

Corso di Programmazione Web e Mobile A.A. 2020-2021

PikBook

◀ **Alessandro Fabio, Farè, 910928** ▶

Autore: Alessandro Fabio Farè
Ultima modifica: 15 maggio 2020 - versione 1.0.1
Prima modifica: 24 aprile 2020

PikBook

Picture Book

1. Introduzione

PikBook è un Social Network che dà la possibilità ai propri utenti di condividere progetti personali tramite contenuti testuali e immagini.

E' un'applicazione web Full Stack sviluppata tramite il pattern Model-View-Controller. I dati risiedono in un database online, MongoDB Atlas, e le foto pubblicate sono state collegate a un cloud storage online, Cloudinary, che permette di ridurre la dimensione delle foto e di applicare ottimizzazioni a piacere.

E' previsto un sistema di likes e di commenti dei post e di download delle foto: la particolarità sta nel fatto che i likes e i commenti saranno visibili solo se l'utente clicca sulla foto, in modo da distaccare la reale bellezza dei progetti dalla popolarità dell'utente caricante. Inoltre è presente una pagina dove poter vedere i migliori 50 progetti in ordine di likes ricevuti.

1.1. Breve analisi dei requisiti

1.1.1. Destinatari

Capacità e possibilità tecniche.

L'applicazione è stata pensata per adattarsi graficamente in modo automatico in base al dispositivo col quale viene utilizzato grazie a un responsive design.

L'interfaccia è intuitiva, ho cercato di utilizzare uno stile minimalista in modo da guidare l'utente a interagire senza particolari problemi con l'applicazione e in modo da non discostarmi con l'esperienza d'uso offerta dagli altri social network.

Ci si aspetta che possa essere usata da qualsiasi tipo di utenza che abbia un minimo di esperienza con i social ma anche da chi si sta da poco avvicinando a questo mondo.

Motivazione.

Si presta per essere un'applicazione di intrattenimento, ma potrebbe prevedere anche forme di business con successive implementazioni.

E' costruita su un modello di comunicazione di tipo "peer": più emittenti, alto livello di interazione. L'utenza dovrebbe essere attiva e predisposta a una ricerca diretta delle informazioni; è UGC, user-generated content.

1.1.2. Modello di valore

L'applicazione è pensata per dare all'utente la possibilità di mostrare al pubblico le proprie capacità mostrando i propri progetti personali. In questo modo può auto-promuoversi creando una personale vetrina virtuale. Non essendo subito visibili i like e i commenti, il valore del progetto verterà sulla bellezza di quest'ultimo e non sulla popolarità dell'utente.

Attualmente è completamente free per ogni utente e non prevede alcuna sottoscrizione di abbonamento. Inoltre non è previsto alcun scambio o distribuzione di denaro ma i progetti potrebbero divenire anche dei beni virtuali creando uno shop online in futuro.

1.1.3. Flusso dei Dati

L'app si appoggia sulla condivisione da parte dell'utente dei propri contenuti personali, resi pubblici; sono quindi prodotti dall'utenza.

L'altro tipo di contenuto sul quale si appoggia sono i dati degli utenti, sia quelli appena creati grazie alla registrazione che quelli aggiornati tramite update del profilo o altre funzioni.

Tutti i contenuti sono archiviati in un database online MongoDB Atlas e le foto in un cloud storage online, Cloudinary.

Non c'è alcun costo di produzione o riadattamento dei contenuti, a meno che non venga superata la soglia di memoria nel database.

1.1.4. Aspetti tecnologici

L'aspetto tecnologico di PikBook può essere scomposto in tre macro blocchi.

1) **Front-End:**

Pikbook è un'applicazione web creata con css e Pug.js per gestire il layout, con Javascript per la parte di logica di comportamento e con Ajax per quanto riguarda le chiamate in background al server.

2) **Back-End:**

Per il Back-End è stato utilizzato un server NodeJS che, tramite framework Express, riceve le richieste del client, le elabora interrogando il database, e risponde con un oggetto JSON. L'utente può anche innescare richieste POST Ajax che verranno gestite sul server da Express.

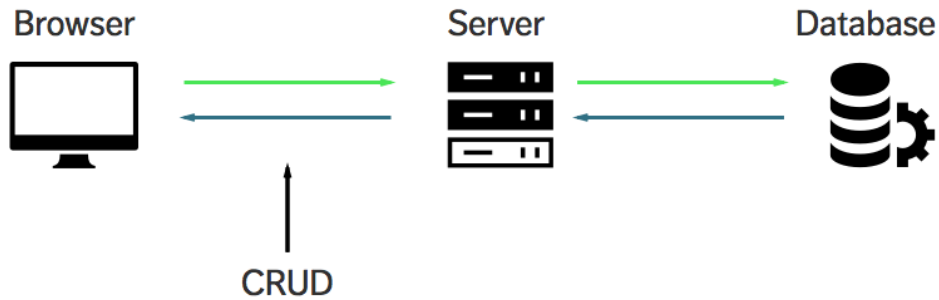
L'applicazione è stata deployata su Heroku, una piattaforma Paas che permette di distribuire le applicazioni NodeJS online. Ho utilizzato il piano gratuito di Heroku, il quale prevede che, dopo mezz'ora in cui non viene utilizzata, l'app si "addormenta".

Così ho collegato PikBook a un'applicazione Heroku, chiamata "Kaffeine", che risveglia la mia applicazione ogni 30 minuti, e anche a Cron-job, che programma l'esecuzione di PikBook in base ai parametri scelti.

Infine, dato che Heroku non salva le foto pubblicate dagli utenti in locale, ho collegato le foto al cloud storage online di Cloudinary in modo da averle sempre presenti.

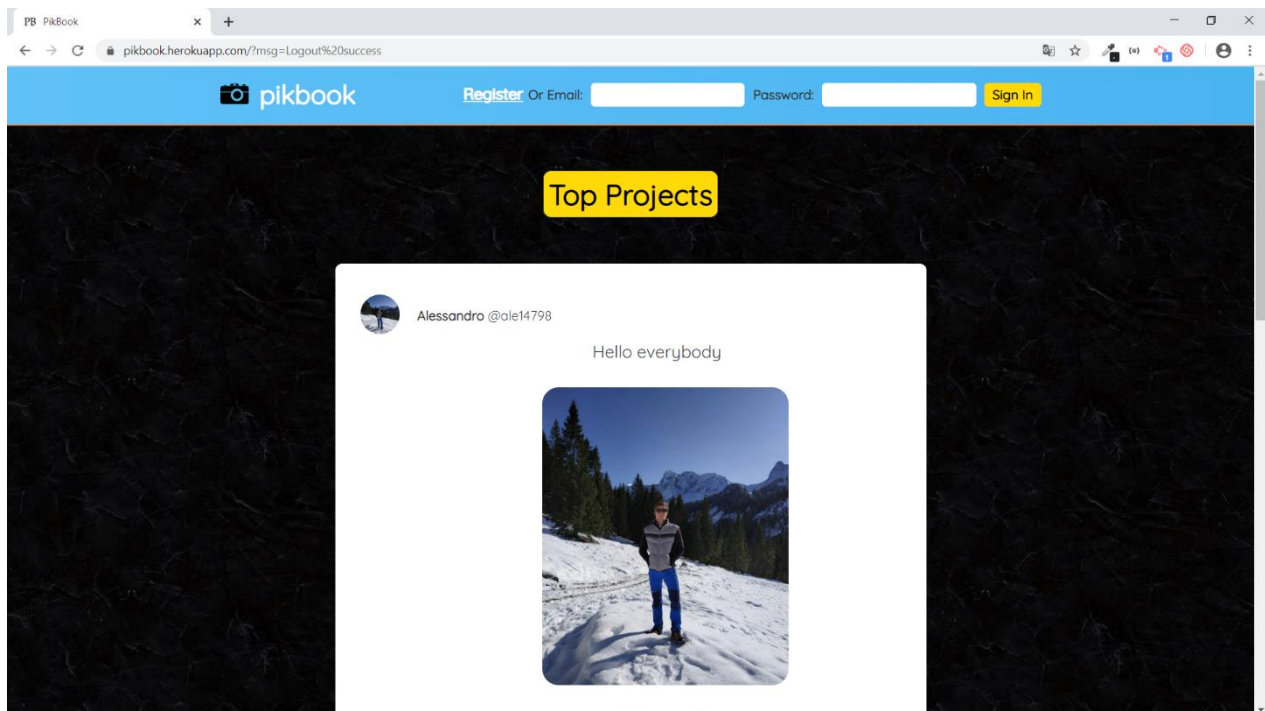
3) **Database:**

Come Database è stato utilizzato MongoDB Atlas. La scelta è ricaduta su quest'ultimo in quanto intuitivo, facile da utilizzare e, rispetto ad altri database, è più sicuro, ha maggiori performance ed è adatto per applicazioni di larga scala.



