ruby on Rails - metodo* ***`add_column`***. URL: `https://api.rubyonrails.org/classes/ActiveRecord/ConnectionAdapters/SchemaStatements.html#method-i-add_column`.

        - `datetime`,
        - `timestamp`,
        - `time`,
        - `date`,
        - `binary`,
        - `blob`,
        - `boolean`,
        - `references`

    - per ogni attributo: l'identificatore `uniq`, che imposta un indice su quella colonna del database, che ne specifica l'unicità nell'entità.

Di conseguenza, la sintassi generale è la seguente:

```
rails g model ModelName attr_1:type:[uniq] attr_2:type:[uniq] ...
```

Portando un esempio, per la generazione di una versione semplificata del modello degli organizzatori può essere usato il comando seguente:

```
rails g model Organizer name:string vat:string:uniq address:string
↪  platform:references creator_id:bigint
```

che ha prodotto la seguente migration:

```ruby
# db/migrate/{timestamp}_create_organizers.rb

class CreateOrganizers < ActiveRecord::Migration[7.0]
  def change
  create_table :organizers do |t|
    t.string :name
    t.string :vat
    t.string :address
    t.references :platform, foreign_key: true
    t.bigint :creator_id

    t.timestamps
  end
  add_index :organizers, :vat, unique: true
  end
end
```

Successivamente questa viene modificata, aggiungendo i vincoli `NOT NULL` e la chiave esterna verso l'utente creatore prima di eseguire effettivamente la migrazione.

Si noti come non sia necessario specificare la chiave primaria. Il comportamento di *default* di Active Record è introdurre automaticamente un identificativo progressivo, chiamato `id`, di tipo `bigint`. Inoltre `timestamps` genera automaticamente degli attributi gestiti dalla gemma, per tracciare l'istante di creazione e ultima modifica dei record.

La migrazione dopo la modifica è la seguente:

```ruby
# db/migrate/{timestamp}_create_organizers.rb

class CreateOrganizers < ActiveRecord::Migration[7.0]
  def change
    create_table :organizers do |t|
      t.string :name, null: false
      t.string :vat, null: false
      t.string :address, null: false
      t.references :platform, null: false, foreign_key: true
      t.bigint :creator_id, null: false

      t.timestamps
    end
    add_foreign_key :organizers, :users, column: :creator_id
    add_index :organizers, :vat, unique: true
  end
end
```

Utilizzando il metodo `change`, le migrazioni possono modificare la struttura del database secondo quando specificato nella migrazione, senza necessità di ricorrere a *downtime* del server e di eseguire il rollback alla versione dello schema del database precedente, se fosse necessario.

Il generatore produce altri due file, oltre alla migrazione:

```ruby
# app/models/organizer.rb

class Organizer < ActiveRecord::Base
  belongs_to :platform
end
```

```ruby
# spec/models/organizer_spec.rb

require 'rails_helper'

RSpec.describe Organizer, type: :model do
  pending "add some examples to (or delete) #{__FILE__}"
end
```

Il primo contiene la definizione della classe, in cui andranno inseriti i metodi, le validazioni e le associazioni sul modello, descritte in §5.1.2 e §5.1.3. Nel secondo andranno definiti i test di unità per il modello, descritti in §5.4.

5.1.2 Associazioni a modelli e file

Una volta generata la struttura del modello attraverso le migrazioni del database, è stato necessario associare tra loro i modelli, al livello dell'applicazione, secondo le relazioni espresse nel diagramma ER prodotto durante la fase di analisi e refactor (§3). Per farlo, sono stati utilizzati i metodi forniti da `ActiveRecord::Base`, classe ereditata da tutti i modelli. Nel progetto, in realtà, tutti i modelli ereditano dalla classe `ApplicationRecord`, che a sua volta eredita da `ActiveRecord::Base`, ma viene utilizzato per aggiunge metodi di utilità comuni a tutti i modelli implementati.

Rails incentiva l'implementazione di associazioni bidirezionali, attraverso l'utilizzo dei metodi:

- `belongs_to`: utilizzato per specificare l'associazione con il modello di cui la classe memorizza la chiave esterna;
- `has_one`: specifica l'associazione con un record di un modello che memorizza la chiave esterna alla classe;
- `has_many`: specifica l'associazione con più record di un modello che memorizza la chiave esterna alla classe;
- `has_and_belongs_to_many`: permette di specificare associazioni del tipo "molti a molti", utilizzando una tabella, creata manualmente, che possiede le chiavi esterne ad entrambi i modelli coinvolti.

Oltre alle associazioni con i modelli sono state specificate le associazioni con i file. Queste vengono gestite con la gemma Active Storage, che fornisce i metodi per eseguire l'associazione (*attach*) dei file, chiamati *attachments*: `has_one_attached` e `has_many_attached`.

