

Tesina:

Gestione della privacy e dei dati

Riduzione di casistica di data breach portata al minimo, i dati di ciascun utente sono al sicuro siccome nessun altro al di fuori del proprietario della sessione può accedere ai dati dell'utente in questione se non facendo l'accesso mediante username e password.

Dati non trasmessi al di fuori dell'unione europea, non verrà effettuata nessuna operazione che implichi l'invio dei dati personali al di fuori dell'UE.

I dati non-sensibili potranno essere soggetti a trattamenti a fini di marketing e si ha la possibilità di revocare in qualsiasi momento questo consenso.

Siccome per l'utilizzo dell'app è richiesto questo consenso, nel caso in cui si voglia revocarlo, si deve dunque utilizzare la funzione di "Elimina utente e relativi dati" esposta nella sezione personale per tutelare anche il diritto all'oblio.

Verranno eliminati tutti i dati dell'utente da tutte le tabelle del database tramite l'api fornita apposta per

Questa Applicazione web utilizza algoritmi di crittografia hash (**md5**) per preservare la riservatezza delle password e delle cartelle contenenti le ricevute pdf.

I termini e le condizioni della privacy sono disponibili cliccando sul link apposito di fianco alla spunta per accettarli così da permettere all'utente di prenderne visione prima di decidere se accettare o meno.

Tecnologie utilizzate:

- PHP



Php è stato utilizzato per la programmazione dell'intero back-end e, poche volte, come view-engine

- Javascript



Javascript è stato usato per manipolare il DOM e il front-end in generale e anche per utilizzare le API che ho messo a disposizione con PHP

- JQuery



Jquery è quell'enorme libreria di Javascript che ho utilizzato per semplificare la manipolazione del DOM dipendentemente da determinati eventi (Es: dopo che si ha a disposizione un risultato di una chiamata di un API)

- CSS



Ho usato CSS per abbellire l'HTML attraverso tutti i suoi attributi, ho scritto un mio file CSS di stile personale di circa 565 righe per ritoccare l'estetica delle componenti html

- Bootstrap 5



Bootstrap 5 è una CDN di librerie CSS e l'ho usato per dare una sorta di “stile base” al mio HTML, che poi ho modificato col mio CSS personale

- HTML



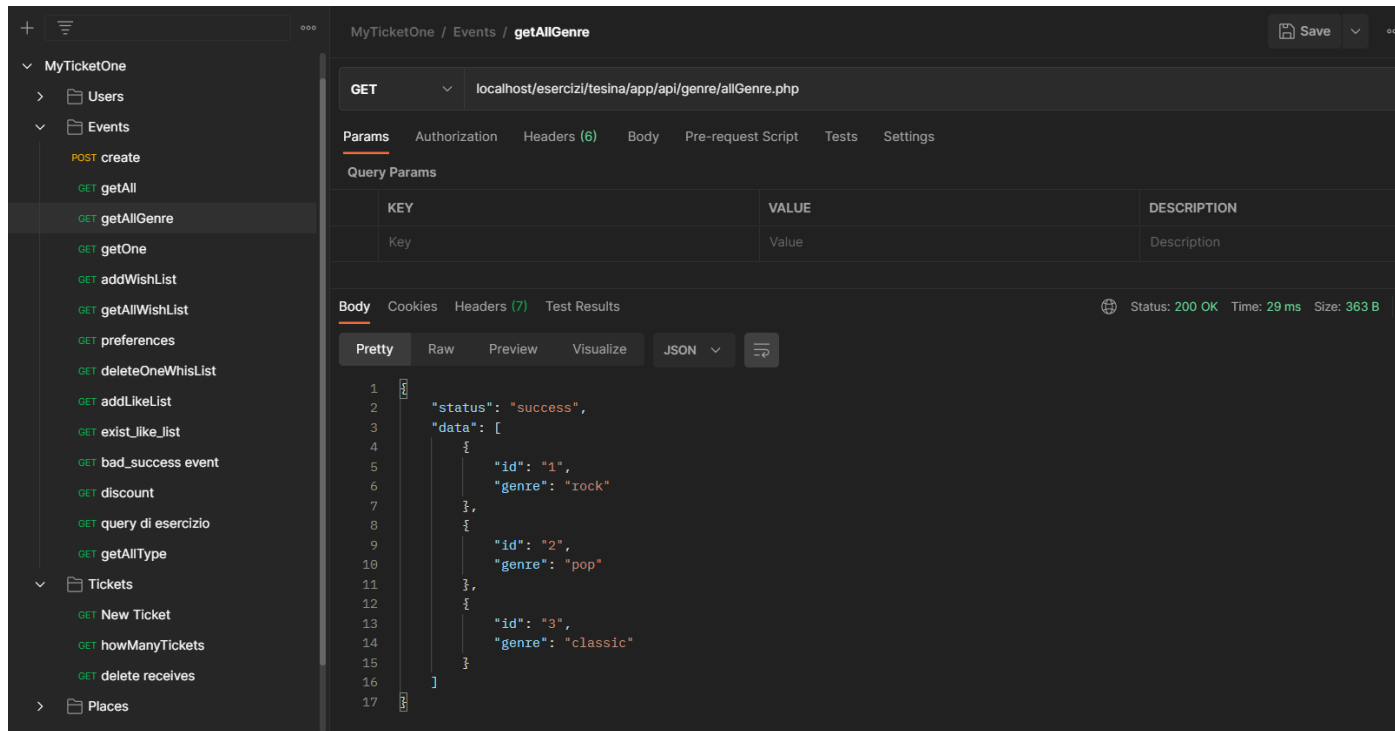
HTML è un famosissimo linguaggio di markup che viene utilizzato per visualizzare le pagine sul web.

Ho usato questo linguaggio per renderizzare le pagine web del mio progetto.

- Postman



Postman è un ambiente dove poter testare le api. Questo è un esempio di schermata che racchiude la maggior parte dei risultati della api salvati



- Git



Ho usato Git per tenere in continuo aggiornamento la mia repo su GitHub (<https://github.com/francescoBassi2002/tesina>) tramite VisualStudio.