2. Interfacce

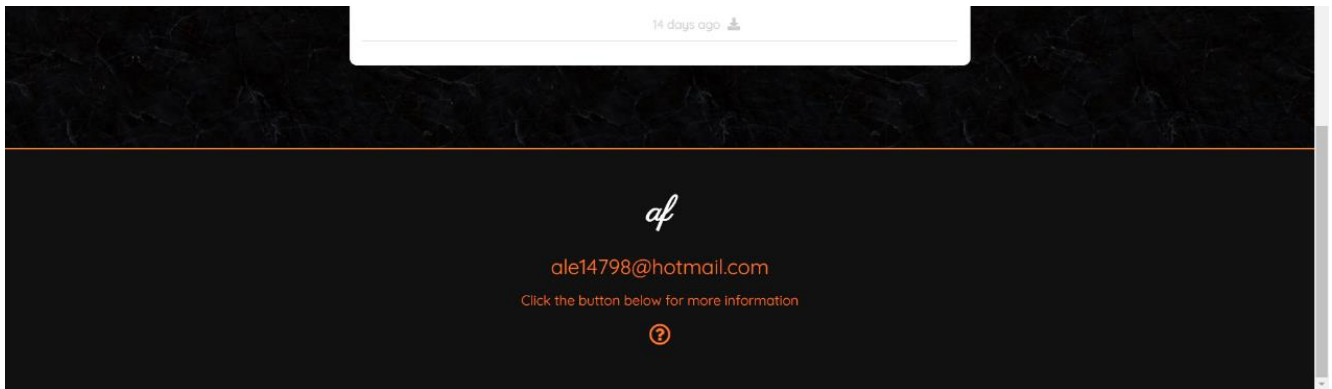
Home



Nell'interfaccia principale, quella di Home, è presente:

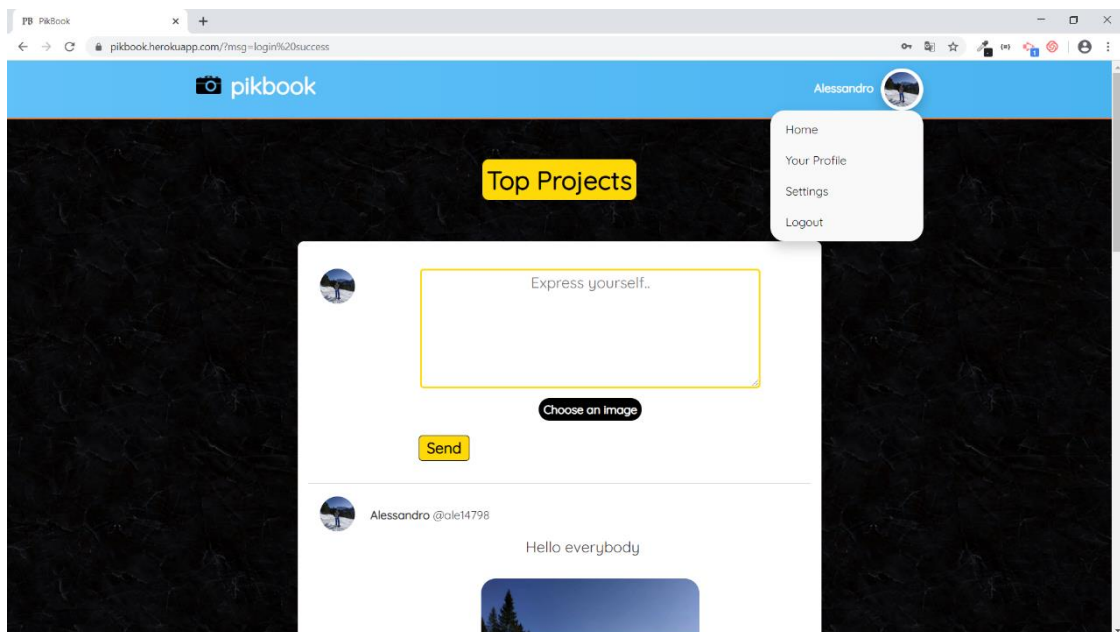
- In alto:
 - . a sinistra il logo dell'applicazione
 - . a destra il form per il login
- In mezzo:
 - . in alto un bottone che reindirizza alla pagina coi migliori 50 progetti
 - . in mezzo i post pubblicati dagli utenti
 - . lo sfondo è un'immagine presa da internet, modificata con css

- In basso



In basso c'è il footer, con le iniziali del mio nome e cognome, con la mia email e con un bottone che tramite chiamata Ajax interroga un file txt dove è scritta la descrizione sintetica dell'applicazione in inglese: "PikBook is a social network based on sharing images. It allows users to create photo books where they can share personal projects."

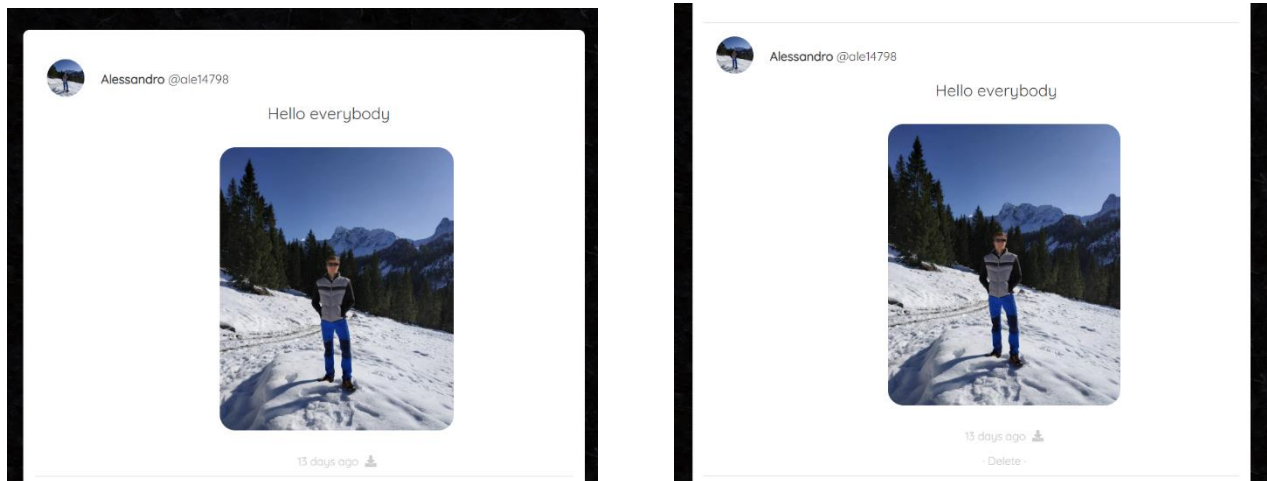
Home dopo Login



Dopo che l'utente ha effettuato il login correttamente (in caso contrario rimarrebbe nella stessa pagina di prima con l'url composto da "/?msg=login%20failed"), alla pagina home, al cui url è stato aggiunto "/?msg=login%20success", viene inserito il form per scrivere il post e nell'header a destra è stato tolto il form per la registrazione e aggiunto il nome utente con la relativa foto profilo. Nel caso in cui l'utente non avesse un'immagine profilo viene aggiunta un'immagine di default. Nel form per i post è presente una textarea dove l'utente può scrivere del testo, un bottone per uploadare l'immagine e in basso a sinistra il bottone per postare.

Muovendo il mouse sull'immagine profilo dell'header appare una navbar contenente i link per le rispettive pagine: Home, Profile, Settings e anche un link per effettuare il Logout.

