



**Università degli Studi di Camerino**

---

**SCUOLA DI SCIENZE E TECNOLOGIE**

**Corso di Laurea in Informatica (Classe L-31)**

**Editor di oggetti 3D per e-commerce  
configurabile e in realtà aumentata:  
il backend di “Customyzz”**

Laureando  
**Paolo Andreassi**

**Matricola 090629**

Relatore  
**Dott. Francesco De Angelis**

Correlatore  
**Dott. Antonio dell’Ava**

---

A.A. 2017/2018



# Indice

<b>1</b>	<b>Introduzione</b>	<b>9</b>
1.1	Progettazione . . . . .	9
1.2	Analisi . . . . .	11
<b>2</b>	<b>E-commerce</b>	<b>13</b>
2.1	Visualizzazione dei prodotti nell'e-commerce . . . . .	13
2.2	3D e realtà aumentata . . . . .	13
<b>3</b>	<b>Scelta Tecnologica</b>	<b>15</b>
3.1	Oggetti 3D . . . . .	15
3.2	JSON . . . . .	17
3.2.1	Gli array literal . . . . .	17
3.2.2	Gli object literal . . . . .	18
3.2.3	La sintassi JSON . . . . .	18
3.3	RESTful API . . . . .	19
3.4	Documentazione . . . . .	22
3.5	NodeJS . . . . .	22
3.6	Configurazione iniziale e Test . . . . .	24
3.7	Express . . . . .	27
3.8	Mongoose . . . . .	28
3.9	Heroku . . . . .	30
<b>4</b>	<b>Long Polling</b>	<b>33</b>
4.1	Implementazione del Long Polling in unicom-product-editor . . . . .	35
4.2	Altri metodi di comunicazione fra client e server . . . . .	36
4.2.1	Normale HTTP . . . . .	36
4.2.2	Polling Ajax . . . . .	36
4.2.3	Long Polling Ajax . . . . .	37
4.2.4	HTML5 Server Sent Events (SSE)/EventSource . . . . .	37
4.2.5	HTML5 Websockets . . . . .	38
<b>5</b>	<b>Implementazione</b>	<b>39</b>
5.1	US-01/03 . . . . .	39
5.2	US-04 . . . . .	42
5.3	US-05 . . . . .	43

5.4	US-06 . . . . .	45
5.5	US-07 . . . . .	47
<b>6</b>	<b>Conclusioni</b>	<b>51</b>
<b>A</b>	<b>Installazione e Uso</b>	<b>53</b>
<b>B</b>	<b>Screenshot</b>	<b>55</b>

# Elenco dei codici

3.1	Cube.obj . . . . .	16
3.2	User Collection . . . . .	20
3.3	List Collection . . . . .	21
3.4	Object Collection . . . . .	21
3.5	Part Collection . . . . .	21
3.6	Option Collection . . . . .	21
3.7	package.json . . . . .	23
3.8	test installazione express . . . . .	24
3.9	test require express . . . . .	25
3.10	test server express . . . . .	25
3.11	test hello world . . . . .	25
3.12	connessione a mongodb . . . . .	26
3.13	mongodb save . . . . .	26
3.14	app.js . . . . .	27
3.15	configurazione server . . . . .	27
3.16	configurazione server . . . . .	28
3.17	userSchema . . . . .	28
3.18	userSchema . . . . .	28
3.19	/app/schema.js . . . . .	29
4.1	HTTP POST /upload . . . . .	35
4.2	HTTP GET /insert . . . . .	35
4.3	gestione degli errori nella richiesta HTTP . . . . .	36
5.1	/routes.js . . . . .	39
5.2	/controllers/objects.js . . . . .	40
5.3	shape API . . . . .	42
5.4	/app/converted.js . . . . .	43
5.5	/public/app.js . . . . .	44
5.6	/public/controllers/lists.js . . . . .	44
5.7	/public/partials/lists.html . . . . .	45
5.8	/app/upload.js . . . . .	46
5.9	/app/convert.js . . . . .	46
5.10	/app/readjsonfile.js . . . . .	47



# Elenco delle figure

1.1	Organizzazione di uncam-product-editor su Trello . . . . .	11
1.2	Piano di realizzazione del progetto . . . . .	12
3.1	Visualizzazione grafica di cube.obj . . . . .	17
3.2	L'interfaccia utente di Postman . . . . .	22
3.3	Test di una richiesta HTTP GET . . . . .	25
3.4	MongoDB URI . . . . .	26
3.5	Inizializzazione di uncam-product-editor su Heroku . . . . .	31
3.6	Rilascio di uncam-product-editor su Heroku . . . . .	31
4.1	Modello di comunicazione Long Polling . . . . .	34
5.1	Documentazione get /objects/:id . . . . .	41
5.2	Documentazione post /objects/:id . . . . .	42
5.3	Documentazione delete /objects/:id . . . . .	42
5.4	Documentazione get /shape/:id . . . . .	43
5.5	Documentazione get / . . . . .	44
5.6	Documentazione get /lists . . . . .	45
5.7	Documentazione post /auth/login . . . . .	49
5.8	Documentazione post /auth/login sbagliato . . . . .	49





# 1. Introduzione

La qui presente tesi rappresenta la terza ed ultima parte dello stage++<sup>1</sup>, che è stato realizzato in collaborazione con il Prof. Francesco De Angelis, e-xtrategy s.r.l. e con il collega Filippo Corona. Durante lo stage ci è stato chiesto di progettare e realizzare un'applicazione per la visualizzazione e l'editing di modelli 3D da impiegare in ambito e-commerce. Il prodotto che ne è risultato ha preso il nome di "CustoMyzx".

Per poter raggiungere i numerosi obiettivi che questo progetto richiedeva, io e Filippo ci siamo divisi il carico di lavoro in maniera il più possibile equa: una metà del progetto si è focalizzata sul front end, e quindi sull'interfaccia mirata all'utente, l'altra invece sul back end, estraneo a quest'ultimo. Nella prima fase della progettazione, i due semi-progetti sono stati nominati rispettivamente "unicam-product-viewer" e "unicam-product-editor" ai fini di una distinzione tra questi. Io mi sono occupato della parte back end, e quindi di "unicam-product-editor", mentre Filippo, in modo complementare, ha sviluppato la parte front end e il relativo sotto-progetto: "unicam-product-viewer".

Quindi il progetto è composto dalle seguenti parti:

1. Il back end
2. Una web app come interfaccia per il back end
3. Il front end
4. Il configuratore 3D e in realtà aumentata sviluppato tramite web app

Tutte le parti di CustoMyzx sono state scritte in Javascript ad eccezione di uno script esterno realizzato in Python.

Questa tesi tratterà nella fattispecie i primi due punti: la realizzazione dell'applicazione web mediante l'utilizzo del MEAN Stack piuttosto che il tradizionale LAMP Stack<sup>2</sup>, e l'uso del modello architetturale nelle web app chiamato "Long Polling".

Il codice completo del progetto è disponibile nei due repository

1. [www.github.com/e-xtrategy/unicam-product-editor](https://github.com/e-xtrategy/unicam-product-editor)
2. [www.github.com/e-xtrategy/unicam-product-viewer](https://github.com/e-xtrategy/unicam-product-viewer)

## 1.1 Progettazione

La fase di progettazione si è sviluppata secondo il metodo extrategy: l'azienda basa lo sviluppo di software sulle metodologie agili, più adatte ad ambienti dinamici come lo

---

<sup>1</sup>Lo stage++ è un progetto unico che comprende l'insieme di tre moduli: stage, project work e tesi.

<sup>2</sup>Acronimo (Linux, Apache, MySQL, PHP) che indica una piattaforma software per lo sviluppo di applicazioni web

è il web, come ad esempio la formazione di team di sviluppo piccoli, cross-funzionali e auto-organizzati, lo sviluppo iterativo e incrementale, la pianificazione adattiva, e il coinvolgimento diretto e continuo del cliente nel processo di sviluppo.

In particolare, abbiamo utilizzato una via di mezzo tra Scrum e Kanban.

L'utilizzo del framework Scrum[**scrum**] consiste nel passaggio attraverso una serie di iterazioni a intervalli di tempo regolari, dette sprint, che consentono di consegnare il progetto passo dopo passo al cliente e con cadenza costante. Attraverso la giusta frequenza adottata per gli sprint, si è capaci di fare delle stime più corrette, e inoltre si possono ricevere feedback in tempo reale sul lavoro svolto. Ogni sprint si divide in quattro fasi:

1. **Pianificazione:** incontro iniziale in cui si definisce cosa fare con lo sprint corrente.
2. **Scrum giornaliero:** mini incontro giornaliero della durata di una decina di minuti per sincronizzare il team.
3. **Demo:** incontro per illustrare il lavoro svolto dal team.
4. **Retrospettiva:** una revisione di ciò che è stato fatto, esponendo anche i possibili miglioramenti attuabili.

Il Kanban[**kanban**] serve per tenere d'occhio l'avanzamento dei lavori durante gli sprint; la kanban board dunque non è altro che una lavagna atta a visualizzare e ottimizzare il flusso di lavoro del team. Un classico kanban può essere dato da una lavagna o da un'intera parete (es. e-xtrategy), ma per il nostro sviluppo software abbiamo utilizzato una board virtuale, in cui abbiamo riscontrato aspetti positivi quali tracciabilità, facilità di collaborazione, accessibilità delocalizzata.

Le funzioni di una kanban sono comunque quelle di standardizzare il flusso di lavoro e di assicurare una buona visibilità e visualizzazione del lavoro del team, per poter individuare e risolvere tempestivamente eventuali criticità. Questa metodologia inoltre è basata sulla trasparenza e sulla comunicazione, pertanto la kanban board dovrebbe essere vista come una visione oggettiva e veritiera agli occhi del team. Di norma si prevedono tre step: To Do, In Progress e Done, che noi abbiamo rinominato rispettivamente in Backlog, Sprint e Approved.

Con questa impostazione, Antonio Dell'Ava, il nostro tutor in e-xtrategy, ci ha guidato verso lo strumento più utile per tenere traccia dello sviluppo del software, implementare le user-stories e coordinare gli sviluppatori assegnati al progetto, in questo caso io e Filippo: Trello ci ha permesso di gestire il nostro piccolo team senza applicare alla lettera Kanban, ma tramite una semplice todo list.

Trello[**trello**] è un gestore di progetti basato sul web e realizzato da Fog Creek Software nel 2011. Qui il progetto è rappresentato da schede, a loro volta contenenti liste corrispondenti ad elenchi di attività. Le liste sono formate da card, che rappresentano le singole attività. Queste card si possono spostare tra le diverse liste con il metodo drag and drop, per seguire il normale flusso di sviluppo passando dalla lista "da fare" ad esempio alla lista "fatto" una volta che è stata realizzata questa parte del progetto.

Nel nostro caso ogni card rappresenta una user-story, ossia una funzionalità utile al raggiungimento di un obiettivo di business, una descrizione volutamente generica di una caratteristica che il software deve avere, e che può essere stata esplicitamente richiesta dal cliente, essere necessità o consiglio dello sviluppatore, o essere nata dalla discussione

tra i due soggetti. Ogni card può vedersi assegnata ad uno o più sviluppatori, e queste si possono organizzare, insieme alle loro schede, in raggruppamenti personalizzati. Si possono inserire inoltre commenti, allegati, date di scadenza, voti e liste di controllo in ogni card. Nel complesso, la suddivisione di un progetto in schede/liste, che a loro volta sono suddivise in card, formano un insieme di dati su misura organizzati in modo gerarchico, che rende più facile ed efficace la gestione dei progetti, ma di conseguenza anche le attività di tutta l'organizzazione.

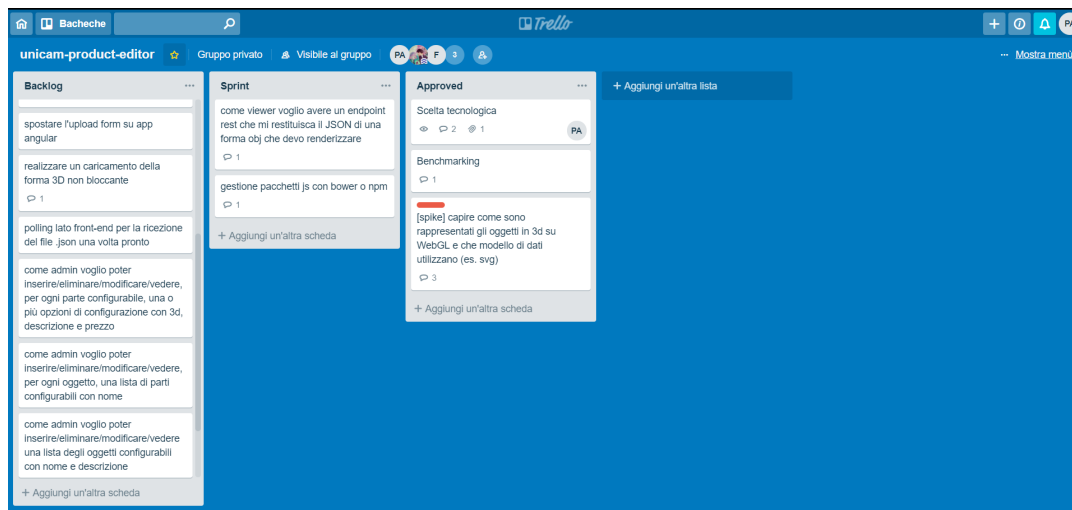


Figura 1.1: Organizzazione di uncam-product-editor su Trello

## 1.2 Analisi

Si vuole realizzare una applicazione web che permetta di gestire gli oggetti 3D tramite upload, modifica e visualizzazione in formato .JSON. Il backend deve ricevere in input una forma 3D sotto forma di file in formato .obj, convertirlo in .JSON e fornire i servizi REST. Un utente deve poter inserire l'oggetto 3D tramite l'uploader, modificare i metadati dell'oggetto convertito e visualizzare il file convertito attraverso un apposito endpoint una volta effettuata l'autenticazione. Il processo di upload e conversione in particolare non deve essere bloccante, in quanto l'utente deve poter continuare ad utilizzare il servizio anche durante l'upload di una forma 3D molto grande.