Il db.php

Primo sguardo a questa classe:

```
db.php M X
config > db.php > db
1  <?php
2  class db {
3      private $pdo = null;
4      private $results;
5      private $query;
6      private $query_count = 0;
7      private $queryStr = '';
8      private $params;
9
10     function __construct($host = 'localhost', $user = 'root', $passowrd = '' , $dbname = 'test'){
11         $dsn = 'mysql:host=' . $host . ';dbname=' . $dbname;
12         $this->pdo = new PDO($dsn , $user , $passowrd);
13         $this->pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_OBJ);
14         $this->pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
15     }
16     public function query($sql, $params = array()){
17         try{
18             if ($this->pdo){
19                 $this->query = $this->pdo->prepare($sql);
20
21
22                 $this->query->execute($params);
23
24
25                 $this->query_count +=1;
26                 return $this;
27             }
28         }catch(PDOException $e){
29             echo ('pdo error: ' . $e->getMessage());
30             return false;
31         }
32     }
33     public function FetchAll(){
34         try{
35             if ($this->queryStr == ''){
36                 return $this->query->fetchAll(PDO::FETCH_ASSOC);
37             }else{
38                 $sql = $this->queryStr;
```

Ln 70, Col 5 S

Questa classe mi permette di semplificare l'utilizzo della classe PDO per gestire il database.

Per esempio, se volessi aprire la connessione al database e prelevare tutti i record da una tabella senza l'utilizzo della classe db dovrei scrivere:

```
$conn = new PDO("mysql:host=<nome host>;dbname:<nome del db" ,
"<utente>" , "<password>");

$query = $conn->prepare("SELECT * FROM <TABELLA>");

$query->execute(["<EVENTUALE ARRAY DI PARAMETRI, IN QUESTO CASO NON
CI SONO>"]);

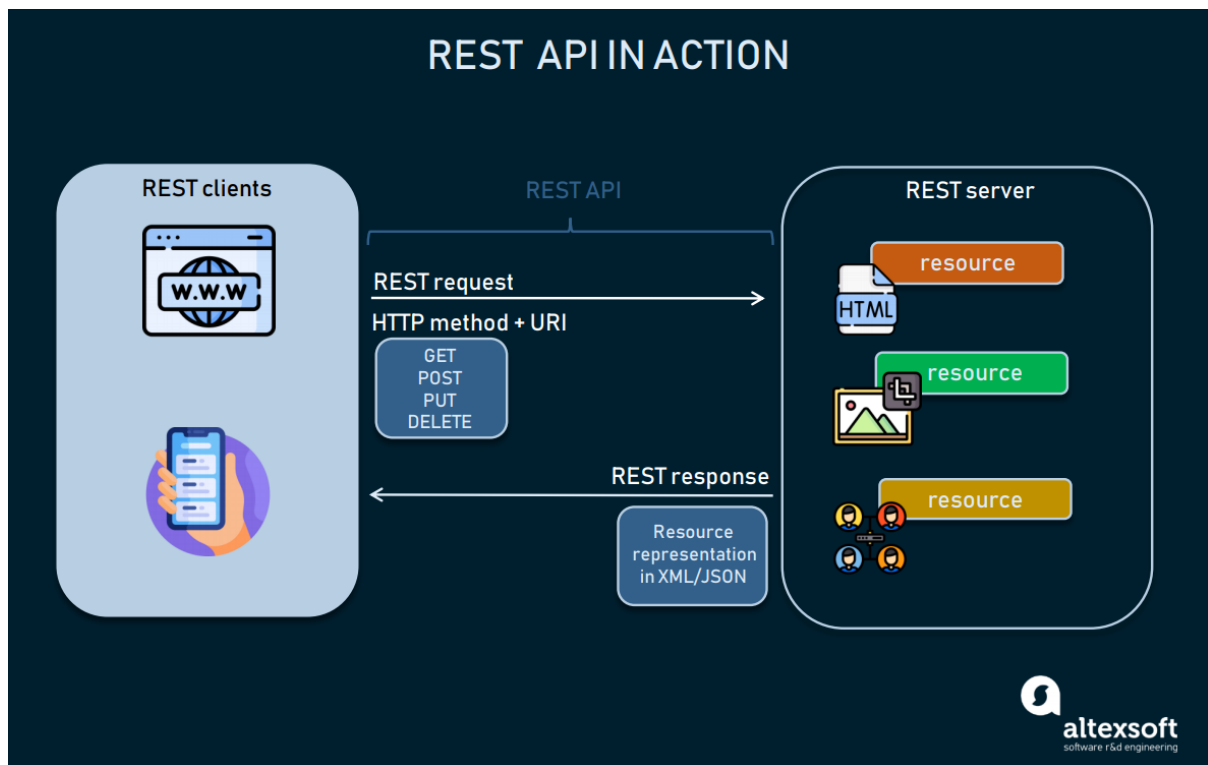
$elenco_record = $query->fetchAll(PDO::FETCH_ASSOC);
```

Mentre invece usando la classe db posso invece scrivere:

```
$conn = new db("<nome host>" , "<utente>" , "<password>" , "<nome
del db>");

$elenco_record = $conn->query("SELECT * FROM <TABELLA>" ,
["<EVENTUALE ARRAY PARAMETRI>"])->fetchAll();
```

REST API App - (MVC structure):



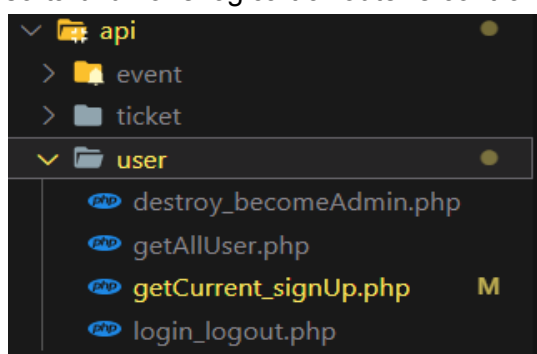
REST (REpresentational State Transfer) è un insieme di vincoli strutturali.

Seguendo questo schema architetturale esiste sempre un client che richiede al server la risorsa e lo stato di quella risorsa. La risorsa può essere in formato JSON, XML, Py, Ecc... Ma solitamente viene usato il formato JSON perché è universale per tutti i linguaggi e facilmente leggibile (user-friendly).