Post



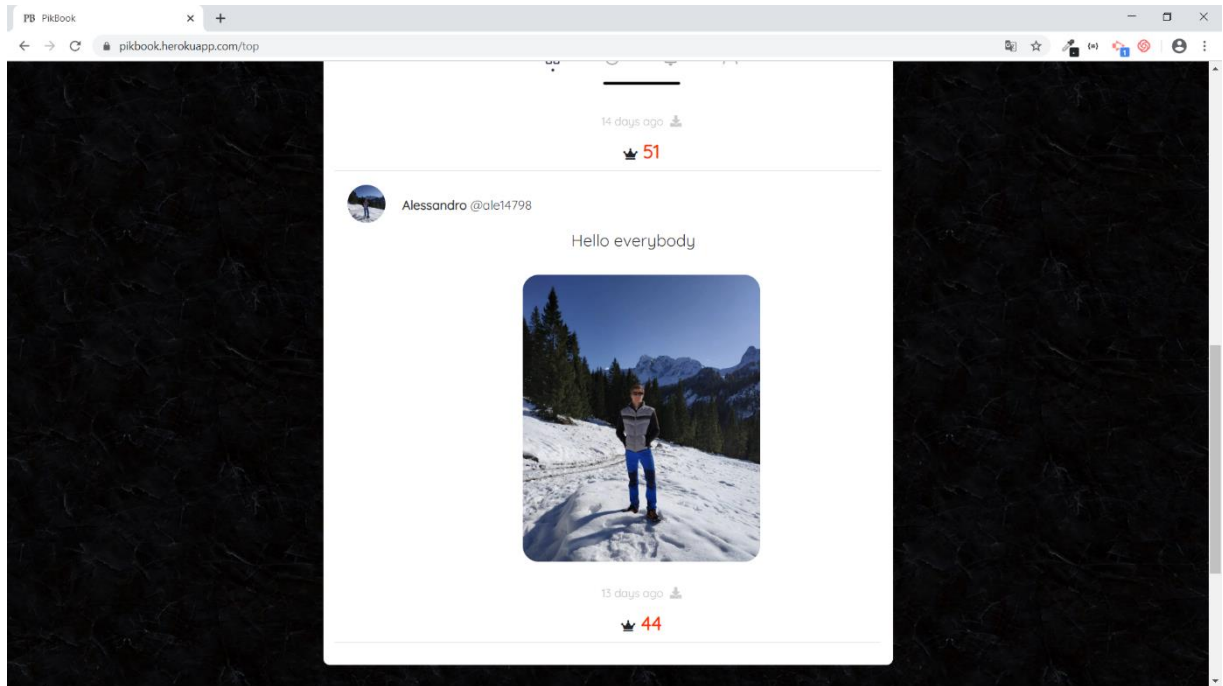
A sinistra si vede il post prima che l'utente si sia loggato, a destra dopo che si è loggato.

La differenza sta nel fatto che dopo che l'utente si è loggato, può andare a eliminare il suo post con il link Delete posto all'ultimo livello del post.

In generale il post è composto in alto a sinistra dall'immagine profilo dell'utente che ha caricato il post, dal suo nome utente e dal suo username preceduto da una @. In mezzo è presente il contenuto del post, con il testo in alto e l'immagine posta sotto. In basso sono presenti i "post meta", con il tempo che intercorre da quando è stato pubblicato il post al momento in cui si sta guardando la pagina, realizzato tramite la libreria Moment di npm, un bottone con l'icona del download per scaricare le foto pubblicate, e sotto il link Delete per chi è loggato, come spiegato precedentemente.

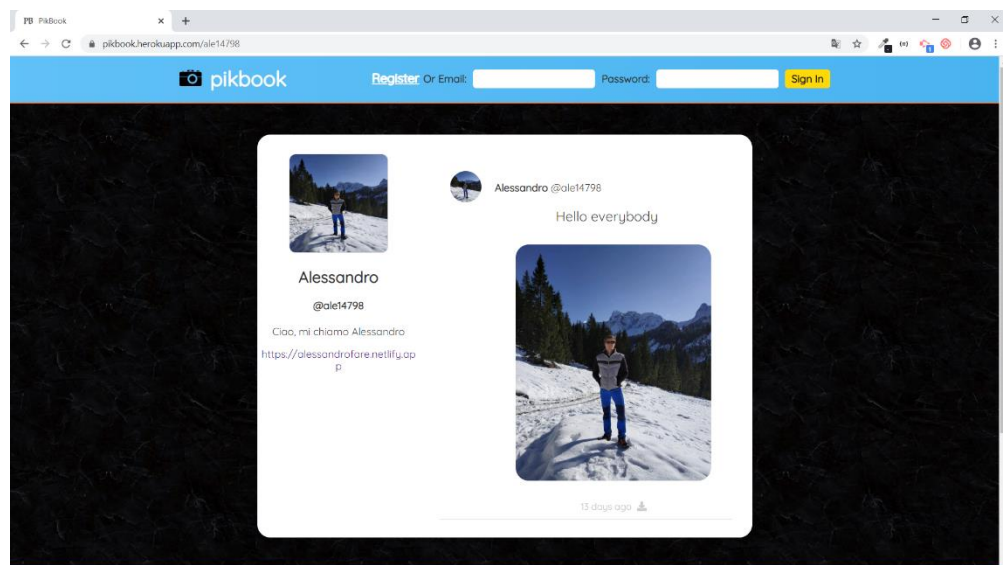
I post sono distanziati da un bordo in basso color grigio chiaro, stesso colore dei "post meta".

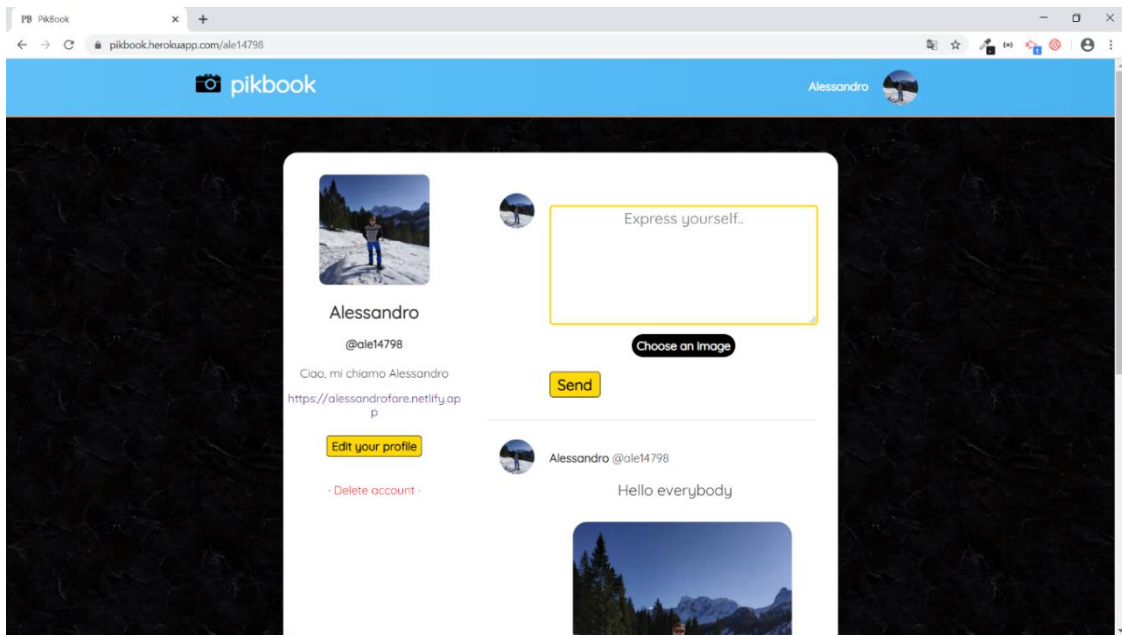
Top Projects



Questa è l'interfaccia dove sono mostrati i migliori 50 progetti in ordine di like, dal post con più like in giù (come si può vedere il post con 51 likes è più in alto di quello da 44 likes). I post si presentano come prima, con la differenza che in basso è presente un'icona della corona affiancata al numero di likes del post. Questa pagina è raggiungibile attraverso il bottone "Top Projects" presente nella home.

Profile



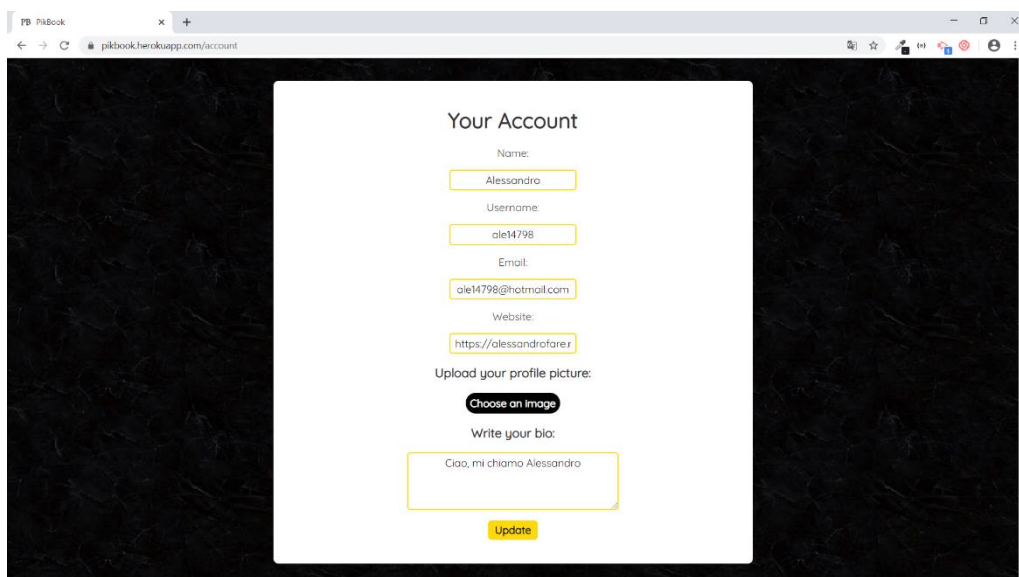


Nella prima immagine è mostrato il profilo di un utente esterno, nella seconda invece il nostro profilo dopo esserci loggati.