La struttura dei documenti all'interno del progetto sarà la seguente:

- Lista
  - Oggetto
    - Omino Lego
      - Parte
        - Testa
        - Busto
          - Opzione
            - Busto standard
            - Busto con papillon
            - Busto con cravatta

Le user stories derivanti da questo processo di analisi verranno identificate dalla sigla "US-XX", in cui XX rappresenta un numero progressivo, e vengono elencate di seguito:

Numero	Descrizione
US-01	Come admin voglio poter inserire/eliminare/modificare/vedere una lista degli oggetti configurabili con nome e descrizione.
US-02	Come admin voglio poter inserire/eliminare/modificare/vedere, per ogni oggetto, una lista di parti configurabili con nome.
US-03	Come admin voglio poter inserire/eliminare/modificare/vedere, per ogni parte configurabile, una o più opzioni di configurazione con 3D, descrizione e prezzo.
US-04	Come viewer voglio avere un endpoint rest che mi restituisca il JSON di una forma obj che devo renderizzare.
US-05	Creare app single-page con Angular.
US-06	Realizzare un caricamento della forma 3D non bloccante (upload, conversione ed endpoint in modo sincrono).
US-07	L'applicazione deve permettere ai nuovi utenti di registrarsi e ai vecchi utenti di effettuare un login.

Tabella 1.1: User stories

Il diagramma di Gantt riguardante l'assegnazione dei task e lo svolgimento temporale, è il seguente:

ATTIVITA'	Settimana 1	Settimana 2	Settimana 3	Settimana 4	Settimana 5	Settimana 6	Settimana 7
Analisi dei requisiti							
Scelta della tecnologia							
Implementazione della parte back end							
Implementazione della parte front end							
Test della soluzione							

Figura 1.2: Piano di realizzazione del progetto

## **2. E-commerce**

In questo capitolo si illustrano brevemente le tecnologie di visualizzazione applicate nell' e-commerce.

### **2.1 Visualizzazione dei prodotti nell'e-commerce**

Foto, grafica 360 (comunque foto), configuratori basati su fotografie

### **2.2 3D e realtà aumentata**

Il 3D e la realtà aumentata (nell'ecommerce)



## 3. Scelta Tecnologica

In questo capitolo viene trattata l'architettura utilizzata per unicam-product-editor e il modo in cui vengono gestiti i dati attraverso la web app, per poi passare ad illustrare l'organizzazione del codice.

Per sviluppare unicam-product-editor è stato utilizzato lo stack MEAN che non è altro che l'insieme dei quattro componenti che ne fanno parte:

- M = MongoDB: il popolare database
- E = Express.js: un framework web leggero
- A = Angular.js: un framework per la creazione di single page application
- N = Node.js v4: un interprete JavaScript costruito sul motore JavaScript V8 di Google Chrome orientato agli eventi che lo rende efficiente e leggero

Lo stack MEAN è un' alternativa al più tradizionale stack LAMP (Linux, Apache, MySQL, PHP/Perl/Python/P...), molto popolare per il suo utilizzo nella costruzione di applicazioni web dagli anni '90 in poi. Di seguito andiamo a costruire le RESTful API, che sono la base per qualsiasi tipo di applicazione, come un sito web, un'app Android, iOS o Ubuntu Touch.

### 3.1 Oggetti 3D

Un modello 3D è una rappresentazione matematica di un oggetto tridimensionale e consiste sostanzialmente in un file di dati strutturati, contenenti le proprietà delle primitive geometriche che costituiscono l'oggetto rappresentato (un corpo fisico). Un oggetto 3D quindi non è altro che un insieme di dati composto da punti nello spazio tridimensionale (tre dimensioni X, Y e Z) e altre informazioni (spesso metadati, o informazioni aggiuntive). I principali formati utilizzati per i modelli 3D:

- OBJ
- FBX
- Collada
- BLEND
- DFX
- STL
- PLY

- 3DS (obsoleto)
- VRML (obsoleto)
- X3D

La scelta per cui abbiamo optato per la realizzazione del nostro progetto è ricaduta sul .obj, o più correttamente Wavefront .obj, un formato di file sviluppato dalla Wavefront Technologies per il software Advanced Visualizer. Il motivo di questa scelta risiede nel fatto che il .obj è un formato aperto che è stato adottato da tantissimi applicativi per la grafica 3D per l'interscambio di dati con altri programmi; ciò lo rende perfetto per il nostro progetto, in quanto questo dovrà poi essere trasformato in un corrispondente file .JSON per poter essere interpretato dal viewer.

Questo è, per intero, il file .obj per la creazione di un cubo

```
1 # cube.obj
2 #
3
4 g cube
5
6 v 0.0 0.0 0.0
7 v 0.0 0.0 1.0
8 v 0.0 1.0 0.0
9 v 0.0 1.0 1.0
10 v 1.0 0.0 0.0
11 v 1.0 0.0 1.0
12 v 1.0 1.0 0.0
13 v 1.0 1.0 1.0
14
15 vn 0.0 0.0 1.0
16 vn 0.0 0.0 -1.0
17 vn 0.0 1.0 0.0
18 vn 0.0 -1.0 0.0
19 vn 1.0 0.0 0.0
20 vn -1.0 0.0 0.0
21
22 f 1//2 7//2 5//2
23 f 1//2 3//2 7//2
24 f 1//6 4//6 3//6
25 f 1//6 2//6 4//6
26 f 3//3 8//3 7//3
27 f 3//3 4//3 8//3
28 f 5//5 7//5 8//5
29 f 5//5 8//5 6//5
30 f 1//4 5//4 6//4
31 f 1//4 6//4 2//4
32 f 2//1 6//1 8//1
33 f 2//1 8//1 4//1
```

Codice 3.1: Cube.obj

in cui sono specificate le posizioni e le grandezze nello spazio tridimensionale dei vertici (v), dei vertici normali (vn), e delle facce (f).



L'oggetto 3D che ne deriva può essere facilmente visualizzato tramite ad esempio il programma Paint 3D:

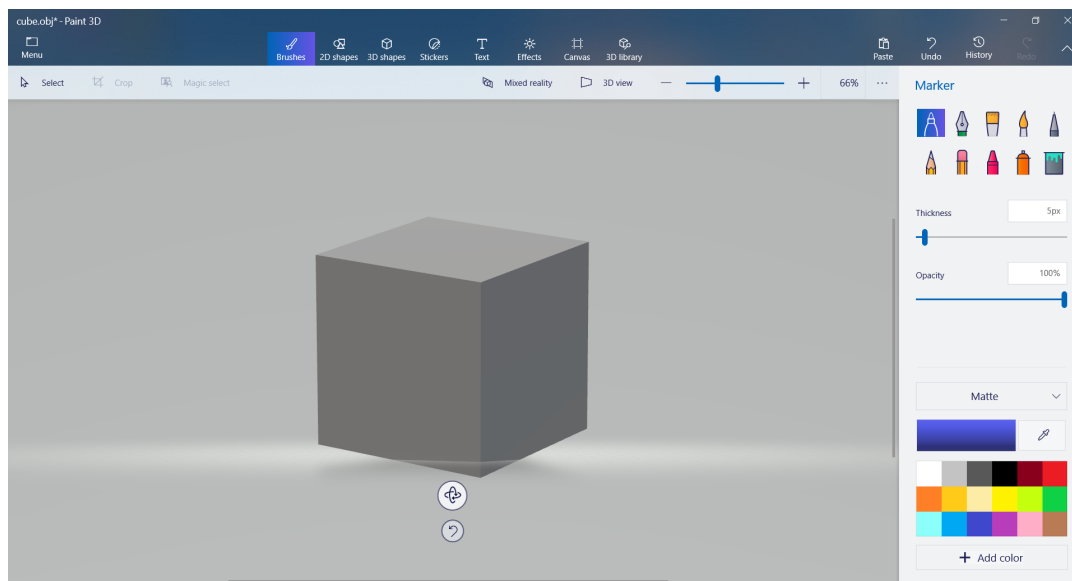


Figura 3.1: Visualizzazione grafica di cube.obj

## 3.2 JSON

JSON è un formato di dati molto leggibile basato su un sottoinsieme della sintassi JavaScript, cioè *array literal* e *object literal*, proposto da Douglas Crockford come formato per i servizi web in sostituzione al verboso XML.

Dato l'utilizzo della sintassi Javascript, le definizioni JSON possono essere incluse all'interno dei file JavaScript.

### 3.2.1 Gli array literal

Gli *array literal* rappresentano il metodo più semplice per creare un array in JavaScript. Basta infatti elencare una insieme di valori separati da una virgola all'interno delle parentesi quadre, per esempio:

```
1 var users = ["Paolo", "Antonio", "Filippo"];
```

A livello di programmazione è equivalente al codice:

```
1 var users = new Array("Paolo", "Antonio", "Filippo");
```

Entrambe le definizioni generano lo stesso risultato ed è possibile accedere ad ogni elemento dell'array utilizzando il relativo indice

```
1 console.log(users[0]); //prints Paolo
2 console.log(users[1]); //prints Antonio
3 console.log(users[2]); //prints Filippo
```

Gli array in JavaScript hanno però due caratteristiche ben precise:

1. In JavaScript l'array non ha un tipo di dati ben definito (non è tipizzato), e quindi ogni elemento può contenere tipi di dati diversi.

2. Nonostante entrambi i metodi sopra citati per la creazione di un array siano validi in JavaScript, in .JSON soltanto gli array literal verranno accettati e interpretati correttamente.

### 3.2.2 Gli object literal

Un object literal è definito tra parentesi graffe. Al suo interno troviamo un numero qualsiasi di coppie chiave-valore, definite con una stringa, i due punti ed il valore. Ogni coppia deve essere seguita da una virgola, tranne l'ultima. Questo insieme di coppie chiave-valore rappresenta, nel suo insieme, un oggetto. Un esempio:

```
1 var user = {  
2   "firstname": "Paolo",  
3   "age": 23,  
4   "student": true  
5 };
```

Il codice qui sopra genera un oggetto *user* con le proprietà *firstname*, *age* e *student*. Per accedere alle proprietà dell'oggetto basta usare la notazione con il punto:

```
1 console.log(user.firstname); \\prints "Paolo"  
2 console.log(user.age); \\prints "23"  
3 console.log(user.student); \\prints "true"
```

Lo stesso oggetto potrebbe essere creato utilizzando il costruttore *Object* di JavaScript

```
1 var user = new Object();  
2 user.firstname = "Paolo";  
3 user.age = 23;  
4 user.student = true;
```

Come già detto, sebbene entrambe le definizioni siano accettate in JavaScript, in JSON è valida solo la seconda notazione, object literal.

### 3.2.3 La sintassi JSON

La sintassi di JSON non è altro che il miscuglio di object literal ed array literal per memorizzare dati e solo e soltanto rappresentarli. Un esempio:

```
1 [  
2   {  
3     "firstname": "Paolo",  
4     "age": 23,  
5     "student": true  
6   },  
7   {  
8     "firstname": "Giuseppe",  
9     "age": 40,  
10    "student": false  
11  }  
12 ]
```

La prima cosa da notare è che in questo documento non sono presenti variabili, così come punti e virgola. In questo modo quando si trasmettono dei dati tramite HTTP ad un browser, il tutto avviene abbastanza velocemente grazie al numero ridotto di caratteri.

Oltre a questo, ci sono altri ovvi benefici nell'utilizzo di JSON come formato dati per la comunicazione in JavaScript: non bisogna preoccuparsi della valutazione dei dati, e quindi, è garantito un accesso più veloce alle informazioni che questo contiene.

### 3.3 RESTful API

Il significato dell'acronimo REST è REpresentational State Transfer, ossia una rappresentazione del trasferimento di stato di un determinato dato.

REST utilizza un modello client-server, dove il server è un server HTTP e il client invia richieste HTTP (GET, POST, PUT, DELETE), con un URL che al suo interno contiene i parametri codificati. L'URL descrive l'oggetto e il server genericamente risponde restituendo un'immagine, un documento HTML, un file CSV o qualunque altro tipo di dato. Nel nostro caso il server risponde restituendo del codice e un JSON (JavaScript Object Notation). Una RESTful API quindi è l'Interfaccia di Programmazione di un'Applicazione che usa le richieste HTTP per gestire i dati che vengono scambiati fra client e server.