Ci sono dunque 3 entità principali che gestiscono questo tipo di architettura: router, controller, il e i models.

I **router** si occupano di gestire l'indirizzamento della richiesta fatta tramite un url ad un'apposita funzione, definita dall'entità **controller**, che risponderà sempre con un JSON (nel mio caso). Spesso e sovente la funzione che elabora la richiesta deve farlo accedendo alle risorse di un database, qui entra in gioco l'entità **model**, che si occupa di gestire le query ("richieste") al database e di fornire il risultato al controller che completerà il processo. Ognuna di queste entità è suddivisa per ogni entità relazionale nel db.

Nel mio caso ho chiamato la cartella dei controller "api" e, usando PHP ho dovuto fare un sorta di unione logica tra router e controller.

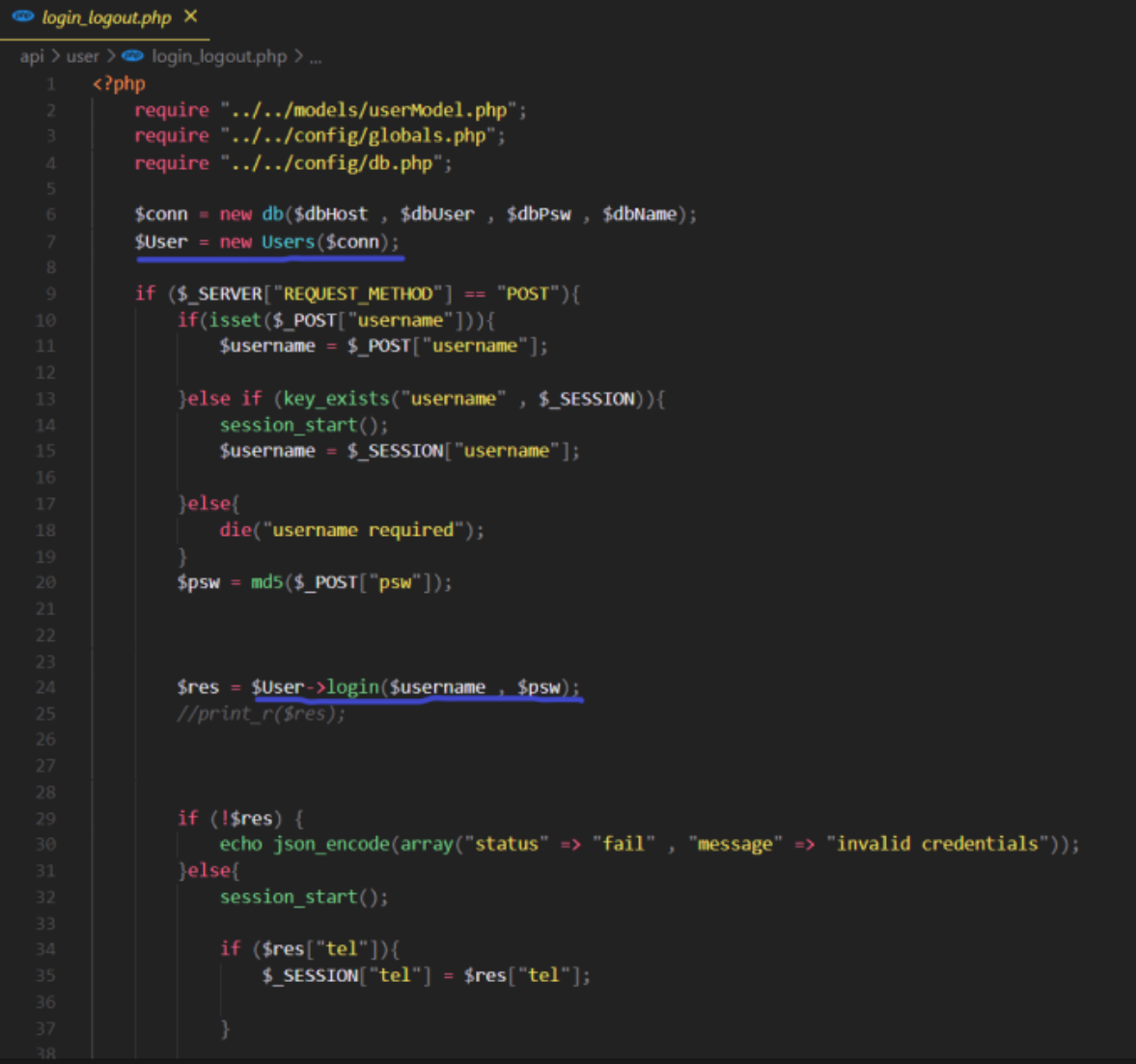


Come si può vedere ci sono diverse cartelle nelle quali sono contenute le API necessarie per ogni entità. Per esempio, se volessi effettuare il login e avere un riscontro in formato JSON di come è andata l'operazione dovrei chiamare l'api con url

"localhost/[...]/api/user/login_logout.php" con metodo POST.

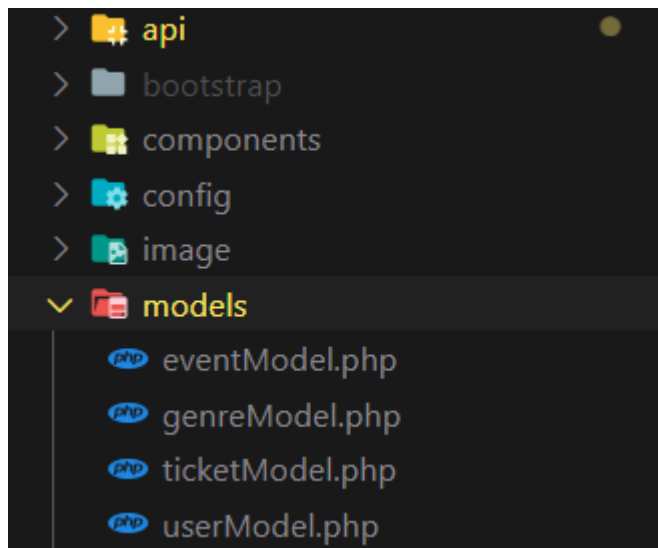
Invece se chiamo la stessa API col metodo GET effettuerò il logout della sessione.