L'interfaccia di profilo possiamo dividerla in due colonne:

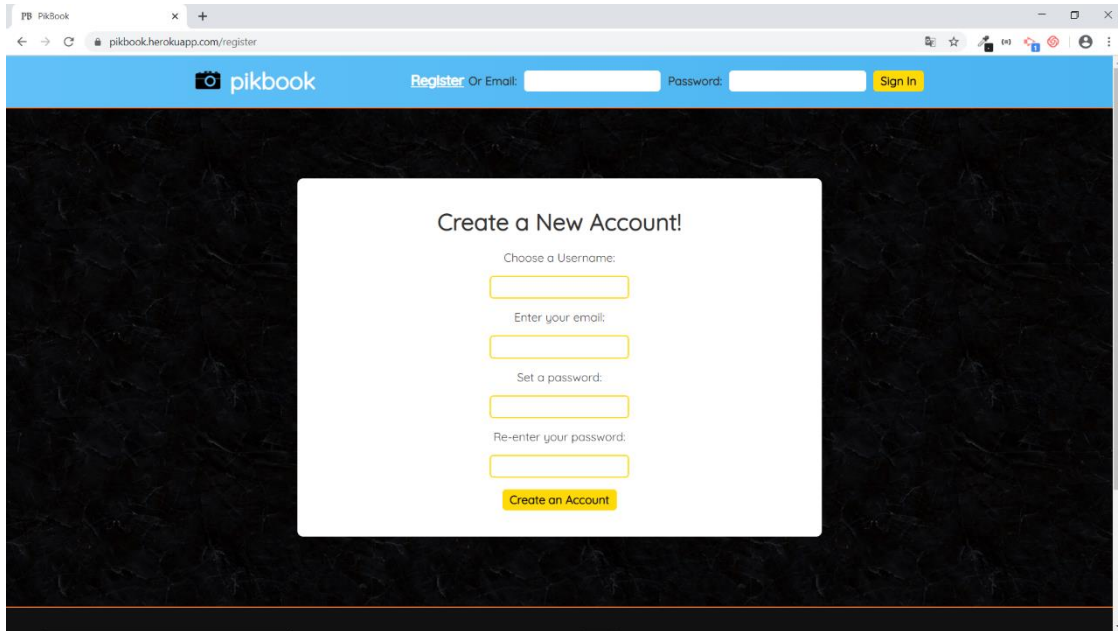
- . nella colonna a sinistra sono presenti le informazioni dell'utente con, in ordine, l'immagine profilo, il nome utente, l'username, la bio, un website di riferimento e, se è il nostro profilo, un bottone per editare il profilo, che riporta alla pagina per cambiare il profilo e infine un link per cancellare il profilo e tutti i nostri post (per evitare che l'utente sbagli a cliccare, ho inserito una window confirm prima di far partire l'eliminazione).
- . nella colonna a destra sono presenti tutti i post dell'utente e, se è il nostro profilo, la form per postare un altro post.

Settings



Nella foto sopra è mostrata la pagina di settings, dove l'utente può modificare il proprio profilo: name, username, email, foto profilo e bio. Una volta schiacciato il bottone Update tutti i cambiamenti verranno applicati sia lato client che lato database.

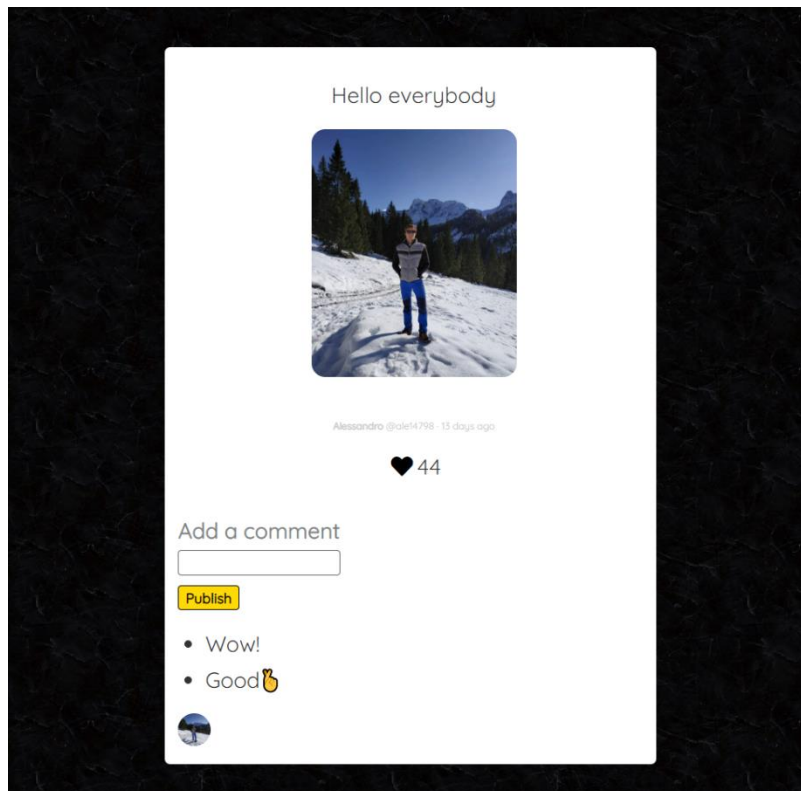
Register



The screenshot shows a web browser window with the URL `pikbook.herokuapp.com/register`. The page features a blue header with the 'pikbook' logo and a navigation bar containing 'Register Or Email:' and 'Password:' input fields, along with a yellow 'Sign In' button. The main content area has a dark background with a white box titled 'Create a New Account!'. Inside this box, there are four input fields: 'Choose a Username:', 'Enter your email:', 'Set a password:', and 'Re-enter your password:'. A yellow 'Create an Account' button is positioned at the bottom of the white box.

Simile alla pagina di Settings è la pagina di registrazione, dove un nuovo utente può registrarsi scegliendo un username, l'email e la password. Una volta cliccato il bottone 'Create an Account' l'utente viene reindirizzato alla pagina home, pronto per inviare il suo primo post.

Like & Comment

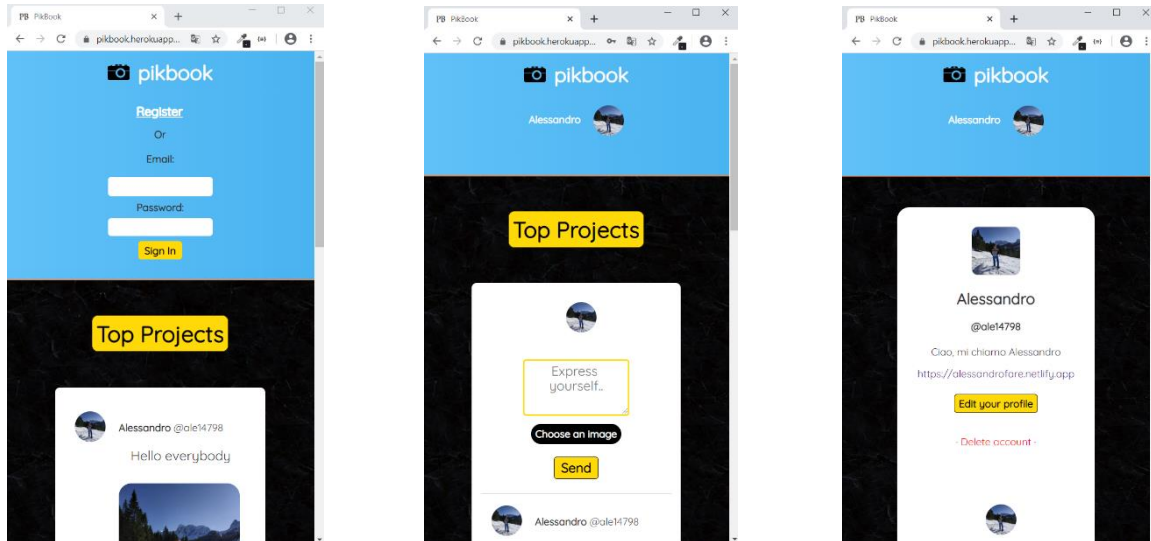


Cliccando sulla foto nella pagina di Home o sulla data in cui è stato caricato il post veniamo reindirizzati in questa pagina. Qui è presente il post con, al posto dell'icona del download, un bottone dei likes a forma di cuore affiancato dal numero dei likes del post. Una volta cliccato il bottone il cuore diventa rosso e il numero di likes aumenta di 1. Ciò è stato realizzato tramite una chiamata Ajax gestita da una funzione di POST Request di Express che va ad aumentare il numero di likes di uno nel database e restituisce il numero ad Ajax che tramite JQuery va a trasformare la pagina in tempo reale senza il refresh della pagina.