Dato che in unicam-product-editor viene utilizzato lo stack MEAN, abbiamo scelto di utilizzare documenti JSON che risultano particolarmente adatti per la nostra applicazione, visto che tutti i nostri componenti sono in JavaScript e MongoDB interagisce perfettamente con questo formato. Faremo degli esempi più dettagliati più avanti quando definiremo i nostri Data Model.

Le operazioni principali che si possono eseguire con questo metodo sono le seguenti:

- GET per richiedere al server un determinato set di dati;
- POST per creare un nuovo documento all'interno del database;
- PUT per modificare o sostituire completamente un documento già esistente;
- DELETE per cancellare un documento contenuto all'interno del database al quale siamo collegati.

Queste operazioni vengono spesso descritte attraverso l'acronimo CRUD, che sta per CREATE, READ, UPDATE e DELETE. Il server HTTP, da parte sua, usa spesso assieme alle API REST dei codici di risposta; di seguito vengono elencati alcuni fra i più comuni:

- 200 - "OK"
- 201 - "Created" (Utilizzato con POST)
- 400 - "Bad Request" (Ad esempio per l'assenza di parametri)
- 401 - "Unauthorized" (Non ci sono i parametri per l'autenticazione)
- 403 - "Forbidden" (L'utente è autenticato ma non ha i permessi)
- 404 - "Not Found"

Una descrizione completa è contenuta nell'apposito RFC 2616[**RFC2616**].

Lo sviluppo di API REST è alla base del nostro sviluppo, in quanto permette alla nostra applicazione di essere multiplatforma. Per fare alcuni esempi di applicazioni di questo tipo si possono citare Google Docs, Pixlr o lo stesso Trello insieme ad Evernote. Le API REST permettono la facile implementazione dell'applicazione su molte piattaforme che potranno essere sviluppate in un secondo momento, trasformando il progetto iniziale in un progetto indipendente dalla piattaforma di partenza.

I dati da gestire nel nostro progetto si possono riassumere in questo modo:

- Dati degli utenti
- Liste
- Oggetti
- Opzioni di ogni oggetto
- Parti di ogni opzione

Come potremo vedere a breve, diversamente dai tradizionali Database relazionali (in cui la forma principale di struttura dati è data dalle Tabelle), in MongoDB, essendo un Database NoSQL, e quindi non relazionale, non si parla di Tabelle, anche se il concetto di Database ad alto livello è lo stesso.

Nel Database MongoDB possono essere contenute più Collection, che possono forse essere paragonate alle Tabelle nei Database relazionali. A sua volta, ogni Collection conterrà uno o più Document, che a confronto con un Database relazionale corrisponde all'incirca alla riga di una Tabella. Ogni Document (così come ogni Collection) non segue però uno schema specifico come in una Tabella, ma può essere formato da una o più coppie chiave-valore, dove il contenuto può essere a sua volta una variabile semplice oppure qualcosa di più complicato come un array.

Il documento JSON appena preso in esempio rappresenta un utente nel nostro sistema. Ogni campo ha il suo specifico scopo, ma il campo più importante in un Document MongoDB è senza dubbio il campo `_id`: questo rappresenta la chiave primaria di ogni Document. Quando un Document viene salvato senza questo campo, MongoDB ne assegna uno automaticamente, e con un valore univoco.

Ora passiamo ai vari Document che compongono le nostre Collection; la Collection degli Utenti conterrà Documenti di questo tipo:

```
1 {
2   "_id": {
3     "$oid": "5894fa05f882a91c7c5cf0e6"
4   },
5   "displayName": "Paolo_Andreassi",
6   "email": "pablo15941@gmail.com",
7   "password": "$2a$10$kyajc1WP9KFvSVWzrPagLe.vgY76FjBQjwt76.HhC5u9oo96qoRo.",
8   "__v": 0,
9   "picture": "https://graph.facebook.com/v2.3/10209758034881202/...",
10  "google": "106815394034042298036"
11 }
```

Codice 3.2: User Collection

Si può notare l'importante presenza del campo `_id`, così come gli altri dati utili al riconoscimento dell'utente. La password inoltre in unicam-product-editor non viene mai mostrata in chiaro all'interno del Database, anche se il campo ad essa corrispondente è presente e popolato. Questo è il risultato di una password impostata dall'utente ma successivamente encriptata con una funzione hash.

Il Document di tipo Lista ha uno scopo puramente descrittivo nei confronti di ciò che contiene:

```

1 {
2   "_id": {
3     "$oid": "589a30d1b2a4c717a8da16b3"
4   },
5   "nome": "Omino_LEGO",
6   "__v": 0
7 }
```

Codice 3.3: List Collection

La Collection degli Oggetti avrà dei Document in cui i campi nome e descrizione hanno una funzione descrittiva, ma oltre al campo `_id` è presente il campo `_list` che funge da collegamento con la Collection delle Liste: in questo campo infatti è indicato l'identificativo della Lista a cui appartiene uno specifico Oggetto:

```

1 {
2   "_id": {
3     "$oid": "589a38f93ec83413e4f010b2"
4   },
5   "nome": "Omino_LEGO",
6   "descrizione": "LEGO_figure",
7   "_list": [
8     "589a30d1b2a4c717a8da16b3"
9   ],
10  "__v": 0
11 }
```

Codice 3.4: Object Collection

Essendo un Database con una struttura nidificata, anche i Document nelle Collection delle Parti e delle Opzioni avranno una struttura simile a quelli delle Collection degli Oggetti: anche in queste infatti è indicato il campo del Documento *parente* a cui fanno riferimento:

```

1 {
2   "_id": {
3     "$oid": "589a3a2857ba982460b83fb7"
4   },
5   "nome": "Testa",
6   "_object": [
7     "589a38f93ec83413e4f010b2"
8   ],
9   "__v": 0
10 }
```

Codice 3.5: Part Collection

La differenza più importante risiede nei Document che compongono la Collection delle Opzioni: in questi vengono indicati anche il campo `prezzo`, ma soprattutto il campo `forma`, che contiene una parte di URL con cui effettuare una chiamata REST API. Il campo `forma`, che viene mostrato all'utente nell'interfaccia della applicazione web, permette di visualizzare il file JSON relativo ad una forma 3D inserita tramite l'uploader e successivamente convertita nel formato usato da MongoDB e JavaScript:

```

1 {
2   "_id": {
3     "$oid": "589a3c5938319d1c8cb4e787"
4   },
5   "nome": "Testa_Yoda",
6   "prezzo": 50,
7   "forma": "/shape/58593a6319290c09984a439b",
8   "_part": [
```

```
9 "589a3a2857ba982460b83fb7"  
10 ],  
11 "__v": 0  
12 }
```

Codice 3.6: Option Collection

Infine la Collection delle forme 3D convertite contiene l'insieme dei file JSON che rappresentano le stesse forme.

## 3.4 Documentazione

Una parte fondamentale della progettazione di API REST è la documentazione. Per scrivere la documentazione necessaria è stato usato lo strumento Postman.

Postman è un ambiente di sviluppo multiplatforma utile per testare le API, ossia per eseguire una qualsiasi chiamata HTTP al nostro server, e monitorare la risposta ottenuta in modo chiaro e leggibile. Tramite questo strumento è possibile effettuare delle chiamate API senza dover mettere mano al codice dell'applicazione, consentendo di effettuare le chiamate tramite questo plugin che fornisce un'utile interfaccia grafica. Le richieste possono essere effettuate sia verso un server locale che verso un server online impostando tutti i dati di una tipica chiamata API, dagli headers al body.

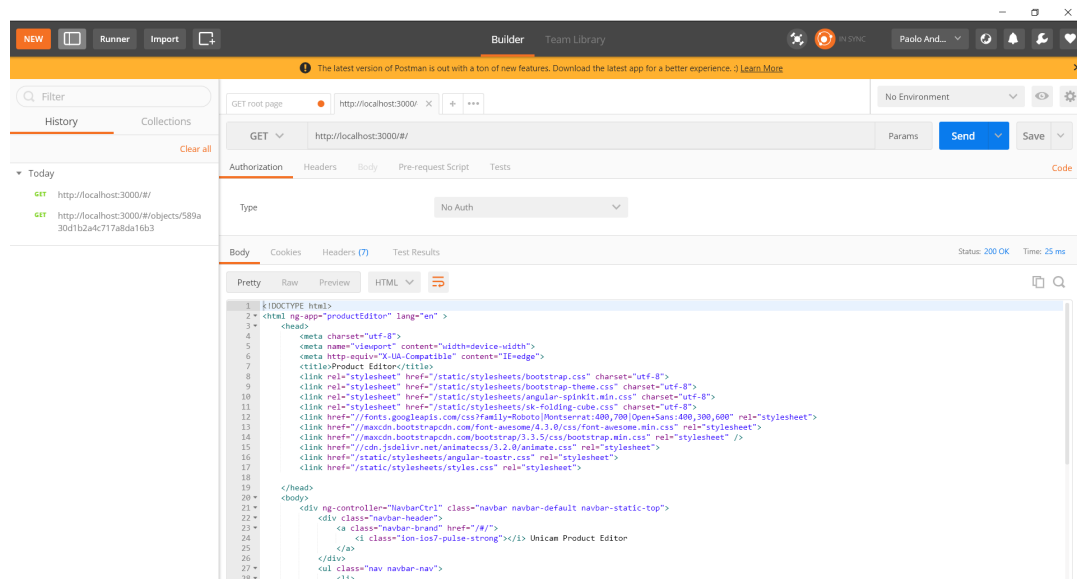


Figura 3.2: L'interfaccia utente di Postman

## 3.5 NodeJS

Node.js[**node**] v4 è un interprete JavaScript orientato agli eventi, progettato per realizzare applicazioni di rete scalabili. La caratteristica principale è che rimane inattivo finché non arriva una connessione in ingresso, o più generalmente un qualsiasi evento, e viene chiamata la relativa callback. Questo paradigma è in contrasto con quello tradizionale basato sui thread. Il principale vantaggio del paradigma ad eventi rispetto a quello basato sui thread si ha quando c'è un alto traffico. In questo caso, infatti, l'alto numero di thread genera un alto overhead diminuendo drasticamente le prestazioni

della macchina host. Inoltre, dato che nessuna funzione di Node eseguirà direttamente operazioni di I/O, il processo non si bloccherà mai.

La versione di Node usata in unicam-product-editor è la v4.6.0. Se da un lato su un server è normale avere solo una versione di Node, potrebbe non esserlo su una macchina di sviluppo. Può capitare, infatti, che uno sviluppatore sia impegnato su più progetti Node che utilizzano versioni differenti. Per risolvere questo problema possiamo installare NVM. Node Version Manager permette di avere più versioni di Node contemporaneamente nel nostro sistema e fornisce una semplice CLI per poterle gestire. In particolare, basta creare un file di testo chiamato `.nvmrc` in cui salvare il numero di versione di Node che si intende usare per la cartella corrente. Poi, sarà sufficiente digitare il comando `nvm install` per scaricare ed utilizzare la giusta versione.

Node è disponibile per diverse piattaforme, come Linux, Microsoft Windows e Apple OS X. Le applicazioni Node.js vengono costruite utilizzando librerie e moduli disponibili per l'ecosistema ed alcuni di essi sono stati utilizzati in unicam-product-editor. Per iniziare ad utilizzare Node.js, dobbiamo per prima cosa creare un file `package.json`, che descrive la nostra applicazione ed elenca tutte le sue dipendenze. NPM (Node.js Package Manager) installa una copia delle librerie in una sotto-cartella chiamata `node_modules/`, della cartella principale dell'applicazione. Questo comporta diversi benefici, ad esempio isola le diverse versioni delle librerie, evitando così i problemi di compatibilità che si sarebbero presentati se avessimo installato tutto in una cartella standard come `/usr/lib`.

Il comando `npm install` creerà la cartella `node_modules/`, con dentro tutte le librerie richieste e specificate nel file `package.json`:

```

1 {
2   "name": "unicam-product-editor",
3   "version": "1.0.0",
4   "private": "true",
5   "description": "Editor di prodotti 3D",
6   "repository": "https://github.com/e-xtrategy/unicam-product-editor",
7   "main": "app.js",
8   "scripts": {
9     "test": "nodemon --ignore tmp/_app.js",
10    "start": "node app.js"
11  },
12  "author": "Paolo Andreassi",
13  "license": "ISC",
14  "dependencies": {
15    ...
16    "async": "^2.1.4",
17    "bcryptjs": "^2.4.0",
18    "body-parser": "^1.15.2",
19    ...
20    "cors": "^2.8.1",
21    "express": "^4.14.0",
22    "express-fileupload": "0.0.5",
23    "express-handlebars": "^3.0.0",
24    ...
25    "mongodb": "^2.2.10",
26    "mongoose": "^4.7.8",
27    ...
28    "python-shell": "^0.4.0",
29    ...
30    "satellizer": "^0.15.5",
31    ...
32  },
33  "devDependencies": {
34    "nodemon": "^1.11.0",
35    "should": "~7.1.0",
36    "supertest": "^1.1.0"
37  },

```

```
38     "engines": {  
39         "node": "4.6.0"  
40     }  
41 }
```

Codice 3.7: package.json

E' interessante analizzare alcune parti di questo file, oltre al fatto che anche questo è in formato JSON, e oltre alle prime righe che descrivono il progetto in sé per sé. Ci sono due script: quello chiamato *start* avvia l'applicazione normalmente tramite il comando *node* seguito dal nome del file principale del server *app.js*. Lo script chiamato *test* invece è interessante, perché avvia uno strumento (installato tramite l'apposita dipendenza) chiamato *nodemon*. Questo utilissimo *demone* fa sì che in fase di test ogni qual volta si verifichi un errore che interrompe l'esecuzione del server *nodemon* attende un salvataggio dei file del progetto, che indica che lo sviluppatore ha modificato il codice per correggere l'errore, e non appena questo evento si verifica il demone riavvia in automatico il server ricominciando ad eseguire il codice.