Guardando bene le parti sottolineate...

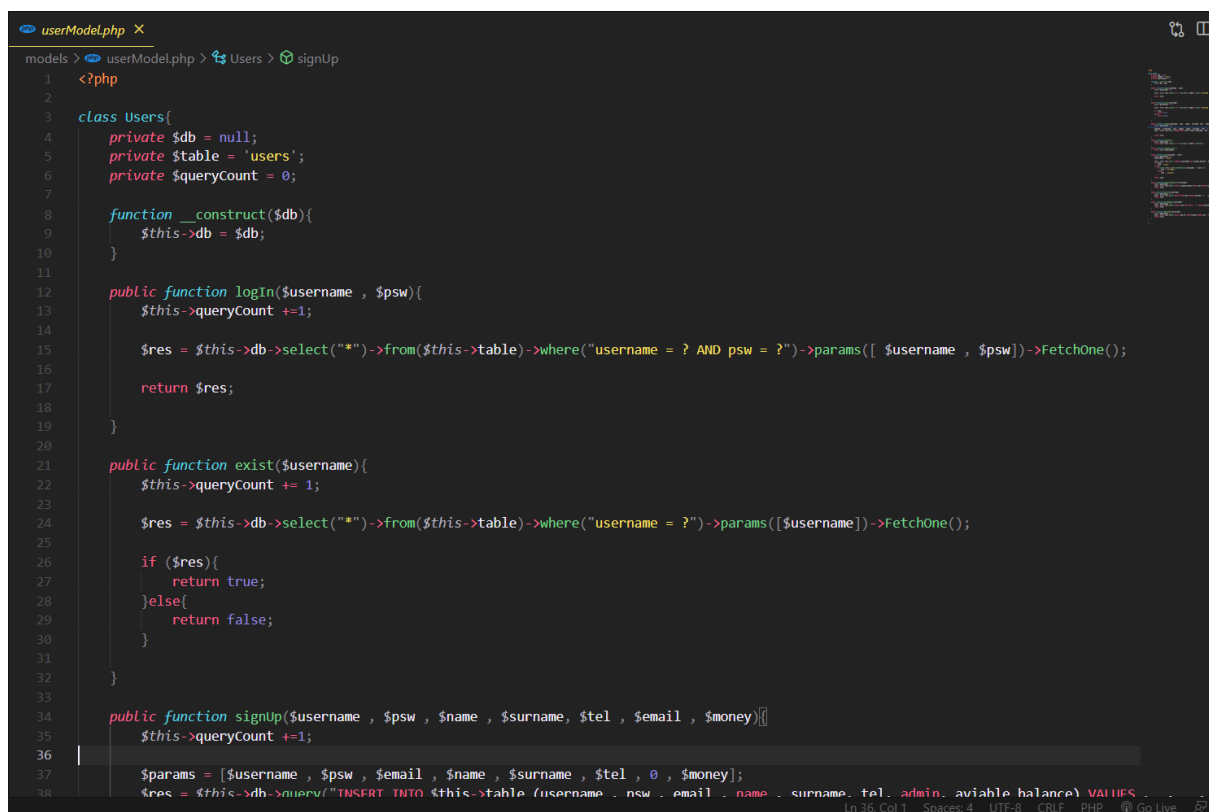


```
login_logout.php X
api > user > login_logout.php > ...
1  <?php
2  require "../models/userModel.php";
3  require "../config/globals.php";
4  require "../config/db.php";
5
6  $conn = new db($dbHost , $dbUser , $dbPsw , $dbName);
7  $User = new Users($conn);
8
9  if ($_SERVER["REQUEST_METHOD"] == "POST"){
10     if(isset($_POST["username"])){
11         $username = $_POST["username"];
12
13     }else if (key_exists("username" , $_SESSION)){
14         session_start();
15         $username = $_SESSION["username"];
16
17     }else{
18         die("username required");
19     }
20     $psw = md5($_POST["psw"]);
21
22
23
24     $res = $User->login($username , $psw);
25     //print_r($res);
26
27
28
29     if (!$res) {
30         echo json_encode(array("status" => "fail" , "message" => "invalid credentials"));
31     }else{
32         session_start();
33
34         if ($res["tel"]){
35             $_SESSION["tel"] = $res["tel"];
36
37         }
38
39     }
```

Come si può notare ho utilizzato un'istanza di una classe "Users". Questa classe viene importata alla riga 2 con "require '../models/userModel.php'". Grazie al metodo login restituirà TRUE se il login è riuscito e FALSE se le credenziali sono errate o se è andato storto qualcosa. Ora andrò a vedere nel dettaglio come funziona e dove si trova questo "../models/userModel.php".



Ecco spiegato il percorso: “..” per spostarsi di un livello indietro (infatti viene usato due volte per tornare alla cartella principale) “/models” per entrare nella cartella models e “userModel.php” per dire da quale file proviene il codice da importare in “login_logout.php”.



Come si può vedere questa è la parte dei models dedicata alla tabella utenti.

C'è una classe (Users) che sfrutta un'istanza della classe “db” creata per semplificare la gestione del database col PDO nel “db.php” nella cartella “config”.

Si possono vedere i primi tre metodi di una lunga serie e uno tra questi è proprio il metodo “login” usato in precedenza. E in questo caso fa una query al database prendendo i dati

dell'utente che ha effettuato il login, se l'utente esiste ritornerà i campi (informazioni) di quell'utente, se non esiste questo metodo ritornerà dunque un valore nullo.

Fatto quest'esempio si può notare come la cartella "models" rappresenta la gestione dei **modelli** e come la cartella API gestisca sia le **route** (i nomi per chiamare le api) e i **controller** (le API stesse, nell'esempio "login_logout.php").