Stesso ragionamento accade per il sistema dei commenti, posto sotto: l'utente scrive nell'input sotto il label "Add a comment", il messaggio viene letto e aggiunto ai commenti del post nel database tramite una funzione Express, che manda inoltre il messaggio ad Ajax. Ajax appende il messaggio ricevuto alla lista di commenti in tempo reale.

Responsive Design

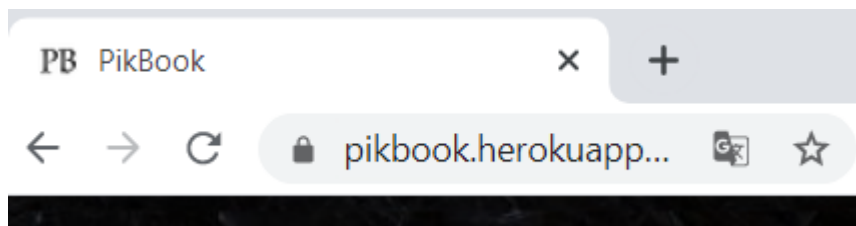
Ogni pagina è stata pensata per adattarsi graficamente al dispositivo utilizzato. Di seguito alcuni screenshot con la larghezza della pagina del browser al minimo:



Il responsive design funziona anche su dispositivi mobile; inoltre, dato che su dispositivi apple il selector `:hover` non funziona normalmente, ho dovuto inserire nel file di layout.pug:

```
body(tabindex='0')
```

Favicon.ico



Come si vede dalla foto l'applicazione ha anche una favicon, che rappresenta le iniziali del nome dell'app: P = Pik, B = Book.

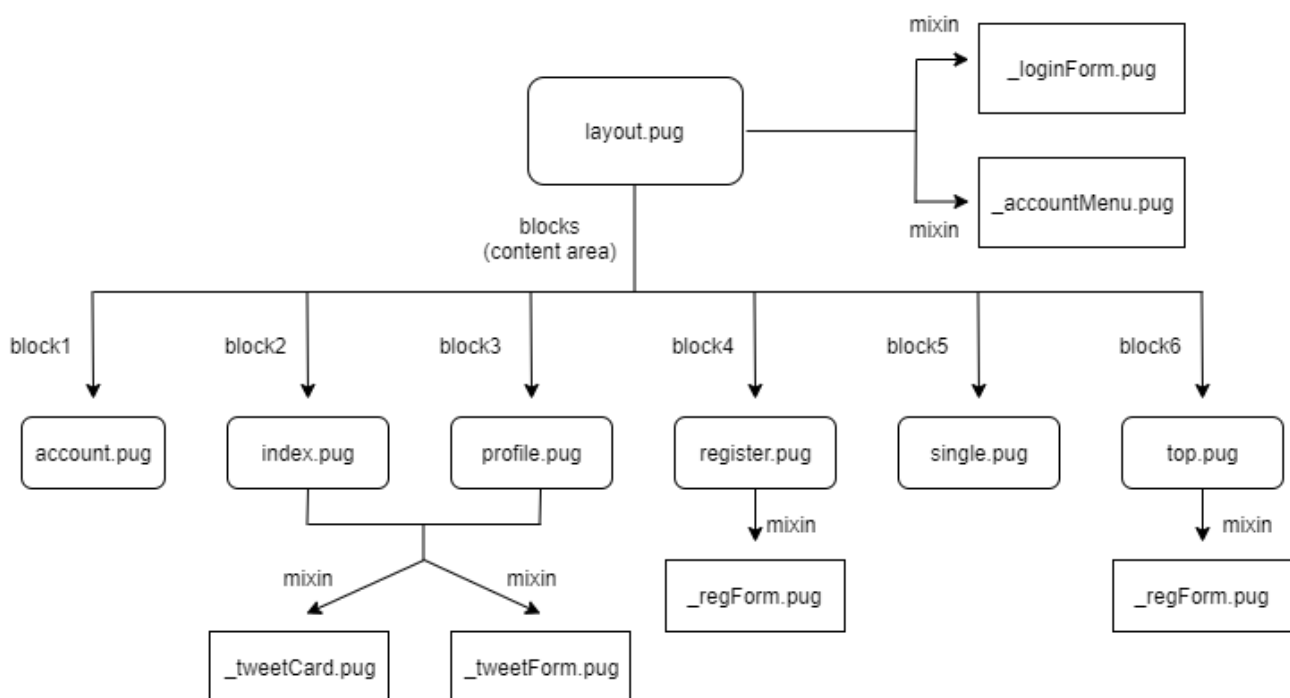
3. Architettura

3.1. Diagramma dell'ordine gerarchico delle risorse

3.1a Diagramma delle risorse Client

Le potenzialità del template engine Pug.js mi hanno permesso di creare una struttura gerarchica delle pagine client: infatti è possibile creare dei veri e propri schemi tra le varie classi grazie ai block e i mixin. Questo mi ha permesso una pulizia notevole del codice e la possibilità, grazie a sottoclassi, di non dover riscrivere codice uguale per varie pagine.

- Diagramma



La pagina `layout.pug` contiene i tre elementi fondamentali della struttura gerarchica delle risorse Client: header, content_area e footer.

L'header e il footer persistono per ogni pagina; il content_area, contenente la struttura del block, invece cambia per ogni pagina, avendo così 6 pagine relative a `layout.pug` e di conseguenza 6 diversi blocks.

Inoltre alcuni blocks contengono anche dei mixin. Grazie infatti alla parola chiave "include" è possibile inserire dei file Pug dentro ad altri file Pug, ed è ciò che succede per i mixin.

Per quanto riguarda i file di stile sono stati usati 3 file css: `style.css`, `grid.css` e `normalize.css`. Il primo è stato usato per modificare lo stile delle pagine, il secondo per modellare la struttura della pagina (larghezza e altezza) e per il responsive design tramite media queries, il terzo per avere compatibilità degli elementi Pug tra i diversi browser.

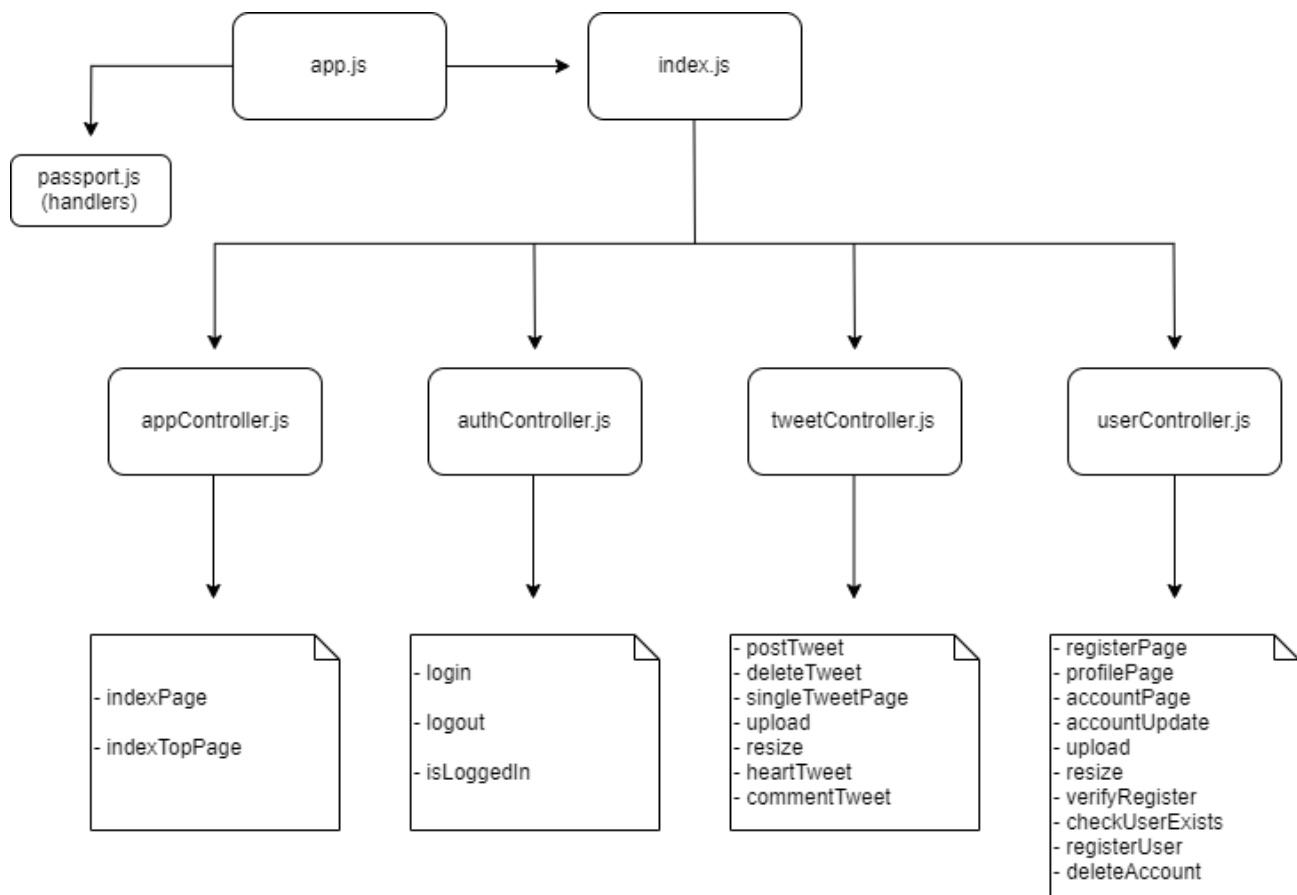
Infine per quanto riguarda la parte di logica di comportamento sono stati utilizzati degli script all'interno dei file Pug.