Un'altra dipendenza interessante riguarda *python-shell*, che nel nostro progetto servirà ad eseguire lo script della libreria *Three.js* scritto in codice Python (altrimenti non eseguibile da Javascript), che si occupa di convertire il file *.obj* in JSON. *python-shell* può ricevere in input interi file scritti in Python per eseguirli in modo asincrono.

### 3.6 Configurazione iniziale e Test

Andiamo ora ad analizzare l'inizializzazione di un server Node.js, la configurazione di MongoDB su Mongolab e l'interazione di questi attraverso Express e Mongoose, che sono elementi che vedremo comunque più tardi.

Il primo comando che si utilizza per inizializzare il progetto e per creare il relativo file *package.json* è *npm init*. Il comando va lanciato dalla root del progetto, che sarà collegata al repository su GitHub. Il prossimo passo consiste nel creare il file JavaScript del server in sé per sé. Questo file in *unicam-product-editor* si chiama *app.js*, ed è contenuto anch'esso nella root del progetto. Per eseguire già da ora il server, basterà posizionarsi con il prompt dei comandi nella root del progetto e lanciare il comando *node* seguito dal nome del file del server appena creato, quindi *node app.js*.

Si passa ora ad importare il framework *express* tramite la sua dipendenza e usando il gestore di pacchetti NPM. Per fare ciò dal prompt dei comandi basterà eseguire lo script *npm install express --save*. L'opzione *--save* serve a far comparire la dipendenza appena installata nell'apposita sezione del file *package.json*. Dalla versione 5.0.0 di NPM comunque questo comando è implicito. Alla fine dell'esecuzione dello script, avremo la dipendenza installata localmente nella cartella *node\_modules/*, e il file *package.json* apparirà come segue:

```
1 {  
2     "name": "node_server",  
3     "version": "1.0.0",  
4     "main": "app.js",  
5     "author": "Paolo Andreassi",  
6     "license": "ISC",  
7     "dependencies": {  
8         ...  
9         "express": "^4.14.0",  
10        ...  
11    }  
}
```



12

}

## Codice 3.8: test installazione express

Di conseguenza, andremo ad utilizzare express nel nostro file `app.js` come segue:

```
1 const express = require('express');
2 const app = express();
```

## Codice 3.9: test require express

e lo utilizzeremo subito per creare il nostro server in ascolto sulla porta 3000 (almeno per questo test) tramite l'apposita istruzione

```
1 app.listen(3000, function() {
2   console.log('listening_on_port_3000')
3 })
```

## Codice 3.10: test server express

In express possiamo gestire le richieste HTTP GET attraverso il metodo `get`:

```
1 app.get(path, callback)
```

che useremo così per la nostra configurazione di test

```
1 app.get('/', function(req, res) {
2   res.send('Hello_World')
3 })
```

## Codice 3.11: test hello world

In questo modo aprendo il nostro browser e digitando `localhost:3000` nella barra URL, dopo esserci assicurati di aver avviato l'applicazione, quello che il nostro server risponderà alla richiesta HTTP GET sarà:



Figura 3.3: Test di una richiesta HTTP GET

Una volta configurato il server per questo test, andiamo a fare lo stesso per il nostro database MongoDB. In unicom-product-editor il database è ospitato in cloud attraverso il servizio mLab, che a sua volta si appoggia a servizi cloud quali Amazon Web Services, Google Cloud Platform e Windows Azure. mLab offre diversi piani tariffari, ma per il nostro progetto il piano Sandbox che offre 0.5GB di spazio gratuito va più che bene. Una volta creato il database su mLab, e dopo aver creato un utente di tipo amministratore per lo stesso database, mLab creerà una indirizzo URL, detto MongoDB URI, utilizzabile per connettersi a questo servizio, mediante l'uso del nome utente e della password dell'account amministratore.

La stringa corrispondente al database di unicom-product-editor è la seguente:

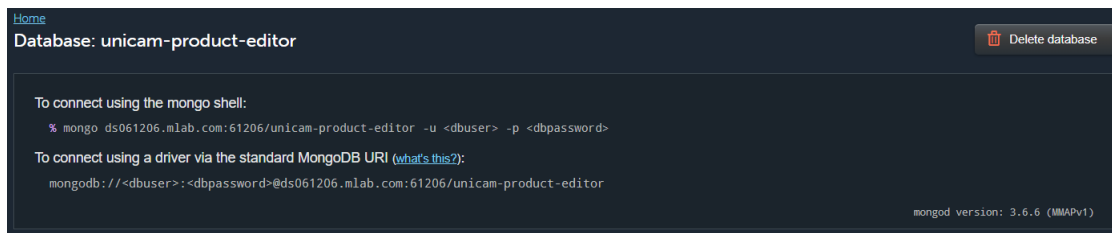


Figura 3.4: MongoDB URI

Come si può notare dalla schermata qui sopra, oltre ad essere indicata la versione del database in uso, è presente anche il comando da utilizzare per connettersi a mLab tramite il mongo shell, ossia un'interfaccia interattiva di MongoDB, che usa anch'essa istruzioni in JavaScript. Il prossimo step consiste nell'utilizzare proprio questo MongoDB URI per connettersi al database. In questo caso di test useremo il client nativo mongodb, mentre in unicom-product-editor verrà utilizzata la libreria Mongoose, che vedremo più tardi.

Per installare il client mongodb basta lanciare il comando `npm install mongodb`, e per poterlo usare nel nostro progetto bisognerà importarlo come già fatto per Express, ossia tramite l'istruzione

```
1 const MongoClient = require('mongodb').MongoClient
```

L'istruzione standard in Express per la connessione al database tramite il client mongodb è la seguente:

```
1 MongoClient.connect('MONGODB_URI', function(err, client){
2   ...
3 })
```

e utilizzeremo la callback della funzione `.connect` per avviare il server Node.js solamente una volta che sarà avvenuta la corretta connessione al database. Per fare ciò basta aggiungere il codice

```
1 MongoClient.connect('MONGODB_URI', function(err, client){
2   if (err) return console.log(err)
3   db = client.db('unicam-product-editor')
4   app.listen(3000, function() {
5     console.log('listening_on_port_3000')
6   })
7 })
```

Codice 3.12: connessione a mongodb

utilizzando la variabile `db` per salvare la sessione di collegamento al database appena ottenuta.

A questo punto la connessione al nostro database è pronta all'uso, e possiamo utilizzare la funzione `.collection` per gestire le Collection presenti nel database. Se volessimo ad esempio inserire un Document in una Collection del database unicom-product-editor, la sintassi sarebbe la seguente:

```
1 db.collection('COLLECTION').save('DOCUMENT', function(err, result){
2   if (err) return console.log(err) //in caso di errore
3
4   console.log('saved_to_database') //in caso di corretto salvataggio
5 })
```

Codice 3.13: mongodb save

### 3.7 Express

Con `express.js`, noi abbiamo creato la vera e propria "applicazione", che ascolta le richieste HTTP in ingresso su una data porta. Quando arriva una richiesta, questa passa attraverso una catena di intermediari, detti *middleware*. Ogni nodo della catena è definito da un `req` (request) object, usato per ricevere dei parametri, e un `res` (results) object, usato per salvare i risultati. Ogni nodo può decidere se eseguire delle operazioni, o se passare i due oggetti al nodo successivo. Per aggiungere un nuovo middleware usiamo la funzione `app.use()`. Il middleware principale, chiamato "router", analizza l'URL e il tipo di richiesta per poterla poi indirizzare verso una specifica funzione.

Il codice della nostra applicazione apparirà così organizzato, visto che i vari handler e le diverse funzioni specifiche saranno poi distribuiti in file separati.

```

1 var path = require('path'),
2   qs = require('querystring'),
3
4   async = require('async'),
5   bcrypt = require('bcryptjs'),
6   bodyParser = require('body-parser'),
7   colors = require('colors'),
8   cors = require('cors'),
9   express = require('express'),
10  exphbs = require('express-handlebars'),
11  favicon = require('serve-favicon'),
12  fileUpload = require('express-fileupload'),
13  fs = require('fs'),
14  http = require('http'),
15  logger = require('morgan'),
16  jwt = require('jwt-simple'),
17  moment = require('moment'),
18  mongoose = require('mongoose'),
19  request = require('request'),
20  url = require('url'),
21
22  upload = require('./app/upload'),
23  convert = require('./app/convert'),
24  converted = require('./app/converted'),
25  readjsonfile = require('./app/readjsonfile'),
26  users = require('./routes/users'),
27  getshapes = require('./app/getshapes'),
28  config = require('./config');
29
30 var app = express();
31 app.set('port', process.env.NODE_PORT || 3000);
32 app.set('host', process.env.NODE_IP || 'localhost');
33 app.set('view_engine', 'handlebars');
34 app.engine('handlebars', exphbs({defaultLayout: 'main'}));
35 app.use(bodyParser.json());
36 app.use('/static', express.static('public'));
37 app.use(express.static('partials'));
38 app.use(fileUpload());
39 require('./app/schema'),
40 require('./routes')(app);

```

Codice 3.14: `app.js`

e la route principale del progetto sarà così dichiarata, essendo Handlebars il motore di rendering utilizzato in `unicam-product-editor`:

```

1 app.get('/', function(req, res) {
2   res.render('layouts/main.handlebars');
3 })

```

Codice 3.15: configurazione server

Handlebars.js è un'estensione del linguaggio di template *Mustache*, creato da Chris Wanstrath. Sia Handlebars.js che Mustache sono linguaggi di template *logicless*, che tengono separate, come da convenzione, la parte visiva del progetto e la parte di codice, migliorando l'organizzazione del progetto stesso.

Per ultimo definiamo il nostro server in ascolto sulla porta indicata con la funzione Express `app.set(port)`, o sulla porta 3000 in tutti gli altri casi.

```
1 server = app.listen(process.env.PORT || 3000, function() {
2   var port = server.address().port
3   console.log("Express_server_listening_on_port_%s", port)
4 })
```

Codice 3.16: configurazione server

## 3.8 Mongoose

A questo punto, i nostri dati vanno gestiti tramite l'uso di Mongoose, un modellatore di oggetti MongoDB progettato per l'uso in Node.js. Come detto in precedenza, le Collection da definire sono le seguenti:

- Users
- Lists
- Objects
- Parts
- Shapes

Per fare ciò, bisogna definire uno schema per ognuna delle Collection appena espote; partendo dal primo, lo UserSchema sarà così dichiarato:

```
1 var userSchema = new mongoose.Schema({
2   email: { type: String, unique: true, lowercase: true },
3   password: { type: String, select: false },
4   displayName: String,
5   picture: String,
6   facebook: String,
7   google: String
8 });
```

Codice 3.17: userSchema

come possiamo notare, con mongoose e MongoDB è possibile non solo dichiarare il tipo di dato che ogni proprietà conterrà, ma anche dei vincoli attraverso l'uso dei trim. Inoltre con mongoose è possibile definire, all'interno dello userSchema, funzioni e metodi che potremo all'occorrenza richiamare da altre parti del progetto.

```
1 userSchema.pre('save', function(next) {
2   var user = this;
3   if (!user.isModified('password')) {
4     return next();
5   }
6   bcrypt.genSalt(10, function(err, salt) {
7     bcrypt.hash(user.password, salt, function(err, hash) {
8       user.password = hash;
9       next();
10    });
11  });
```

```

12 });
13
14 userSchema.methods.comparePassword = function(password, done) {
15     bcrypt.compare(password, this.password, function(err, isMatch) {
16         done(err, isMatch);
17     });
18 };

```

Codice 3.18: userSchema

La funzione `userSchema.pre` viene eseguita appena prima dell'inserimento di un nuovo utente (tramite la funzione `save`) e si assicura di encriptare nuovamente la password di un utente ogni qual volta questa venga cambiata, prima di inserirla nel database. Il metodo `.comparePassword` invece si avvale dell'uso della libreria *bcrypt* per decriptare la password passata al metodo come parametro prima di confrontarla con la password presente nel database, anch'essa opportunamente decriptata.