Proseguendo con l'esempio dell'implementazione degli organizzatori, il file della classe `Organizer` con le associazioni dichiarate risulta essere il seguente:

```ruby
# app/models/organizer.rb

class Organizer < ApplicationRecord
  belongs_to :platform
  belongs_to :creator, class_name: 'User', inverse_of: :created_organizers,
  ↪  optional: true

  has_many :locations, dependent: :nullify

  has_one_attached :logo
end
```

L'associazione `has_many` è l'altra estremità di un'associazione `belongs_to` dichiarata nel modello `Location` e si basa sulla convenzione dei nomi di Rails per cui `Location` deve avere una colonna nel database chiamata `organizer_id`, che punta all'identificativo di un organizzatore. Serve per rendere accessibili le *location* gestite dall'organizzatore, avendo il record dell'organizzatore stesso.

5.1.3 Validazioni

L'ultima fase di sviluppo della parte di modello sono le validazioni sugli attributi e le associazioni, che vengono codificate utilizzando principalmente i validatori forniti da Active Record, da altre gemme o creati manualmente. Le validazioni inseriscono dei messaggi di errore in un array associato al modello che si sta validando; prima di memorizzare nel database i dati del record che si vuole salvare, ad esempio chiamando i metodi `save` o `create`, viene chiamato il metodo `valid?`, che controlla che l'array di errori sia vuoto, altrimenti è possibile accedervi e mostrare gli errori, anche come risposta della API. Le validazioni fornite da Active Record permettono di verificare vari aspetti degli attributi, come la presenza di un valore (`NOT NULL`), il formato di una stinga o i controlli sui valori numerici e molti altri.

Aggiungendo le validazioni sugli attributi, il file di esempio degli organizzatori diventa:

```
# app/models/organizer.rb

class Organizer < ApplicationRecord
  belongs_to :platform
  belongs_to :creator, class_name: 'User', inverse_of: :created_organizers,
  ↪  optional: true

  has_many :locations, dependent: :nullify

  has_one_attached :logo
  validates :logo, attached: true, content_type: ['image/png', 'image/jpeg']

  validates :name,
            :vat,
            :address,
            presence: true

  validates :vat, uniqueness: true

  validates :creator, presence: true, on: :create
  validate :creator_not_changed

private

  def creator_not_changed
    errors.add :creator, :cannot_change if creator_changed? && persisted?
  end
end
```

5.1.4 Associazione a `creator`

Quasi tutti i modelli implementati nel corso del progetto hanno un'associazione con il modello degli utenti, che traccia l'utente che ha creato il record. Sono necessarie diverse righe di codice per implementare questa funzionalità, non sempre intuitive oltretutto, quindi ho deciso di incapsculare questa configurazione all'interno di un concern, una funzionalità offerta da Active Support, che sfrutta i moduli di Ruby. Dopo l'operazione di refactor il file del modello di esempio diventa:

```
# app/models/organizer.rb

class Organizer < ApplicationRecord
  include Creator
  belongs_to :platform

  has_many :locations, dependent: :nullify

  has_one_attached :logo
  validates :logo, attached: true, content_type: ['image/png', 'image/jpeg']
```

```
  validates :name,
      :vat,
      :address,
      presence: true

  validates :vat, uniqueness: true
end
```

5.2 Controller

I controller sono le classi che si occupano di rispondere alle chiamate effettuate agli endpoint esposti dalla API. I metodi che gestiscono queste richieste sono chiamati "action" in Rails. Di seguito viene descritta l'implementazione delle action realizzate nei controller.

5.2.1 Implementazione delle action

Index

Implementa gli endpoint descritti in §4.3.1 e §4.3.7. Verifica che l'utente sia autorizzato ad eseguire questa azione ed effettua il rendering paginato della lista dei record richiesti, dopo che questa è stata filtrata in base ai permessi dell'utente che ha effettuato la richiesta. Serializza i record mostrati, utilizzando un serializzatore conforme a quanto deciso durante la fase di progettazione e descritto in §4.3.1.

Show

Implementa l'endpoint descritto in §4.3.2. Cerca e carica il record con l'identificativo richiesto dal database. Se questo è presente, verifica che l'utente sia autorizzato a visualizzarlo e lo restituisce, serializzato in modo conforme a quanto dichiarato nella descrizione dell'endpoint.

Create

Implementa gli endpoint descritti in §4.3.3 e §4.3.8. Controlla che tutti i parametri passati nel corpo della richiesta siano permessi e procede con la creazione del nuovo record. Prima di salvarlo nel database si assicura che l'utente sia autorizzato a creare quel record. Se la creazione avviene all'interno di un altro record, questo viene associato con quello appena creato. Infine associa i file al record, se richiesto e previsto, e lo restituisce serializzato come descritto in §4.3.2.

Update

Implementa l'endpoint descritto in §4.3.4. Cerca e carica il record con l'identificativo richiesto dal database. Se questo è presente, verifica che l'utente sia autorizzato a modificarlo. Controlla che tutti i parametri passati nel corpo della richiesta siano permessi e procede con la modifica degli attributi del record, corrispondenti ai parametri passati. Associa i file al record, se richiesto e previsto, infine lo restituisce serializzato come descritto in §4.3.2.