Tornando a "login_logout.php":

```
db.php M login_logout.php X
api > user > login_logout.php > ...
18     die("username required");
19 }
20 $psw = md5($_POST["psw"]);
21
22
23
24 $res = $User->login($username, $psw);
25 //print_r($res);
26
27
28
29 if (!$res) {
30     echo json_encode(array("status" => "fail", "message" => "invalid credentials"));
31 } else {
32     session_start();
33
34     if ($res["tel"]){
35         $_SESSION["tel"] = $res["tel"];
36     }
37
38
39     $_SESSION["username"] = $res["username"];
40     $_SESSION["name"] = $res["name"];
41     $_SESSION["surname"] = $res["surname"];
42     $_SESSION["email"] = $res["email"];
43     $_SESSION["admin"] = ($res["admin"] == 1 ? true : false);
44     $_SESSION["psw"] = $res["psw"];
45     echo json_encode(array("status" => "success", "message" => "ok"));
46 }
47
48 } else {
49     session_start();
50     session_destroy();
51     echo json_encode(array("status" => "success", "message" => "session destroyed"));
52 }
53
54 ?>
```

La scrittura alle righe 30, 45, 51 permette al codice di rispondere con un messaggio in questo formato:

```
1 {
2     "status": "success",
3     "message": "ok"
4 }
```

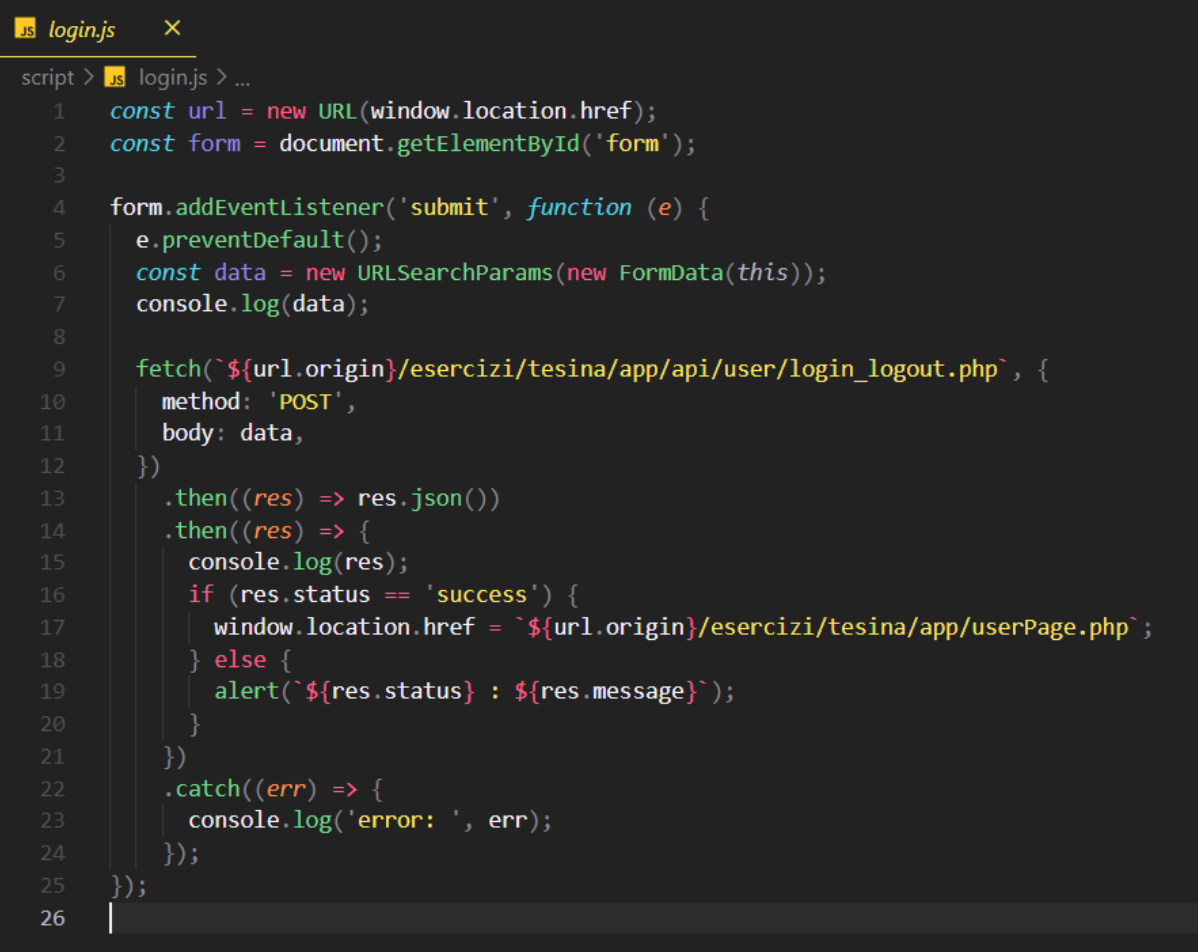
Un altro tipo di risposta possibile per esempio è:

```
1 {
2     "status": "fail",
3     "message": "invalid credentials"
4 }
```

Quando invece un API restituisce un dato o un insieme di dati e non un semplice stato:

```
1  {
2    "status": "success",
3    "data": [
4      {
5        "COUNT(*)": "2",
6        "genre": "rock"
7      },
8      {
9        "COUNT(*)": "1",
10       "genre": "pop"
11      },
12      {
13        "COUNT(*)": "1",
14        "genre": "classic"
15      }
16    ]
17  }
```

I risultati JSON di queste API vengono sfruttati lato client dagli script delle pagine html e php per avere delle reazioni nella pagina html in base alla risposta. Ecco un esempio:



```
login.js
script > JS login.js > ...
1  const url = new URL(window.location.href);
2  const form = document.getElementById('form');
3
4  form.addEventListener('submit', function (e) {
5    e.preventDefault();
6    const data = new URLSearchParams(new FormData(this));
7    console.log(data);
8
9    fetch(`${url.origin}/esercizi/tesina/app/api/user/login_logout.php`, {
10      method: 'POST',
11      body: data,
12    })
13      .then((res) => res.json())
14      .then((res) => {
15        console.log(res);
16        if (res.status == 'success') {
17          window.location.href = `${url.origin}/esercizi/tesina/app/userPage.php`;
18        } else {
19          alert(`${res.status} : ${res.message}`);
20        }
21      })
22      .catch((err) => {
23        console.log('error: ', err);
24      });
25  });
26
```

N.b:

In questo screen e in molte altre parti del mio codice Javascript utilizzo questa sintassi `()=>{}`, questo è solo un 'nuovo' modo di dichiarare le funzioni con la nuova sintassi ES6 di Javascript che si chiama "arrow functions". Infatti scrivere "const funzione = (param)=>{}" è come scrivere "function funzione (param){}". Se la funzione si limita a restituire un valore allora si può direttamente scrivere "`()=>valore`", senza parentesi graffe.