Gli altri schema sono stati dichiarati nell'apposito file `schema.js`, ad eccezione della `Collection Shape` che non ha bisogno di un vero e proprio schema, e appaiono in questo modo:

```

1 var mongoose = require('mongoose'),
2   Schema = mongoose.Schema;
3
4 var lists = Schema({
5   nome      : String
6 });
7
8 var objects = Schema({
9   _list     : [{type: String, ref: 'List'}],
10  nome      : String,
11  descrizione : String,
12  parts     : [{type: Schema.Types.ObjectId, ref: 'Part'}]
13 });
14
15 var parts = Schema({
16   _object   : [{type: String, ref: 'Object'}],
17   nome      : String,
18   options   : [{type: Schema.Types.ObjectId, ref: 'Option'}]
19 });
20
21 var options = Schema({
22   _part     : [{type: String, ref: 'Part'}],
23   nome      : String,
24   prezzo    : Number,
25   forma     : String
26 });
27
28 module.exports = mongoose.model('Option', options);
29 module.exports = mongoose.model('Part', parts);
30 module.exports = mongoose.model('Object', objects);
31 module.exports = mongoose.model('List', lists);

```

Codice 3.19: /app/schema.js

E' interessante notare come i vari schema si possono collegare tra loro attraverso l'uso del trim `ref`, che punta al nome di uno schema esportato. Nel caso di `unicam-product-editor`, che ha una struttura dati nidificata, questo trim è di cruciale importanza.

In `mongoose` alcuni dei principali metodi usati per l'interfacciamento con il database sono:

- `find()` per trovare tutti i `Document` all'interno di una `Collection`;
- `findById()` per trovare un `Document` a seconda del suo attributo `_id`;

- `findOne()` per trovare il primo Document che rispetti le condizioni dichiarate;
- `deleteOne()` per eliminare il primo Document che rispetti le condizioni dichiarate;
- `updateOne()` per aggiornare il primo Document che rispetti le condizioni dichiarate;
- `replaceOne()` simile a `updateOne()`, con la differenza che mongoose rimpiazzerà il Document esistente con il Document che si darà in input.

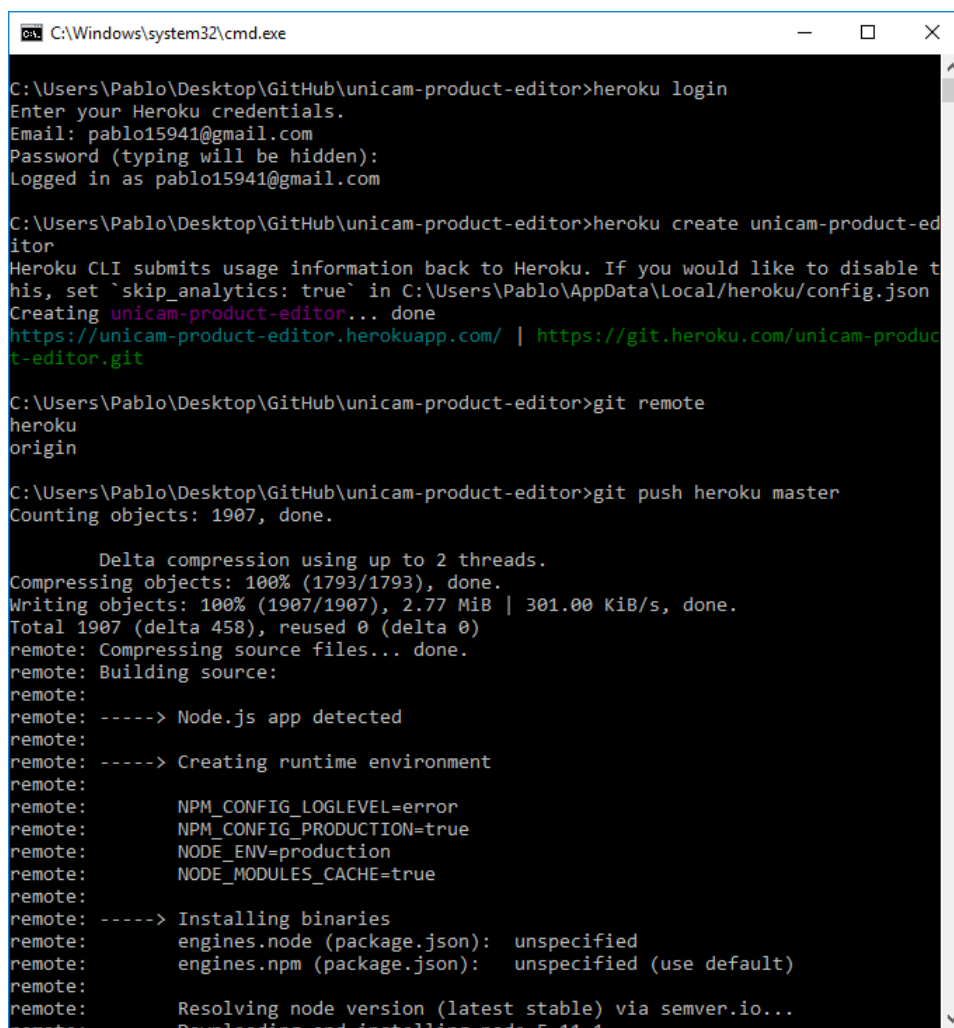
## 3.9 Heroku

Heroku è un platform as a service (PaaS) sul cloud che supporta linguaggi di programmazione come JavaScript(quindi Node.js), Ruby, Python, Java, PHP ed altri. Il fatto che sia un PaaS vuol dire che Heroku mette a disposizione di chi sviluppa una piattaforma dove poter rilasciare rapidamente una applicazione senza dover conoscere come configurare l'infrastruttura che c'è dietro. Nel nostro caso, Heroku ci è utile per rilasciare `unicam-product-editor` come web app su un servizio apposito.

Per la configurazione e il rilascio della applicazione su Heroku, bisogna prima di tutto installare il suo CLI (Command Line Interface) e dopo essersi autenticati al servizio Heroku si procede a creare un clone del repository di `unicam-product-editor` presente su GitHub attraverso il comando `git clone 'git repository'`:

```
1 $ git clone https://github.com/e-xtrategy/unicam-product-editor.git
```

A questo punto, non resta che creare una nuova *commit*, ma invece che eseguire un *push* sul repository originale, si fa lo stesso utilizzando il comando `git push heroku master`. Alla fine dell'operazione, in caso di successo, il CLI mostra l'URL su cui è ospitata la nostra applicazione. Di seguito uno screenshot delle operazioni appena descritte eseguite in sequenza.



```

C:\Windows\system32\cmd.exe

C:\Users\Pablo\Desktop\GitHub\unicam-product-editor>heroku login
Enter your Heroku credentials.
Email: pablo15941@gmail.com
Password (typing will be hidden):
Logged in as pablo15941@gmail.com

C:\Users\Pablo\Desktop\GitHub\unicam-product-editor>heroku create unicam-product-editor
Heroku CLI submits usage information back to Heroku. If you would like to disable this, set `skip_analytics: true` in C:\Users\Pablo\AppData\Local\heroku\config.json
Creating unicam-product-editor... done
https://unicam-product-editor.herokuapp.com/ | https://git.heroku.com/unicam-product-editor.git

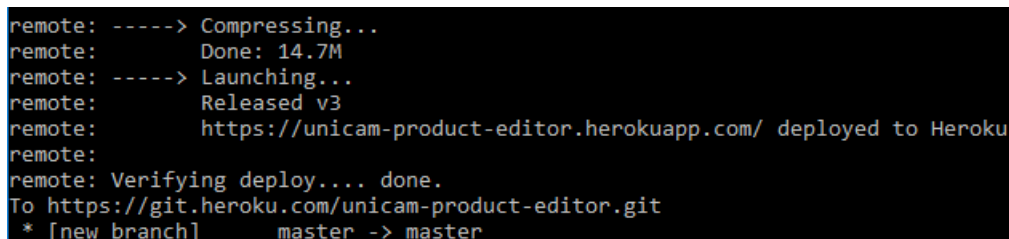
C:\Users\Pablo\Desktop\GitHub\unicam-product-editor>git remote
heroku
origin

C:\Users\Pablo\Desktop\GitHub\unicam-product-editor>git push heroku master
Counting objects: 1907, done.

Delta compression using up to 2 threads.
Compressing objects: 100% (1793/1793), done.
Writing objects: 100% (1907/1907), 2.77 MiB | 301.00 KiB/s, done.
Total 1907 (delta 458), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Node.js app detected
remote:
remote: -----> Creating runtime environment
remote:
remote:           NPM_CONFIG_LOGLEVEL=error
remote:           NPM_CONFIG_PRODUCTION=true
remote:           NODE_ENV=production
remote:           NODE_MODULES_CACHE=true
remote:
remote: -----> Installing binaries
remote:           engines.node (package.json):  unspecified
remote:           engines.npm (package.json):   unspecified (use default)
remote:
remote:           Resolving node version (latest stable) via semver.io...
remote:           Downloading and installing node 5.11.1

```

Figura 3.5: Inizializzazione di unicam-product-editor su Heroku



```

remote: -----> Compressing...
remote:           Done: 14.7M
remote: -----> Launching...
remote:           Released v3
remote:           https://unicam-product-editor.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy.... done.
To https://git.heroku.com/unicam-product-editor.git
 * [new branch]      master -> master

```

Figura 3.6: Rilascio di unicam-product-editor su Heroku





## 4. Long Polling

In questo capitolo si spiega il funzionamento del modello di comunicazione Long Polling, per poi spiegare come è applicato in unicam-product-editor, ed infine compararlo con altre metodologie in uso.

L'XMLHttpRequest Long Polling è, di base, una tecnologia dove il client richiede un'informazione al server senza aspettarsi una risposta immediata. Questo fa sì che la connessione che esiste fra client e server abbia una connessione di lunga durata, durante la quale il server esegue un'operazione complessa, e solo una volta portata a termine, restituisce una risposta al client inviando i dati richiesti come risposta, o anche solo una notifica.

Innanzitutto il fatto che questa tecnologia sia basata su XMLHttpRequest ci fa capire che le chiamate che il client effettua nei confronti del server sono delle semplici chiamate Ajax, ma a differenza dell'uso tradizionale di queste chiamate, la gestione degli eventi non viene eseguita dalla parte del client (client-side), ma dalla parte del server (server-side). Inoltre, a differenza delle chiamate Ajax, che vengono effettuate a intervalli regolari, ad esempio ogni 10 secondi al termine delle quali la connessione al server viene chiusa, in una chiamata di tipo Long Polling, la connessione col server rimane aperta finché il server stesso non invia una risposta, oppure finché non si raggiunge un limite di tempo impostato, detto timeout.

Questo tipo di tecnologia usato nelle applicazioni non è nuovo, le web chat sono sempre esistite. Quello che è cambiato negli ultimi anni è l'approccio tecnologico a basso livello. Una volta i client facevano richieste HTTP ogni pochi secondi al server richiedendo eventuali messaggi (come ad esempio nel protocollo POP3 per la ricezione delle email) mentre nel Long Polling è il server stesso a notificare il client solamente a fronte di novità.

Di seguito uno schema che riassume l'architettura Long Polling, mostrando l'interazione fra un client e un server, e le relative richieste e risposte.

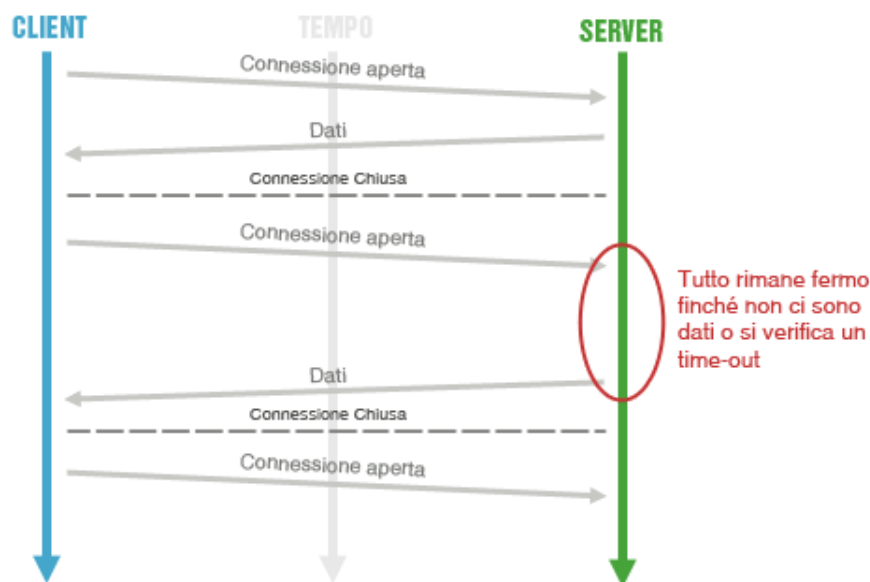


Figura 4.1: Modello di comunicazione Long Polling

Questo metodo, assieme ad altri con lo stesso funzionamento di base (ad esempio lo Streaming) insieme costituiscono il modello architetturale Comet, in cui le richieste HTTP di lunga durata che permettono ad un web server di inviare dati ad un browser senza che quest'ultimo li abbia esplicitamente richiesti ne sono la caratteristica principale.