In layout.pug c'è uno script Ajax per aprire un file txt che risiede nella cartella public.

In single.pug ci sono due script Ajax JQuery: il primo per attivare la POST Request per aggiungere un like al post e per colorare l'icona del cuore di rosso, il secondo per attivare la POST Request per aggiungere un commento alla lista di commenti del post.

Infine nel file index.pug sono state usate le api HTML5 Geolocation e Web Storage per scrivere in console la latitudine e la longitudine dell'utente collegato.

3.1b Diagramma struttura Server



App.js è il file principale che serve per configurare il server, i file statici e le variabili nascoste del database e del cloud. Nella figura è collegato all'handler Passport.js che permette di avere informazioni dell'utente grazie ai cookie, e al router Index.js che è il file contenente tutte le GET e POST Request. Al router sono collegati tutti i 4 controller che contengono ciascuno le funzioni per le richieste al server, rappresentate dalla figura a forma di nota.

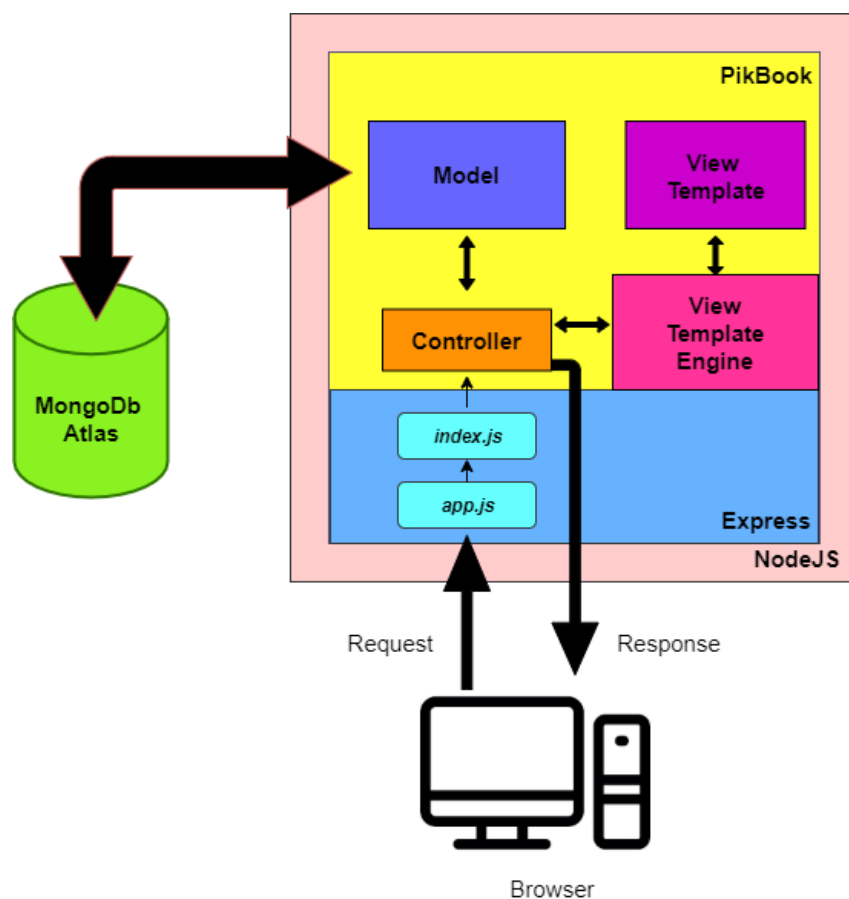
3.2. Descrizione delle risorse

L'applicazione è sviluppata tramite il paradigma MVC.

I componenti in gioco sono:

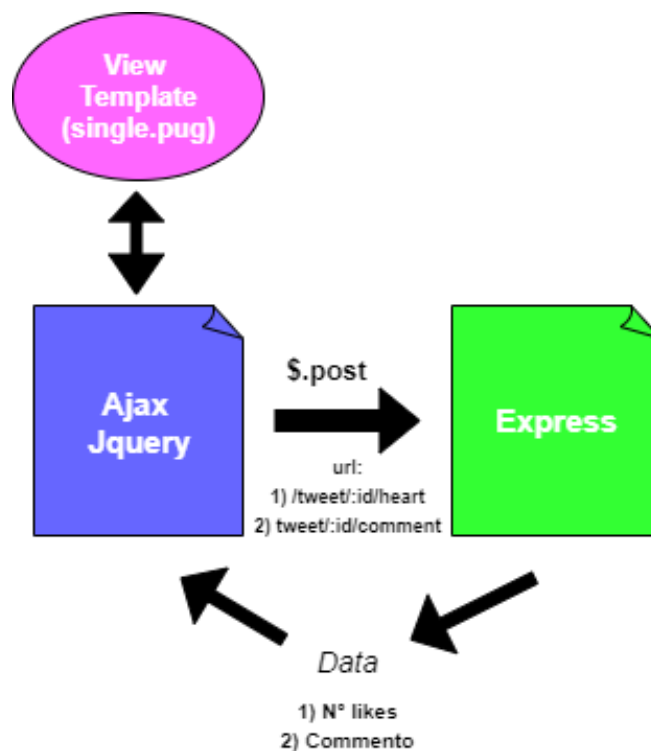
- **Model:** abbiamo 2 Model, User e Tweet, che rappresentano rispettivamente i dati dell'utente e dei post
- **View:** per la View è stato usato il template engine Pug, che mi ha permesso di creare un view template in grado di ricavare i dati aggiornati direttamente dopo le chiamate al server
- **Controller:** abbiamo 4 controller, appController, authController, userController e tweetController, che rappresentano rispettivamente:
 - . il controller per indirizzare alla homepage e alla pagina della classifica dei progetti
 - . il controller per l'autenticazione del login e per controllare il login e il logout tramite l'aggiunta all'url del messaggio di errore o successo e successivo reindirizzamento
 - . il controller per l'aggiornamento dei dati dell'utente, per la sua registrazione e aggiornamento del profilo e per l'indirizzamento a pagine relative all'utente (profilo, registrazione e update del profilo)
 - . il controller per la gestione dei post, ovvero inviare il post, eliminarlo, mettere like, inviare commenti e uploadare l'immagine.

In generale il flusso di risorse avviene come spiegato nell'immagine sotto:



Quando il client invia una richiesta al server, essa viene gestita da `index.js`, che è il mio router, elaborata dai controller che aggiornano i dati trasformando i dati dei Model presi dal database online MongoDB Atlas; questi dati vengono inviati tramite `res.render` al template engine che avrà i dati aggiornati in tempo reale e adatterà la view secondo questi nuovi dati. Tutto ciò che riguarda il server avviene grazie a NodeJS e in particolare al framework Express, oltre ad altre librerie utili di npm (come `multer`, `jimp`, `uuid` e altre ancora).

Per quanto riguarda i controller per le funzioni di `heartTweet` e `commentTweet`, essi gestiscono le funzioni di POST Request innescate tramite Ajax JQuery; il funzionamento è il seguente:



Per spiegare il comportamento delle transazioni delle risorse prenderò in considerazione una view andando a spiegare ogni passaggio.