Per chiamare le API lato client con Javascript ci sono diverse librerie come JQuery, axios, superagent (viene usato con nodeJs) e fetch. Ho deciso di usare quest'ultimo per questioni di semplicità di scrittura del codice. Fetch funziona con il meccanismo delle *promise* di JS: se volessi descrivere in un linguaggio parlato il più possibile quello che succede dalla riga 9 in giù direi che fetch effettua una richiesta all'api che è descritta da quell'url, **aspetta** la risposta e la trasforma in un *oggetto/array di oggetti* Javascript (riga 13). Dopodichè **aspetta** che finisca di trasformare il JSON in un oggetto/array di oggetti Javascript e usa quell'oggetto come parametro di una funzione (riga 14).

Questa funzione verifica prima lo status della risposta (

```
1  {  
2    "status": "fail",  
3    "message": "invalid credentials"  
4  }
```

)

e se il login non è andato a buon fine il codice fa comparire un alert in cui si descrive lo status e il messaggio d'errore. Se invece è andato a buon fine il codice esegue un reindirizzamento della pagina alla pagina personale dell'utente.

Se si genera un errore nell'interprete di questo codice Javascript il codice esegue ciò che c'è nel `.catch`, e in poche parole in questo caso stampa solamente l'errore.

Used network protocols:

TCP (Transmission Control Protocol): Is a connection-based protocol that ensures the arrival of the packages to the destination in the correct order.

Not by chance it's main feature is affidability.

SMTP (Simple Mail Transfer Protocol): Simple Mail Transfer Protocol is a standard protocol for sending emails.

HTTP (HyperText Transfer Protocol): Is an application-level protocol that rules how all the sources are transferred by server to all its clients.

It's based on URL (Uniform Resource Locator), basically a string that identifies only one resource/api.

We can apply parameters to the URL to define some variables that the API uses.

Request in http can be many, but main types of request are:

- GET
- POST
- PATCH
- PUT
- DELETE

GET is the most commonly used for calling an API or a resource without encrypting the request (then we use it for transferring resources that aren't secret). It doesn't has body, for the possibility of define parameters in the URL

POST is used for encrypted communication. It has a body and it's content is encrypted.

We in fact use it to send private information or, more in general, for a secure communication. For example with login information sending (password and username).

PUT is typically used for modifying a record in a database. It can be done specify the entire tuple of the database that we want to update (also if we only want update one field of this tuple)

PATCH is typically used for modifying a record in a database. It can be done also by specifying the only one field that we want to update

DELETE is used for deleting record in a database or more in general to delete a resource on server

I will use HTTPS in the future because it's more secure. It's the standard technology for keeping an internet connection secure and safeguarding any sensitive data that is being sent between two systems, preventing criminals from reading and modifying any information transferred, including potential personal details. In this way data breaches are less

Cryptographic functions:

Encrypted functions are pieces of code that allow a resource's access only to the it's addressee.

The resource takes a lot of mathematics operations on itself so that it is illegible or not understandable. Only the other sources that has the private key for decrypt can reverse previous mathematics operations and read the real content.

- Symmetric
- Asymmetric
- **Hash**

The difference between symmetric and asymmetric encryption algorithms is that the symmetric sender and recipient uses the same key (known only for them) for encrypting and decrypting the resource. The asymmetric instead uses a public key and a private key. Public key is often used for encryption and it is known by everyone, private key is known only by the one that must decrypt and read the resource.

In my project I only use a hash cryptographic algorithm.

Hash cryptographic algorithms apply to the resource very complex mathematics functions in such a way as to allow the complete irreversibility of this procedure. For example, I can't have a password in **no way** if I have it's hash encoding.

Basic example:

```
"ciao" => "6e6bc4e49dd477ebc98ef4046c067b5f"  
"6e6bc4e49dd477ebc98ef4046c067b5f" => ??
```

I used hash cryptography in my project mainly for the passwords that are stored in the database and for all the directories, one per user, in which there are the receipts of that user.

This is because I want to hide the path for any receipt for users that don't own it. If the user is logged Javascript automatically generates the path for its receipts' directory including the md5 hashing for the name.

In the future I think I will change the hash algorithm from MD5 to SHA-256 because MD5 is easier to guess with a brute-force algorithm. In Facts MD5 has only 32 bit, while SHA-256 has 256 bit

Es:

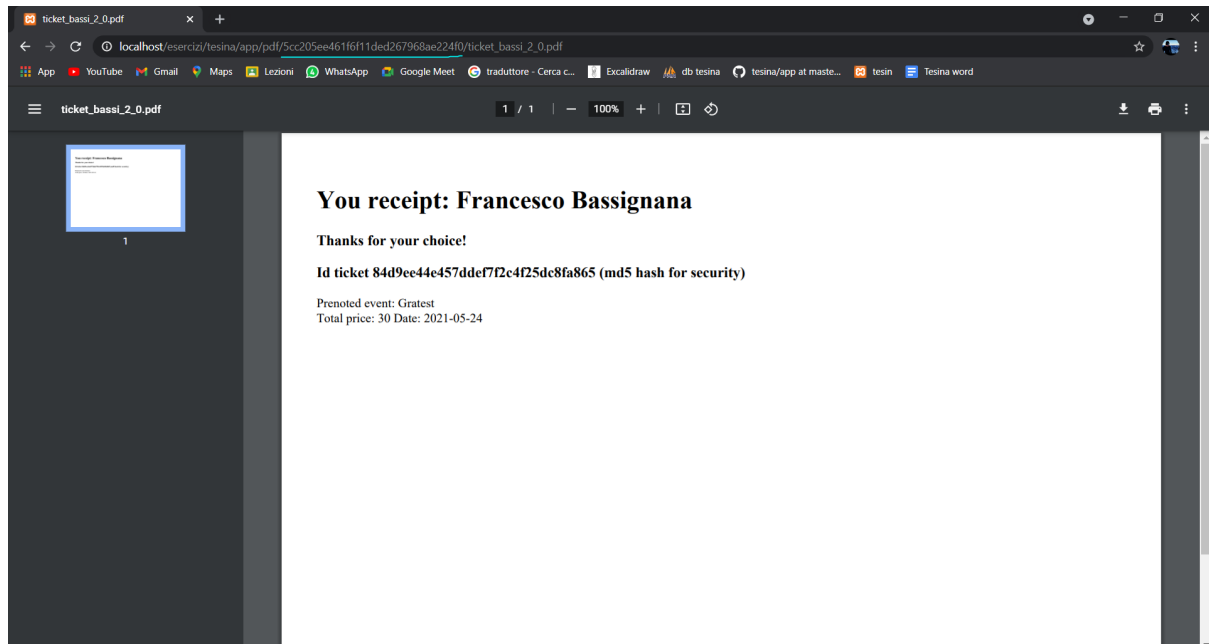
Username: bassi	
Psw: 6e6bc4e49dd4	
Email: bacobas.f@gr	
Name: Francesco	
Surname: Bassignan	
Tel: 3349628407	
Admin: admin	
Aviable_balance: 985	
Delete this user	D

[ticket_bassi_2_0.pdf](#)

[ticket_bassi_2_1.pdf](#)

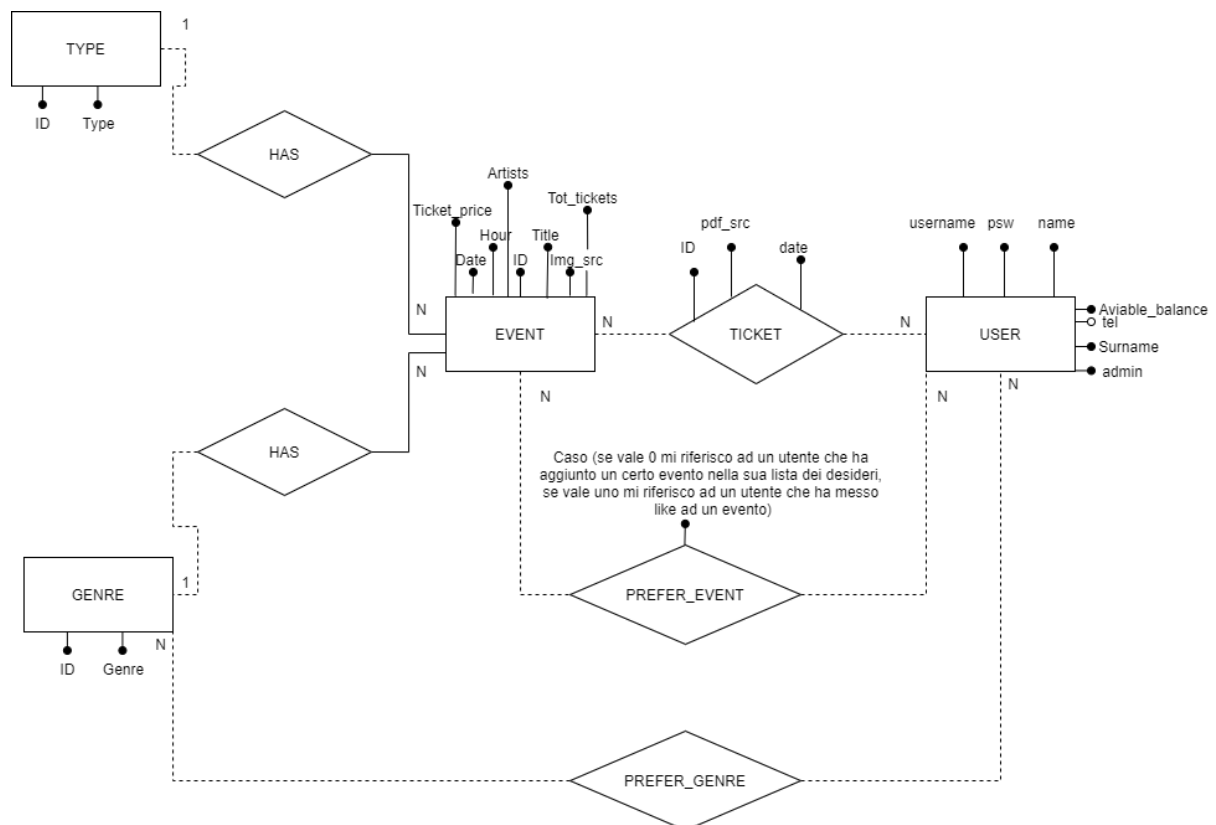
[ticket_bassi_2_2.pdf](#)

If I click on the first link "ticket_bassi_2_0.pdf"



The underlined part is the hash of the username of the user that is in that session.

Schema ER



(il file .drawio è nella cartella del progetto, con un'estensione di visual studio code si può usare un editor apposito)

MAPPING

events(**id**, title, *id_type*, *id_genre*, img_src, date, hour, ticket_price, artists, tot_tickets, discounted, *place_id*)

genres(**id**, genre)

places(**id**, place, city, nation, lat, lng)

prefer_events(**username**, *id_e*, caso)

prefer_genres(**username**, *id_genre*)

tickets(**id**, pdf_src, *id_e*, user, date)

types(**id**, type)

users(**username**, psw, email, name, surname, tel-, admin, available_balance)

CREAZIONE DB

Creazione tabella events:

```
CREATE TABLE `events` (  
  `id` int(4) NOT NULL,  
  `title` varchar(20) NOT NULL,  
  `id_type` int(1) NOT NULL COMMENT '->tabella dei tipi',  
  `id_genre` int(1) NOT NULL COMMENT '->tabella dei generi',  
  `img_src` varchar(255) NOT NULL,  
  `date` date NOT NULL,  
  `hour` char(5) NOT NULL,  
  `ticket_price` float NOT NULL,  
  `artists` varchar(50) NOT NULL,  
  `tot_tickets` int(11) NOT NULL,  
  `discounted` int(1) NOT NULL,  
  `place_id` int(11) NOT NULL  
)
```