La tecnica del Long Polling, nonostante non sia difficilissima da implementare una volta compresa, non può essere però eseguita in qualsiasi server web proprio per la caratteristica di avere richieste in stato "pending". I principali server web e application server non offrono questa funzionalità nativa perchè agiscono, diversamente da Node.js, in maniera sincrona. Hanno a disposizione un numero finito di "slot di richieste", esaurito il quale sono obbligati a rilasciarne qualcuna. NodeJS grazie al suo modello *event-driven* offre una struttura di basso livello che eccelle in questo tipo di applicazioni. Il meccanismo basato su callback rappresenta il *deus ex machina* per un'architettura basata su Long Polling.

In unicam-product-editor, e più precisamente nella fase di upload di un oggetto 3D, il caso base che si verificherà sarà quello del client che inizia il caricamento della forma attraverso l'interfaccia, inviando una richiesta HTTP al server Node.js, il quale inizierà in sequenza le fasi di upload, conversione in JSON e inserimento del file convertito nel database. Durante questo processo, che per sua natura ha un tempo di esecuzione lungo (essendo i file 3D in formato .obj ed i corrispondenti file JSON una dimensione media nell'ordine dei MegaByte) la richiesta HTTP rimane in una fase di congelamento, finché il server non avrà eseguito tutto il processo, ed invierà al client una risposta (sia essa di successo nell'esecuzione o di errore), o finché non si sarà raggiunto il tempo limite, detto timeout, superato il quale la richiesta si esaurirà.

Il Long Polling HTTP inoltre è utilissimo per creare API affidabili, in quanto le azioni

di sincronizzazione e di ascolto possono essere combinate in una stessa richiesta. Nel nostro caso, essendo questo progetto basato su RESTful API, è facile espandere una di queste trasformandola in una Long Polling API, mantenendo comunque la stessa semantica, usando anche questo sistema delle interazioni di tipo *request/response*. Dei timeout brevi possono oltretutto aumentare la solidità delle richieste fra client e server nel caso in cui, ad esempio, l'indirizzo IP di un client cambi come conseguenza del roaming da rete wireless a rete mobile o tethering.

E' per questi motivi che in unicom-product-editor è stata implementata la tecnologia di Long Polling nella fase di upload di una forma 3D. Il tutto rende il processo di upload non bloccante, ossia permette all'utente, durante l'esecuzione delle operazioni sopra elencate, di continuare ad eseguire altre operazioni sulla applicazione web, mentre il processo viene portato a termine in background. Per rendere ciò possibile, dobbiamo ricordarci che il progetto è basato su un server Node.js, il quale è di tipo *event-driven*, ed esegue le operazioni in modo asincrono. Proprio per questa caratteristica, la fase di upload di un oggetto 3D non è bloccante nei confronti dell'utente, che può così continuare ad inviare al server normali richieste HTTP.

## 4.1 Implementazione del Long Polling in unicom-product-editor

In questa fase andremo a vedere l'uso pratico della tecnologia appena descritta nel nostro editor di forme 3D.

Per quanto riguarda le normali chiamate HTTP Ajax, è stata usata jQuery, una libreria JavaScript veloce, leggera e ricca di funzionalità.

Prendiamo in esame il Controller Angular su cui si sviluppa la pagina dell'uploader:

```
1 $http.post('/upload', fd, {
2   withCredentials: true,
3   headers: {'Content-Type': undefined },
4   transformRequest: angular.identity
5 })
```

Codice 4.1: HTTP POST /upload

Per prima cosa si esegue la HTTP POST request relativa all'upload e successiva conversione del file .obj. La chiamata API è indicata subito dopo la funzione `$http.post`, ed è `/upload`. La risposta che arriverà da parte del server sarà l'endpoint del file JSON convertito, che metteremo nella variabile `filename`. Dopo questo si esegue una chiamata HTTP GET, passando come parametro il nome del file appena convertito, per inserire il file JSON nel database.

```
1 $http({method: 'GET', url: filename})
2   .then(function successGet(filename) {
3     $http({method: 'POST',
4           url: '/insert',
5           data: {shape: filename.data}
6         })
7     ...
```

Codice 4.2: HTTP GET /insert

La gestione degli errori avviene tramite la funzione `.then` della richiesta HTTP: se l'operazione va a buon fine, la risposta del server sarà il codice 200, OK, mentre in caso contrario il server risponderà con un codice di errore. In entrambi i casi il risultato è contenuto nel parametro `response`.

```
1 .then(function successResponse(response) {  
2     console.log(response);  
3     console.log('file_inserted_in_db');  
4     $scope.isRouteLoading = false;  
5     $scope.uploadsuccess = true;  
6 }, function errorResponse(response) {  
7     console.log(response);  
8     console.log('error_in_insert');  
9 });
```

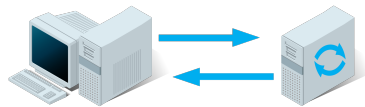
Codice 4.3: gestione degli errori nella richiesta HTTP

## 4.2 Altri metodi di comunicazione fra client e server

Di seguito vengono elencati, oltre al Long Polling, gli altri metodi di comunicazione fra client e server, con particolare riguardo al modo in cui vengono scambiate le richieste fra i due interlocutori.

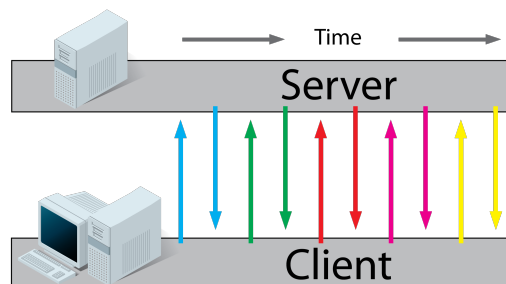
### 4.2.1 Normale HTTP

1. Il client richiede una pagina web al server.
2. Il server calcola la risposta.
3. Il server invia la risposta al client.



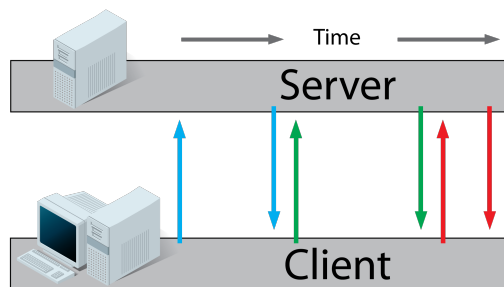
### 4.2.2 Polling Ajax

1. Il client richiede una pagina web al server usando il normale HTTP.
2. Il client riceve la pagina web richiesta ed esegue il codice JavaScript contenuto nella pagina, il quale richiede un file al server ad intervalli regolari (es. 0.5 secondi).
3. Il server calcola ogni risposta e la invia al client, come del normale traffico HTTP.



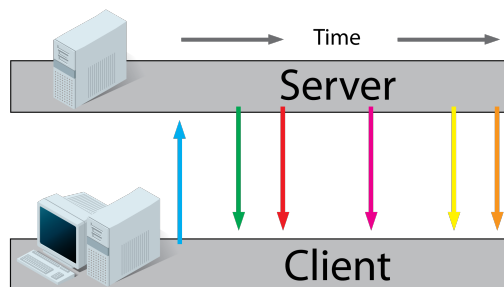
### 4.2.3 Long Polling Ajax

1. Il client richiede una pagina web al server usando il normale HTTP.
2. Il client riceve la pagina web richiesta ed esegue il codice JavaScript contenuto nella pagina, il quale richiede un file al server.
3. Il server non risponde immediatamente con l'informazione richiesta, ma aspetta finché non c'è una nuova informazione disponibile.
4. Quando l'informazione è disponibile, il server risponde con questa.
5. Il client riceve la nuova informazione e manda immediatamente un'altra richiesta al server, riavviando il processo.



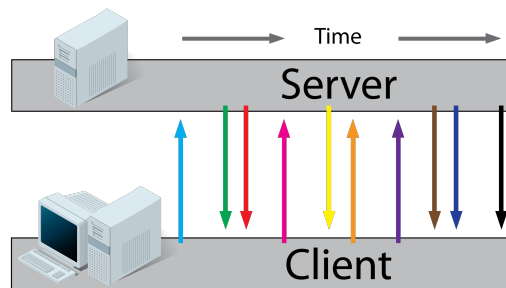
### 4.2.4 HTML5 Server Sent Events (SSE)/EventSource

1. Il client richiede una pagina web al server usando il normale HTTP.
2. Il client riceve la pagina web richiesta ed esegue il codice JavaScript contenuto nella pagina, il quale apre una connessione al server.
3. Il server invia un evento al client non appena è disponibile una nuova informazione.



#### 4.2.5 HTML5 Websockets

1. Il client richiede una pagina web al server usando il normale HTTP.
2. Il client riceve la pagina web richiesta ed esegue il codice JavaScript contenuto nella pagina, il quale apre una connessione al server.
3. Il client e il server possono ora scambiarsi messaggi quando sono disponibili nuove informazioni (da qualsiasi lato).



## 5. Implementazione

In questo capitolo si mostra come sono state implementate le funzionalità richieste dalle user stories mostrate nella tabella 1.1. Per ogni funzionalità vedremo quali API sono state create, come sono state documentate attraverso Postman ed il codice utilizzato per realizzarle.

Tutte le rotte relative ai servizi API sono presenti nel file `/routes.js`, ad esclusione dei servizi relativi alla registrazione e autenticazione di un utente, che si trovano direttamente nel file `/app.js`.

### 5.1 US-01/03

Per l'implementazione delle prime tre user stories, che verranno trattate insieme, essendo per lo più simili tra loro, sono stati creati i seguenti servizi:

```
1 module.exports = function(app) {
2   var List = require('./controllers/lists');
3   var Object = require('./controllers/objects');
4   var Part = require('./controllers/parts');
5   var Option = require('./controllers/options');
6
7   app.get('/lists', List.findAll);
8   app.get('/lists/:id', List.findById);
9   app.post('/lists', List.add);
10  app.put('/lists/:id', List.update);
11  app.delete('/lists/:id', List.delete);
12
13  app.get('/objects/:listid', Object.findAll);
14  app.get('/objects/:id', Object.findById);
15  app.post('/objects/:listid', Object.add);
16  app.put('/objects/:id', Object.update);
17  app.delete('/objects/:id', Object.delete);
18
19  app.get('/parts/:objectid', Part.findAll);
20  app.get('/parts/:id', Part.findById);
21  app.post('/parts/:objectid', Part.add);
22  app.put('/parts/:id', Part.update);
23  app.delete('/parts/:id', Part.delete);
24
25  app.get('/options/:partid', Option.findAll);
26  app.get('/options/:id', Option.findById);
27  app.post('/options/:partid', Option.add);
28  app.put('/options/:id', Option.update);
29  app.delete('/options/:id', Option.delete);
30 }
```

Codice 5.1: `/routes.js`

Come si può vedere, ad ogni operazione CRUD è associata la relativa API, che esegue una specifica funzione presente nel file corrispondente. I file relativi alle suddette fun-

zioni sono contenuti nella cartella `controllers` e vengono richiamati all'inizio del file `routes.js`.

Le funzioni sono sufficientemente autoesplicative, ma per chiarezza vengono elencate ed esposte qui di seguito:

- `app.get` si divide in due casi:
  1. `/API` in cui vengono richiesti tutti i Document di una Collection. Nel caso in cui la Collection sia un Object, una Option o una Part, viene passato in input il parametro `:id` che identifica il Document *parente*.
  2. `/API/:id` in cui viene passato il parametro `:id`, per interrogare la Collection nel database MongoDB secondo il criterio di identificazione.
- `app.post`

Con questa operazione si aggiunge un Document alla corrispondente Collection. Questo è l'unico caso in cui vanno distinte le Lists dagli altri oggetti. Quando si aggiunge una List tramite metodo POST, non viene passato nessun parametro alla chiamata API. Si noti invece che negli altri casi, durante la medesima operazione, viene sempre passato l'id dell'oggetto *parente*, dell'oggetto ossia a cui fa riferimento il Document che stiamo inserendo.
- `app.put`

Questa operazione viene usata per modificare un Document già presente in una Collection. Prende in input il parametro `:id` per poter ritrovare il giusto oggetto su cui eseguire un *update*.
- `app.delete`

Infine l'operazione *delete* si usa per eliminare un Document specifico, che viene identificato, come sopra, prendendo in input il parametro `:id`.