Destroy

Implementa l'endpoint descritto in §4.3.5. Cerca e carica il record con l'identificativo richiesto dal database. Se questo è presente, verifica che l'utente sia autorizzato a eliminarlo. Procede con l'eliminazione fisica o logica, dove previsto, e infine lo restituisce serializzato come descritto in §4.3.2.

5.2.2 APIController

`APIController` è la classe base da cui eredita ogni controller della API implementato. Include del codice per gestire automaticamente le eccezioni che possono essere sollevate dal backend e il relativo *rendering* dell'errore. A quelli già presenti ho deciso di aggiungere la gestione delle eccezioni:

- `Pundit::NotAuthorizedError`: sollevata in caso di autorizzazione fallita. Restituisce il codice HTTP `403`;
- `ActionController::UnpermittedParameters`: sollevata nel caso in cui sia presente un parametro non consentito nel corpo di una richiesta. Restituisce il codice HTTP `400`.

Inoltre, ho deciso di rifattorizzare due operazioni che si ripetevano in quasi tutti i controller:

***Attach* dei file** I metodi `create` e `update` accettano un hash (simile a un array associativo di altri linguaggi), quindi è sufficiente passare ai metodi i parametri presenti nel corpo della richiesta, per assegnare il valore dei normali attributi di un record. Gli *attachment* non vengono considerati normali attributi, quindi devono essere associati uno alla volta al record. Questo porta a una grande quantità di codice *boilerplate*, quindi ho creato il metodo `attach_files_to record, from_attributes_in: :attachments`, che associa ogni file contenuto nel parametro della richiesta, specificato con `from_attributes_in`, al record `record`.

***Rendering* della paginazione** Nelle risposte paginate è necessario ritornare, oltre alla lista dei record, anche le informazioni relative al numero totale di pagine e di record disponibili. Tutto questo viene gestito dal metodo `render_pagination collection_hash, each_serializer:`, a cui bisogna passare un hash contenete la lista dei record e un serializzatore, che agisce su ogni record.

5.3 Gestione dei permessi

5.3.1 Policy

Per l'implementazione della gestione dei permessi, è stata utilizzata la gemma Pundit. Utilizza delle policy, dei *plain old ruby object* (PORO), chiamati `ModelPolicy`, per definire le condizioni per cui l'utente è autorizzato a eseguire una specifica azione sul record del modello `Model` interessato. I PORO sono classi che non hanno alcuna dipendenza dal framework o da librerie esterne.

5.3.2 Action

Per autorizzare una specifica action viene utilizzato un metodo con lo stesso nome, seguito dal punto di domanda, che nelle convenzioni di Ruby indica i metodi che ritornano un valore booleano. Questo metodo viene chiamato nel momento in cui, nella action di un controller, viene chiamato il metodo `authorize record`, dove `record` è il record interessato dalla action.

Ad esempio, se il metodo `update?` della policy `PlatformPolicy` ritorna `true`, allora l'utente che intende modificare il record interessato dalla action è autorizzato a farlo. L'implementazione dei metodi delle policy rispetta quanto deciso nella fase di progettazione, descritta in §4.4.1.

5.3.3 Scope

Le action `index` richiedono che la lista dei record da ritornare venga filtrata, restituendo solamente i record che l'utente ha il permesso di visualizzare. Questo è stato implementato nel metodo `resolve` della classe `Scope`, definita all'interno di ogni policy. Chiamando il metodo `policy_scope` nella action di un controller, passandogli la lista di record da filtrare, si ottiene la lista filtrata.

5.3.4 Parametri permessi

Per implementare le condizioni definite sui parametri permessi in §4.4.2, si definisce il metodo `permitted_attributes` all'interno della policy, che deve restituire sempre un array di simboli, rappresentanti i parametri permessi. Questo metodo, chiamato nella action di un controller, restituisce i parametri presenti nel corpo della richiesta, se sono tutti permessi, altrimenti solleva un'eccezione.

5.4 Test di unità

Durante la codifica dei modelli sono stati definiti alcuni test di unità, prevalentemente sulle validazioni dei dati. Non sono stati testati tutti i modelli e, in particolare nessun controller, perché non era un'attività richiesta esplicitamente dal committente, quindi mi è stato chiesto dal *project manager* di investirci una quantità limitata di tempo. I test definiti sono stati implementati utilizzando la gemma RSpec, che fornisce strumenti per lo sviluppo *behaviour-driven development* (BDD), come da prassi aziendale di Moku.

Capitolo 6

Conclusioni

6.1 Raggiungimento dei requisiti

Tabella con stato di completamento dei requisiti, con commento (dove necessario)

6.2 Valutazione personale

Messe alla prova le competenze fornite dal corso di laurea, verificata l'efficacia dei corsi e dei progetti svolti, imparato un nuovo linguaggio e framework con filosofia di sviluppo a me nuova, scoperto ambiente lavorativo aziendale con i ruoli e le dinamiche interne.

Bibliografia

Siti web consultati

Documentazione ufficiale di Ruby on Rails - metodo `add_column`. URL: https://api.rubyonrails.org/classes/ActiveRecord/ConnectionAdapters/SchemaStatements.html#method-i-add_column.