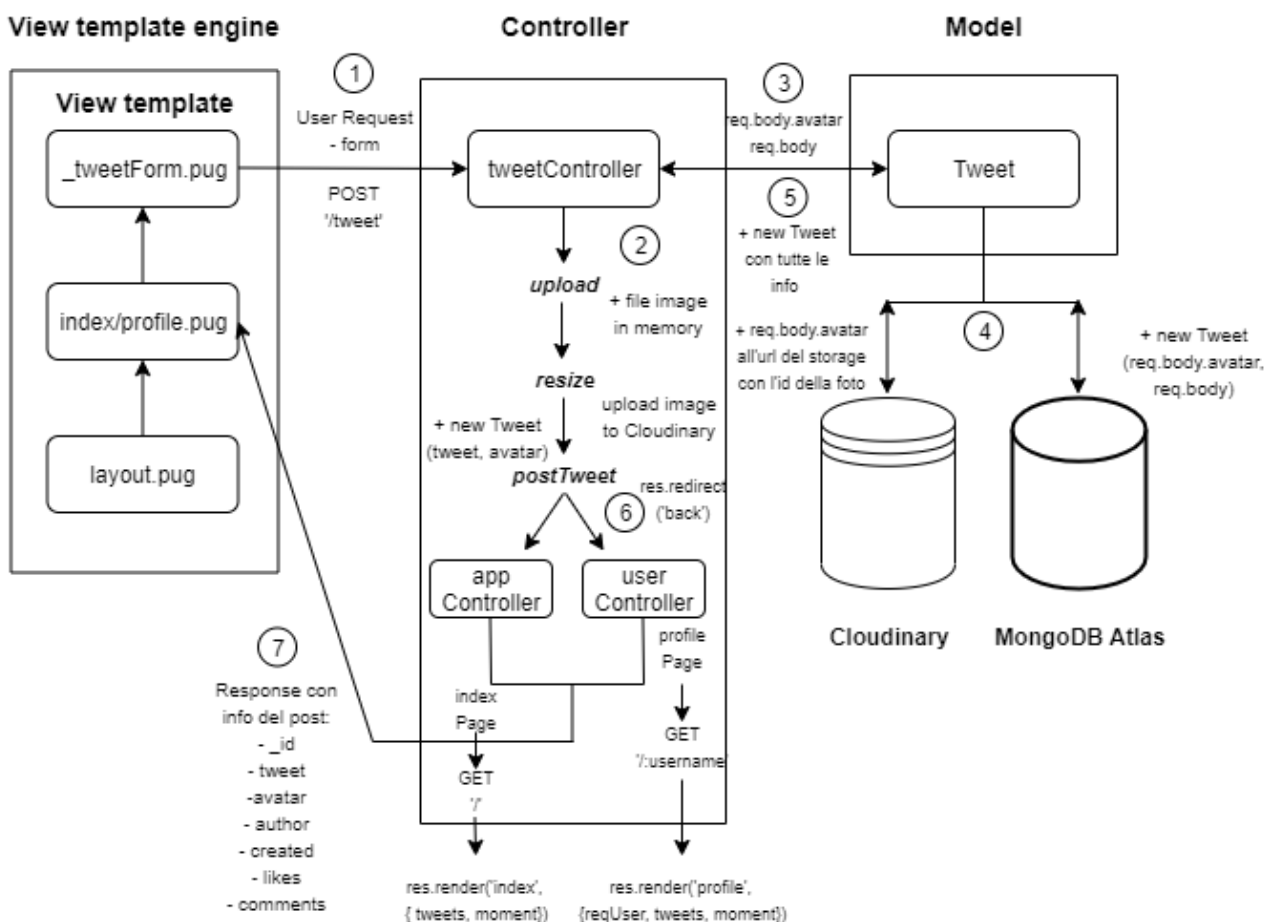
1) _tweetForm Transaction Schema

Nel file `_tweetForm.pug` abbiamo la chiamata POST per inviare il post con contenuto testuale e immagine. Una volta premuto il bottone con scritto “Send” partirà la richiesta POST all’indirizzo `/tweet`. La richiesta verrà elaborata dal `tweetController` il quale eseguirà tre funzioni asincrone in sequenza: `upload`, `resize` e `postTweet`.

`Upload` e `resize` servono per uploadare l’immagine scelta dall’utente per il post e sia per scriverla in memoria in locale nella cartella `uploads` che per uploadarla sul cloud storage online di `cloudinary` (in questo modo ce l’avrò sempre disponibile, sia che lavoro in locale che online).

La funzione `postTweet` serve invece per prendere sia il testo scritto dall’utente nella textarea che l’immagine uploadata, creare il nuovo post aggiungendo il testo e la foto, creando così un nuovo Model `Tweet` nel database online, e infine per redirigere l’utente, attraverso `res.redirect('back')`, alla pagina iniziale. Il server risponde quindi con un oggetto JSON del nuovo post creato contenente tutte le informazioni relative al modello: contenuto testuale, immagine, autore del post, data di creazione, numero di likes e commenti.

Queste informazioni vengono prese dalla view iniziale, `index.pug` (o se eravamo nel nostro profilo `profile.pug`), e le view vengono aggiornate coi nuovi dati.



3.3. Altri diagrammi

Qui di seguito il diagramma del database MongoDB Atlas. Ho creato una collezione, chiamata “test”, al cui interno ho due documenti: tweets e users.

Ecco un estratto di entrambi i documenti:

TWEETS:

```

1. _id: 5eb976e2e543ee0017ec9d9c
2. tweet: "Hey"
3. avatar: "0fd7369e-6f60-4eab-bfa0-99e8e3674da6.png"
4. author: 5eb976b9e543ee0017ec9d9b
5. created: 2020-05-11T16:01:38.576+00:00
6. __v: 0
7. likes: 51
8. comments: ["Wow!", "Good 🍌"]

```

USERS:

```

1. _id: 5eabd915f00a791b586a8c37
2. username: "ale14798"
3. email: "ale14798@hotmail.com"
4. created: 2020-05-01T08:08:53.428+00:00
5. salt: "3df3d7fc59872d9a4f3a8cc1f13b4cf89ddde7789368fc57100c8f8902e1b65b"
6. hash: "b3f1703360b98ce7f474f80f7d81b52a61f59a36e628489f3cf1f8140d1dd87971ae6f..."
7. __v: 0
8. Avatar: "98cbe1d3-5321-43f2-bd3d-9371a85e481b.jpeg"
9. bio: "Ciao, mi chiamo Alessandro"
10. name: "Alessandro"
11. website: "https://alessandrofare.netlify.app"

```

Dato che il modello Tweet ha come campo anche l’autore di chi ha fatto il post, ho utilizzato la parola chiave ‘ref’ in modo da creare quello che in mongoose è chiamato ‘Population’. Così posso riferirmi ad esempio all’immagine profilo dell’autore del post semplicemente con: tweet.author.avatar.

4. Codice

Comments

Di seguito il codice più significativo per il sistema di commenti dei post dell’applicazione. Importanti sono il form usato per commentare nella view single.pug contenente anche lo script di Ajax per la POST Request e la funzione commentTweet presente nel controller tweetController.

Single.pug

```

p.comment_actions
  form.comment(action=`/tweet/${tweet._id}/comment` method='POST' id="commentId")
    label(for='comment') Add a comment
    input.typecomment(type='text' name='comment' id='idComment')
    input.typepublish(type='submit' class='button' value='Publish')
  ul#list
    each comments in tweet.comments
      li= comments

script.
  $(document).ready(function(){
    $('#commentId').submit(function(e) {
      e.preventDefault();
      $.ajax({
        type: 'POST',
        url: `/tweet/` + $('#enter').val() + `/comment`,
        data: {myData: ($('#idComment').val())},
        success: function(res) {
          $('#list').append('<li>'+res.message+'</li>');
        },
        error: function() {
          console.log('Failed');
        }
      });
    });
  });
});

```

Nel form .comment c'è l'azione di POST Request all'url '/tweet/\${tweet_id}/comment'. \${tweet_id} è la variabile dell'id del post a cui stiamo dando il commento; questo è possibile grazie a res.render di NodeJS nel quale ho passato la variabile tweet e con cui ho quindi a disposizione tutti i parametri relativi al modello tweet nel file pug. Schiacciato l'input col valore di 'Publish' la richiesta viene letta dallo script Ajax il quale passa al server, come data, la stringa di testo scritta dall'utente nell'input typecomment. A questo punto il server elabora la richiesta a quell'url grazie al controller tweetController con la funzione commentTweet, che mostro qui di seguito:

commentTweet

```

exports.commentTweet = async (req, res) => {
  try {
    const comment = req.body.myData;
    const tweet = await Tweet.findByIdAndUpdate(
      req.params.id,
      { $push: {comments: comment} },
      { new: true }
    );

    res.send({message: comment});
  }
  catch (err) {
    console.log(err);
    res.redirect('/?msg=No tweets found')
  }
}

```

Avendo un file di router, chiamato index.js, tutte le funzioni dei controller sono dei moduli che grazie a exports vengono passate al router. In index.js la funzione è la seguente:

```
router.post('/tweet/:id/comment', tweetController.commentTweet);
```

La funzione commentTweet è una funzione asincrona in cui viene preso il dato mandato da Ajax, contenente il commento al post; questo commento viene aggiunto in coda all'array dei commenti del modello tweet relativo al post che stiamo commentando (opportunamente trovato grazie all'id contenuto nei parametri dell'url) tramite l'operatore di mongoose \$push. Una volta aggiornato il database il commento viene rimandato al client tramite res.send e letto dallo script di Ajax nella funzione success; in questa funzione il commento viene aggiunto in coda alla lista dei commenti nella view single.pug, così da avere in tempo reale il nuovo commento nella pagina.