Di seguito prendiamo in esame il controller relativo agli Objects per analizzare le relative funzioni, seguite dalla relativa documentazione realizzata con Postman:

```
1 var mongoose = require('mongoose'),
2 Object = require('../app/schema.js').model('Object');
3
4 exports.findAll = function(req, res){
5   var listid = req.params.listid;
6   Object.find({'_list': listid}, function(err, docs) {
7     return res.send(docs);
8   });
9 };
10
11 exports.findById = function(req, res){
12   var id = req.params.id;
13   Object.findOne({'_id': id, '_list': listid}, function(err, docs) {
14     return res.send(docs);
15   });
16 };
17
18 exports.add = function(req, res) {
19   var listid = req.params.listid;
20   Object.create({"nome": req.body.nome, "descrizione": req.body.descrizione,
21     "_list": listid}, function (err, docs) {
22     if (err) return console.log(err);
23     return res.send(docs);
24   });
25 };
26
27 exports.update = function(req, res) {
```



```

28     var id = req.params.id;
29     var updates = req.body;
30     console.log(id);
31     Object.update({"_id":id}, req.body,
32     function (err, numberAffected) {
33         if (err) return console.log(err);
34         console.log('Updated %d Lists', numberAffected);
35         return res.sendStatus(202);
36     });
37 };
38
39 exports.delete = function(req, res) {
40     var id = req.params.id;
41     Object.remove({'_id':id}, function(result) {
42         return res.send(result);
43     });
44 };

```

Codice 5.2: /controllers/objects.js

L'interfacciamento con il database è gestito tramite mongoose, di cui viene importata la dipendenza all'inizio del file. La prima funzione è `findAll`, che prende in input, attraverso i parametri presenti nella request, l'id della List a cui l'Object è associato. Viene quindi eseguita la funzione `.find` di mongoose sulla Collection Objects, filtrando per l'id appena passato, e quello che si avrà come risposta sarà una callback, che conterrà un messaggio di errore in caso di fallimento, o i Document richiesti in caso di successo. A questo punto il server invia la risposta tramite la funzione `res.send`, includendo i risultati dell'interrogazione appena effettuata sul database. La funzione `findById` è molto simile alla precedente, pertanto non verrà spiegata nuovamente.

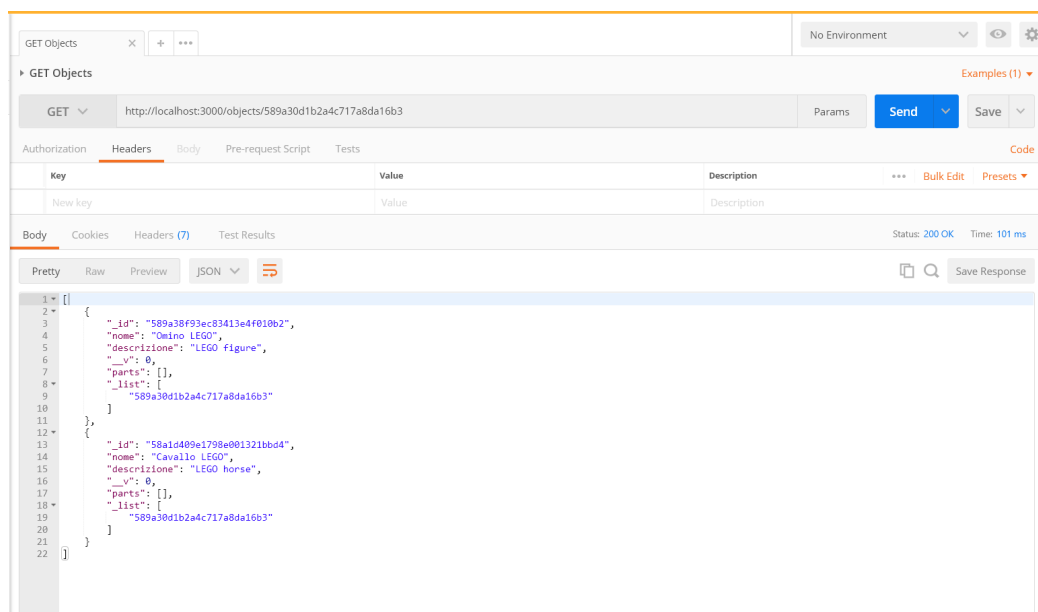
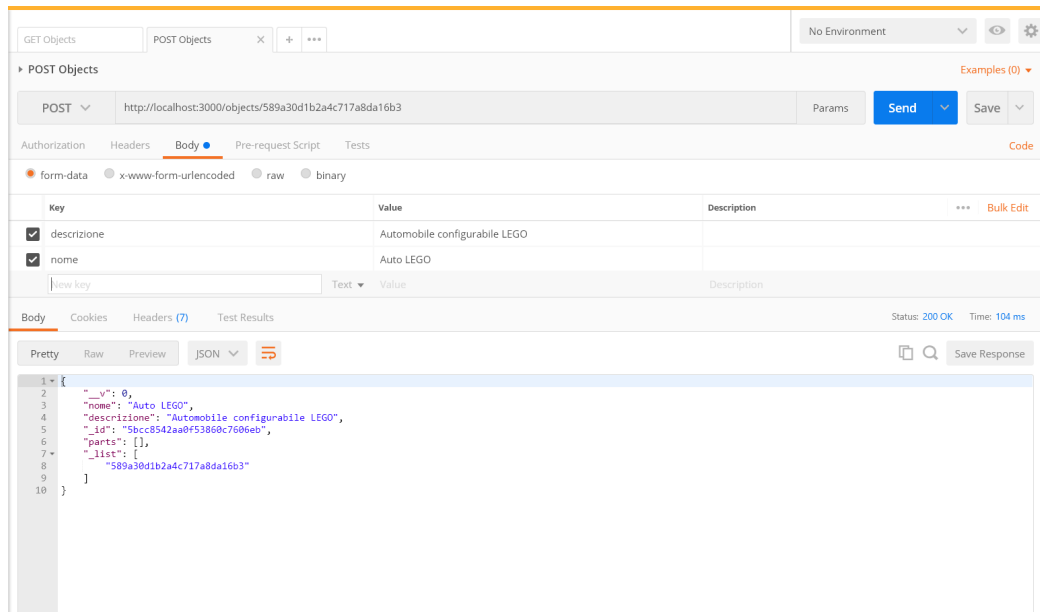
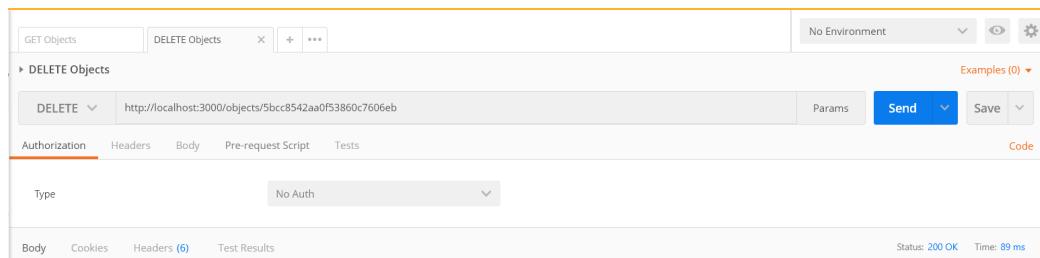


Figura 5.1: Documentazione get /objects/:id

La funzione `add` inserisce un nuovo Object Document nella sua Collection, solo dopo però averlo inizializzato inserendo l'id della List a cui fa riferimento nell'attributo `_list`, il nome (preso dal `body` della richiesta HTTP) nell'attributo `nome` e la descrizione nel corrispondente attributo. I parametri `nome` e `descrizione` sono stati inseriti dall'utente negli appositi campi presenti nella pagina di visualizzazione degli Objects, e vengono trasmessi al server attraverso la richiesta HTTP POST.

Figura 5.2: Documentazione post `/objects/:id`

L'ultima funzione è `delete`. Questa rimuove l'Object cercandolo attraverso il suo `id`. La funzione non necessita di alcun parametro aggiuntivo.

Figura 5.3: Documentazione delete `/objects/:id`

Le funzioni si comportano in modo analogo, o comunque come spiegato sopra, con gli altri tipi di oggetti in `unicam-product-editor`. Per quanto riguarda la funzione di visualizzazione dei documenti richiesta dalle user stories dalla 01 alla 03, è stata sviluppata una applicazione single-page in AngularJS, che andremo a vedere nella US-05.

## 5.2 US-04

La user story 04 richiede di avere un endpoint che restituisca la forma JSON di un oggetto 3D da visualizzare.

Per implementare questa funzione, è stata creata una chiamata API apposita, contenuta nel file del server `app.js`.

```

1 app.get('/shape/:id', cors(corsOptions), function(req, res){
2   var search = req.params.id;
3   converted(req, search, res);
4 });

```

Codice 5.3: shape API



La root del client sviluppato in AngularJS è interamente contenuta nella cartella `/public`, ed oltre al file principale, `/public/app.js`, sono presenti le cartelle dei principali componenti usati dal framework, come i *controllers*, le *directives*, i *services*, la cartella `vendors` che ospiterà in locale le librerie usate in `unicam-product-editor`, come la già citata `jQuery`, la cartella `partials` che conterrà le pagine HTML richieste come risorse e la cartella `stylesheets` che racchiude i file `.css`, ossia i fogli di stile, usati dall'applicazione.

Nel file principale, `/public/app.js` sono indicati tutti gli stati usati dalla libreria `$stateProvider` e in cui il client può trovarsi in fase di navigazione.

```
1 $stateProvider
2 .state('home', {
3     url: '/',
4     controller: 'HomeController',
5     templateUrl: '/static/partials/home.html'
6 })
```

Codice 5.5: `/public/app.js`

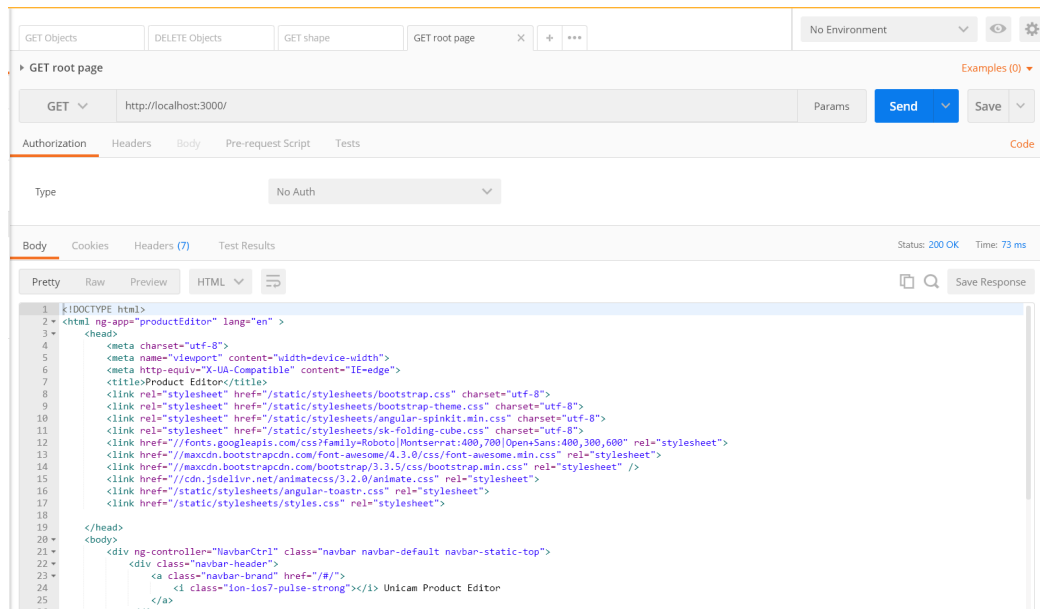


Figura 5.5: Documentazione get `/`

Quello appena esposto, ad esempio, è lo stato relativo alla route principale. Non appena si digiterà l'URL `http://localhost:3000/` nel caso il server sia ospitato in locale, o `http://unicam-product-editor.herokuapp.com/` nel caso in cui si sfrutti l'applicazione rilasciata su Heroku, AngularJS riconoscerà di trovarsi nello stato di *home*, e invierà al client come risposta la pagina HTML indicata nella proprietà `templateUrl`, `/static/partials/home.html`, e a questa associerà il relativo controller, `HomeController`. Tutti i controller sono contenuti nella cartella `/public/controllers`.

I controller servono per eseguire le operazioni lato client, per inizializzare le richieste HTTP a seguito di un'azione. Andiamo per esempio ad analizzare il controller relativo alla pagina web delle *Lists*:

```
1 angular.module('productEditor')
2 .controller('ListsCtrl', function($scope, $http, $location, toastr){
3     $http.get('/lists')
```

```

4      .success(function(docs) {
5          $scope.lists = docs;
6      })
7      .error(function(err) {
8          toastr.error('Error!_Something_went_wrong');
9      });
10     ...

```

Codice 5.6: /public/controllers/lists.js

le istruzioni esposte qui sopra fanno sì che, non appena il client richiede la pagina web delle Lists, il controller avvia la richiesta HTTP `/lists`, tramite la quale si richiedono tutte le Lists presenti nella Collection Lists. La risposta viene messa nella variabile `$scope.lists`, e viene poi renderizzata tramite AngularJS nella pagina HTML

```

1     ...
2     <tr ng-repeat="list_in_lists">
3         <td ng-hide="list.edit">{{list.name}}</td>
4         <td ng-show="list.edit"><input type="text" ng-model="list.name"></td>
5     ...

```

Codice 5.7: /public/partials/lists.html

La direttiva `ng-repeat` ripete l'elemento *lista* contenuto nella variabile `lists` dello `$scope`, esponendone la proprietà `nome` tante volte quante sono le istanze degli elementi ripetuti nella variabile.