Send Post

Qui invece spiegherò il sistema di invio del post e come viene visualizzato sulla homepage. Importanti sono il form per l'invio del post, contenuto nel file mixin _tweetForm.pug, la pagina di visualizzazione del nuovo post, _tweetCard.pug, e le tre funzioni express per uploadare le foto e aggiungere il post al database, grazie al controller tweetController.

tweetForm

```

mixin tweetForm()
  form.tweet_card.tweetform(action='/tweet' method='POST' enctype='multipart/form
-data')
    .tweet_avatar
      if !user.avatar
        img(src='/uploads/no-avatar.jpg')
      else
        img(src='https://res.cloudinary.com/pikbook/image/upload/${user.ava
tar}')
    .tweet_text
      .tweetcenter
        textarea(name='tweet' class='tweet_textarea' placeholder='Express y
ourself..' maxlength="140" rows='3' required)
        input(type='file' id='avatar' name='avatar' class='image_upload' ac
cepts='image/png, image/jpeg')
        label(for='avatar') Choose an image
      .tweet_button
        input(type='submit' class='button tweet_button' value='Send')

```

Il form `.tweet_card` fa innescare la POST Request all'url `/tweet`, una volta premuto l'input col valore `Send`. Il form è composto dall'immagine profilo dell'utente, dalla textarea nel quale va scritto il contenuto testuale del post(ho messo "required" perché preferivo dei post con foto e del contenuto testuale), dal bottone per scegliere l'immagine del post e infine l'input per la richiesta al server. Schiacciato l'input submit la richiesta viene elaborata dal server:

upload

```

//Upload images
const multerOptions = {
  storage: multer.memoryStorage(),
  fileFilter(req, file, next) {
    const isImage = file.mimetype.startsWith('image/');

    if(isImage) {
      next(null, true);
    } else {
      next({message: 'You must supply a valid image file.'}, false)
    }
  }
}

exports.upload = multer(multerOptions).single('avatar');

```


La prima funzione innescata all'url '/tweet' è quella di upload: questa funzione serve per uploadare l'immagine caricata dall'utente nell'input col name 'avatar'. E' stata utilizzata la libreria di npm Multer, la quale permette di gestire file multipart/form-data e uploadare file. La parola chiave .single('avatar') serve per leggere il file dell'input, il quale grazie a multerOptions è scritto in memoria come oggetto Buffer con il tipo di file 'image'.

resize

```
exports.resize = async (req, res, next) => {
  if(!req.file) {
    next();
    return;
  }
  const extension = req.file.mimetype.split('/')[1];
  req.body.avatar = `${uuid.v4()}.${extension}`;
  const image = await jimp.read(req.file.buffer);
  await image.resize(600, jimp.AUTO).quality(100);
  await image.writeAsync(`./public/uploads/${req.body.avatar}`);
  cloudinary.uploader.upload(`./public/uploads/${req.body.avatar}`, {public_id: `
  ${req.body.avatar}`.split('.')[0], format: `${req.body.avatar}`.split('.')[1]}, fun
  ction(result) {
    console.log(result);});
  next();
}
```

La seconda funzione innescata all'url '/tweet' è quella di resize: questa funzione serve per dare un _id all'immagine uploadata, per modificarla e per scriverla sia in locale che su Cloudinary. Sono state utilizzate le librerie uuid per dare l' _id all'immagine e Jimp per modificare la dimensione dell'immagine e la qualità.

postTweet

```
exports.postTweet = async (req, res) => {
  try {
    req.body.author = req.user._id;
    const tweet = new Tweet(req.body);
    tweet.avatar = req.body.avatar;
    await tweet.save();
    res.redirect('back');
    // res.json(req.body)

  } catch (e) {
    console.log(e);
    res.redirect('/?msg=Failed to tweet')
  }
}
```

L'ultima funzione è quella di postTweet: qui viene creato il nuovo post con il contenuto testuale e l'immagine uploadata, aggiunto e salvato al database, e infine l'utente viene rediretto alla pagina iniziale.

tweetCard

```

mixin tweetCard(tweet={})
  .tweet_card.tweet_hover
    .tweet_avatar
      a(href=`/${tweet.author.username}`)
        if !tweet.author.avatar
          img(src=`/uploads/no-avatar.jpg`)
        else
          img(src=`https://res.cloudinary.com/pikbook/image/upload/${tweet.author.avatar}`)
    .tweet_text
      p.author_name
        a(href=`/${tweet.author.username}`)
          strong #{tweet.author.name || tweet.author.username}
          span @#{tweet.author.username}
      br
      p.the_tweet #{tweet.tweet}
      if tweet.avatar
        a(href=`/tweet/${tweet._id}`)
          img.the_image(src=`https://res.cloudinary.com/pikbook/image/upload/${tweet.avatar}`)
      p.tweet_actions

      p.tweet_meta
        span
          a(href=`/tweet/${tweet._id}` title=`${tweet.created}`) #{moment(tweet.created).fromNow()}
          a.download_image(href=`https://res.cloudinary.com/pikbook/image/upload/fl_attachment/${tweet.avatar}`, download='')
          i.fas.fa-download
          if user && (tweet.author.equals(user._id) || user.username == 'ale14798')
            br
            br
            span &nbsp;
            a(href=`/delete/${tweet._id}`) Delete
            span &nbsp;

```

Una volta creato il nuovo post, nella pagina di homepage, in particolare nel mixin `_tweetCard.pug`, viene visualizzato il contenuto di quest'ultimo. Nel div `tweet_text` il contenuto testuale è visualizzato tramite `{tweet.tweet}`, mentre l'immagine tramite il link al mio storage di cloudinary con l'url che termina con l'id dell'immagine del post:

```
/${tweet.avatar}
```

5. Conclusioni

5.1. Sfide Riscontrate

Le maggiori difficoltà riscontrate in questo progetto sono state:

1) Utilizzo del template engine Pug: seppur più pulito e più chiaro da leggere rispetto ad altri template, è molto carente di documentazione e spesso ho trovato molta diversità rispetto ad html e ho dovuto gestire quest'ultime grazie alla mia esperienza con altri linguaggi. Inoltre non esegue se tutto il codice non è indentato perfettamente e ciò mi ha fatto perdere molto tempo.

2) Mantenere l'applicazione veloce e integra gratuitamente: trattandosi di un social network l'applicazione dovrebbe essere sempre disponibile online, cosa che con un abbonamento free di heroku ciò non succede. Grazie però, come già detto, ad applicazioni esterne sono riuscito a mantenerla sempre online, senza aver pagato nulla.

Inoltre inizialmente le foto pubblicate e le immagini profilo risiedevano in una cartella in locale, ma deployandola su heroku ho dovuto trovare uno storage online che me le facesse avere disponibili sempre; esistono diversi storage online ma tutti richiedono un abbonamento oppure una prova gratuita con i dati della carta per il pagamento successivo. Invece grazie a Cloudinary sono riuscito ad avere un cloud storage online gratuitamente senza alcun abbonamento e a rendere le foto con la miglior qualità possibile e a comprimerle per risparmiare spazio.

Nonostante ciò queste difficoltà sono state un momento per imparare un nuovo linguaggio, per scoprire a fondo il mondo di heroku e conoscere diversi cloud storage online.

6. Nota sitografica

Risorse sitografiche usate come riferimento

- (1) <https://nodejs.org/en/>
- (2) <https://expressjs.com/>
- (3) <https://www.mongodb.com/>
- (4) <https://mongoosejs.com/>
- (5) <https://www.passportjs.org/>

(6) <https://cloudinary.com/>

Documentazione del lavoro

(1) *PikBook*: <https://pikbook.herokuapp.com/>

(2) *Github*: <https://github.com/AlessandroFare/PikBook>