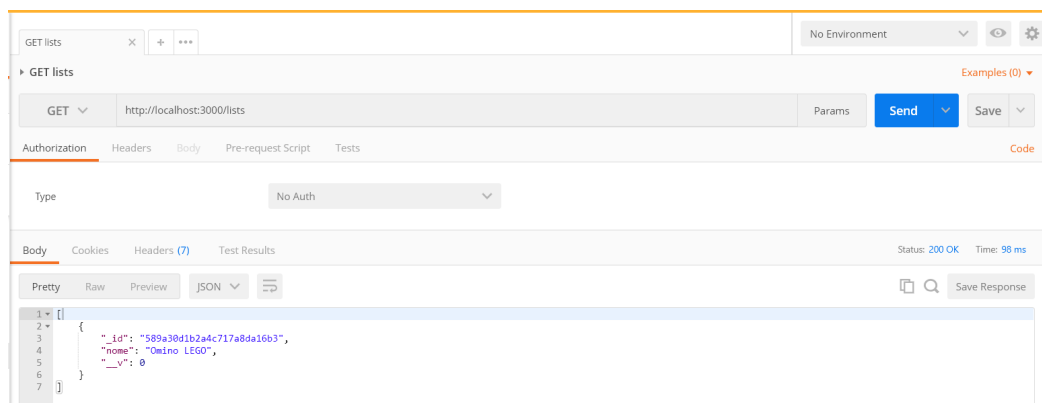


Figura 5.6: Documentazione get /lists

La breve panoramica appena esposta spiega quindi il percorso che seguono le informazioni dalla fase di richiesta del client, all'elaborazione della risposta dal server, fino alla gestione della risposta ricevuta dal client.

## 5.4 US-06

La prossima user story richiede che il processo di caricamento della forma 3D avvenga in modo sincrono, e che non sia bloccante. Abbiamo già trattato il modo in cui il long polling sia stato implementato nel client nel capitolo 4, pertanto andremo a vedere soltanto il codice lato server.

Nel processo di caricamento di una forma 3D, la prima richiesta HTTP che viene effettuata è una richiesta POST, che contiene nel body il file `.obj` della forma che si vuole caricare. A questo punto la API chiama la funzione `upload` dal file `/app/upload.js`.

```

1 var express = require('express'),
2   fileUpload = require('express-fileupload'),
3   bodyParser = require('body-parser'),
4   app = express(),
5   mongodb = require('mongodb'),
6   path = require('path'),
7   http = require('http'),
8   fs = require('fs'),
9   path = require('path'),
10  convert = require('./convert');
11
12 app.use(express.static('app'));
13 app.use(fileUpload());
14
15 module.exports = function upload(req, res) {
16   var filename = req.files.file.name;
17
18   var ext = path.extname(filename);
19   if (ext !== '.obj') {
20     console.log('File_is_not_.obj');
21     res.redirect(505, 'Error_in_file_inserted');
22   }
23   else {
24     var target_path = './public/tmp/' + filename;
25     req.files.file.mv(target_path, function (err) {
26       if (err) throw err;
27       console.log('File_uploaded!');
28       convert(req, filename, res);
29     });
30   }
31 };

```

Codice 5.8: /app/upload.js

La funzione di upload salva il file 3D .obj nel percorso /public/tmp con lo stesso nome con cui viene caricato. A questo punto viene chiamata a sua volta la funzione convert, localizzata nella stessa directory del file upload.js.

```

1 var express = require('express'),
2   app = express(),
3   PythonShell = require('python-shell'),
4   bodyParser = require('body-parser'),
5   fs = require('fs'),
6   path = require('path'),
7   upload = require('./upload'),
8   readjsonfile = require('./readjsonfile');
9
10 module.exports = function convert (req, filename, res){
11   filename = './public/tmp/' + filename;
12   var truncate = filename.substr(0, filename.lastIndexOf('.')) || filename;
13   var options = {
14     mode: 'text',
15     scriptPath: './scripts',
16     args: ['-i', filename, '-o', truncate + '.json']};
17
18   PythonShell.run('converter.py', options, function (err, results) {
19     if (err){
20       throw err;
21     }
22     console.log('results:_%j', results);
23     filename = truncate + '.json';
24     res.setHeader('Content-Type', 'application/json');
25     res.send(JSON.stringify(filename, null, 3));
26   });
27 };

```

Codice 5.9: /app/convert.js

Il file appena caricato viene convertito in JSON attraverso lo script Python `converter.py` contenuto nella cartella `/scripts`, che come detto in precedenza fa parte della libreria `Three.js`. Per essere eseguito, lo script viene passato come parametro alla funzione `run` della libreria `PythonShell`. Una volta finita l'operazione di conversione, si invia il file JSON in risposta come endpoint, tramite la funzione `JSON.stringify`. L'ultima operazione che viene eseguita è quella di inserimento del file JSON nel database, che avviene tramite la chiamata `HTTP /insert`. La API corrispondente rimanda il body dell'endpoint (quindi il JSON da inserire) alla funzione `readjsonfile`, che dopo aver inserito il JSON nel database, elimina i file temporanei creati durante l'intero processo.

```

1 var express = require('express'),
2   app = express(),
3   bodyParser = require('body-parser'),
4   fs = require('fs'),
5   path = require('path'),
6   mongoose = require('mongoose'),
7   mongodb = require('mongodb'),
8   convert = require('./convert');
9
10 var id;
11 var screenid;
12 var path;
13
14 module.exports = function readjsonfile(res, data, db){
15   var coll = mongoose.connection.db.collection('shapes');
16   coll.save(data);
17   path = "./public/tmp/" + data.metadata.sourceFile;
18   id = data._id;
19   screenid = 'ID:' + id;
20   console.log(screenid);
21   fs.unlinkSync(path);
22   path = (path.substr(0, path.lastIndexOf('.')) || path) + '.json';
23   fs.unlinkSync(path);
24   res.setHeader('Content-Type', 'application/json');
25   res.send(JSON.stringify('insert_success', null, 3));
26 }

```

Codice 5.10: `/app/readjsonfile.js`

Va da sé che tutte queste operazioni devono essere eseguite consecutivamente ed in un certo ordine, e per questo il processo di upload usato in `unicam-product-editor` entra in contrasto con la natura di `Node.js` che, essendo un sistema event-driven, esegue le operazioni in modo asincrono. La chiave che fa sì che le operazioni di questo processo siano eseguite nel giusto ordine sono le callback: durante il passaggio da una sequenza di istruzioni all'altra, nei punti chiave, le funzioni vengono chiamate usando le callback, pertanto l'esecuzione di queste istruzioni non si conclude finché non finisce il processo successivo, e finché quindi non si riceve il risultato attraverso la callback, formando una catena di operazioni eseguite in modo sincrono.

## 5.5 US-07

La ultima user story prevede che un nuovo utente all'interno di `unicam-product-editor` possa registrarsi, o che un utente già registrato possa effettuare un login.

Per questa funzionalità in `unicam-product-editor` è stato usato `Satellizer`, un modulo aggiuntivo creato per `AngularJS` di autenticazione basato sui token. Con questo sistema l'utente è in grado di registrarsi ed eseguire il login sia con una semplice autenticazione basata su username e password che con l'autenticazione da terze parti (OAuth 1.0 e OAuth 2.0), come Google, Facebook, LinkedIn, Twitter, Instagram, GitHub ed altri.

Nel nostro progetto è stata implementata l'autenticazione attraverso Google e Facebook, oltre a quella con indirizzo e-mail e password.

L'implementazione di questo modulo prevede i seguenti step (l'esempio qui riportato è quello dell'autenticazione Google):

1. Installazione tramite il comando `npm install satellizer`.
2. Importazione del Javascript nel client attraverso la pagina HTML, usando il tag `<script src="satellizer.js"></script>`.
3. Importazione del modulo nel controller principale della Single Page Application

```
1   $authProvider.google({  
2       clientId: 'Google_Client_ID'  
3   });
```

4. Implementazione del sistema di chiamate HTTP nel controller della pagina di autenticazione

```
1   $scope.authenticate = function(provider) {  
2       $auth.authenticate(provider);  
3   };
```

5. Configurazione della chiamata HTTP specifica per ogni opzione di autenticazione

```
1   $authProvider.google({  
2       url: '/auth/google',  
3       authorizationEndpoint: 'https://accounts.google.com/o/oauth2/auth',  
4       redirectUri: window.location.origin,  
5       requiredUrlParams: ['scope'],  
6       optionalUrlParams: ['display'],  
7       scope: ['profile', 'email'],  
8       scopePrefix: 'openid',  
9       scopeDelimiter: '_',  
10      display: 'popup',  
11      oauthType: '2.0',  
12      popupOptions: { width: 452, height: 633 }  
13  });
```

6. Nel caso di autenticazione OAuth come lo è Google, ottenere le Chiavi OAuth nella sezione sviluppatori di Google (Google+ APIs).

Una volta configurato correttamente, si può usare Satellizer all'interno di un qualsiasi controller della nostra applicazione web usando il modulo `$auth` con una delle funzioni messe a disposizione, come `login` per effettuare il login, `signup` per registrarsi o `isAuthenticated` per controllare la presenza di un token attivo di autenticazione. In caso di login corretto, il server invierà come risposta il token di autenticazione. In caso contrario, la risposta del server sarà un messaggio di errore con stato 401, `Unauthorized`.



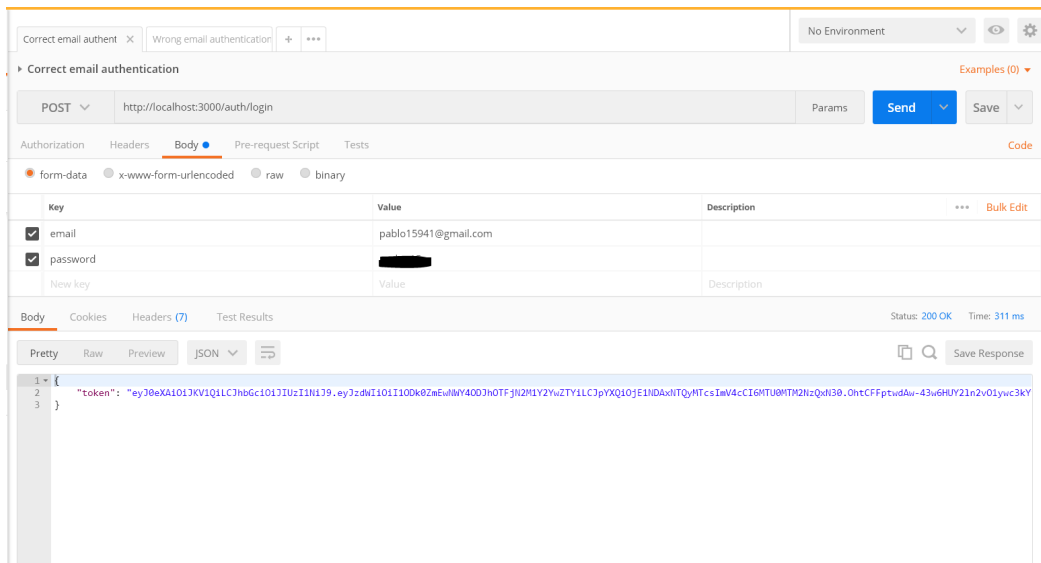


Figura 5.7: Documentazione post /auth/login

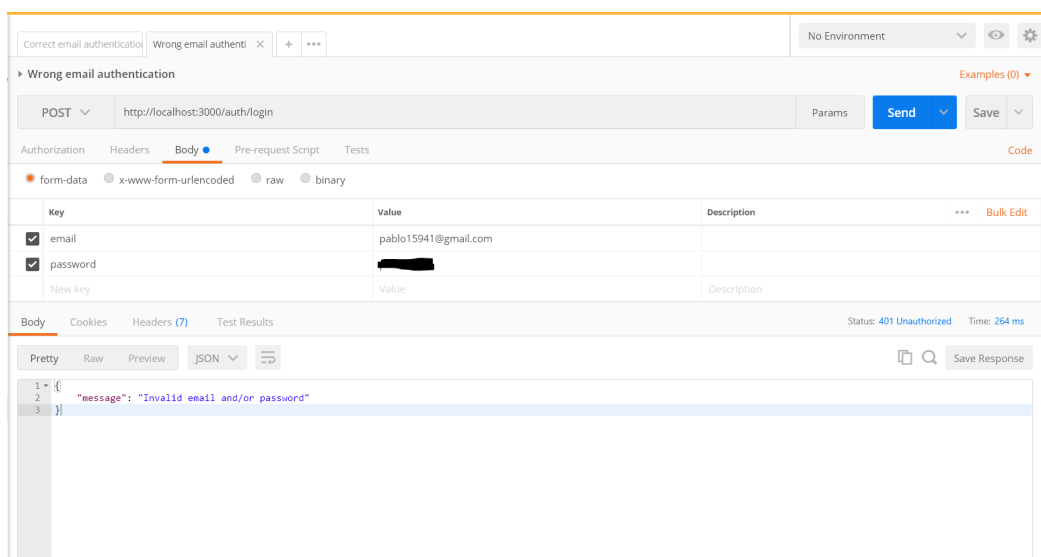


Figura 5.8: Documentazione post /auth/login sbagliato



## 6. Conclusioni



## A. Installazione e Uso



## B. Screenshot