Creazione tabella genres:

```
CREATE TABLE `genres` (  
  `id` int(1) NOT NULL,  
  `genre` varchar(15) NOT NULL  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

Creazione tabella places:

```
CREATE TABLE `places` (  
  `id` int(11) NOT NULL,  
  `place` varchar(255) NOT NULL,  
  `city` varchar(255) NOT NULL,  
  `nation` varchar(255) NOT NULL,  
  `lat` double NOT NULL,  
  `lng` double NOT NULL  
)
```

Creazione tabella prefer_events:

```
CREATE TABLE `prefer_events` (  
  `username` varchar(10) NOT NULL,  
  `id_e` int(4) NOT NULL,  
  `caso` int(1) NOT NULL COMMENT 'se è lista desideri o no'  
)
```

Creazione tabella prefer_genres:

```
CREATE TABLE `prefer_genres` (  
  `username` varchar(10) NOT NULL,  
  `id_genre` int(1) NOT NULL  
)
```

Creazione tabella tickets:

```
CREATE TABLE `tickets` (  
  `id` int(11) NOT NULL,  
  `pdf_src` varchar(30) NOT NULL,  
  `id_e` int(11) NOT NULL,  
  `user` varchar(15) NOT NULL,  
  `date` date NOT NULL  
)
```

Creazione tabella types:

```
CREATE TABLE `types` (  
  `id` int(1) NOT NULL,  
  `type` varchar(10) NOT NULL  
)
```

Creazione tabella users:

```
CREATE TABLE `users` (  
  `username` varchar(10) NOT NULL,  
  `psw` char(32) NOT NULL,  
  `email` varchar(25) NOT NULL,  
  `name` varchar(15) NOT NULL,  
  `surname` varchar(15) NOT NULL,  
  `tel` char(10) DEFAULT NULL,  
  `admin` int(1) NOT NULL,  
  `aviable_balance` float UNSIGNED DEFAULT NULL  
)
```

Aggiunta di vincoli e indici:

```
--  
-- Indici per le tabelle `events`  
--  
ALTER TABLE `events`  
  ADD PRIMARY KEY (`id`),  
  ADD KEY `id_genre` (`id_genre`),  
  ADD KEY `id_type` (`id_type`),  
  ADD KEY `author` (`artists`),  
  ADD KEY `place_id` (`place_id`);  
  
--  
-- Indici per le tabelle `genres`  
--  
ALTER TABLE `genres`  
  ADD PRIMARY KEY (`id`);  
  
--  
-- Indici per le tabelle `places`  
--  
ALTER TABLE `places`  
  ADD PRIMARY KEY (`id`),  
  ADD UNIQUE KEY `lat_lng` (`lat`,`lng`);  
  
--  
-- Indici per le tabelle `prefer_events`  
--  
ALTER TABLE `prefer_events`  
  ADD PRIMARY KEY (`username`,`id_e`,`caso`),
```

```
    ADD KEY `id_e` (`id_e`);

--
-- Indici per le tabelle `prefer_genres`
--
ALTER TABLE `prefer_genres`
    ADD PRIMARY KEY (`username`,`id_genre`),
    ADD KEY `id_genre` (`id_genre`);

--
-- Indici per le tabelle `tickets`
--
ALTER TABLE `tickets`
    ADD PRIMARY KEY (`id`),
    ADD KEY `user` (`user`),
    ADD KEY `id_e` (`id_e`);

--
-- Indici per le tabelle `types`
--
ALTER TABLE `types`
    ADD PRIMARY KEY (`id`);

--
-- Indici per le tabelle `users`
--
ALTER TABLE `users`
    ADD PRIMARY KEY (`username`);

--
-- AUTO_INCREMENT per le tabelle scaricate
--
--
-- AUTO_INCREMENT per la tabella `events`
--
ALTER TABLE `events`
    MODIFY `id` int(4) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=15;

--
-- AUTO_INCREMENT per la tabella `genres`
--
ALTER TABLE `genres`
```

```
MODIFY `id` int(1) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=4;

--
-- AUTO_INCREMENT per la tabella `places`
--
ALTER TABLE `places`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=7;

--
-- AUTO_INCREMENT per la tabella `tickets`
--
ALTER TABLE `tickets`
  MODIFY `id` int(11) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=231;

--
-- AUTO_INCREMENT per la tabella `types`
--
ALTER TABLE `types`
  MODIFY `id` int(1) NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=6;

--
-- Limiti per le tabelle scaricate
--
--
-- Limiti per la tabella `events`
--
ALTER TABLE `events`
  ADD CONSTRAINT `events_ibfk_1` FOREIGN KEY (`id_genre`) REFERENCES
`genres` (`id`),
  ADD CONSTRAINT `events_ibfk_2` FOREIGN KEY (`id_type`) REFERENCES
`types` (`id`),
  ADD CONSTRAINT `events_ibfk_3` FOREIGN KEY (`place_id`) REFERENCES
`places` (`id`);

--
-- Limiti per la tabella `prefer_events`
--
ALTER TABLE `prefer_events`
  ADD CONSTRAINT `prefer_events_ibfk_2` FOREIGN KEY (`username`)
REFERENCES `users` (`username`),
  ADD CONSTRAINT `prefer_events_ibfk_3` FOREIGN KEY (`id_e`) REFERENCES
`events` (`id`);
```

```
--  
-- Limiti per la tabella `prefer_genres`  
--  
ALTER TABLE `prefer_genres`  
  ADD CONSTRAINT `prefer_genres_ibfk_1` FOREIGN KEY (`username`) REFERENCES `users` (`username`),  
  ADD CONSTRAINT `prefer_genres_ibfk_2` FOREIGN KEY (`id_genre`) REFERENCES `genres` (`id`);  
  
--  
-- Limiti per la tabella `tickets`  
--  
ALTER TABLE `tickets`  
  ADD CONSTRAINT `tickets_ibfk_3` FOREIGN KEY (`user`) REFERENCES `users` (`username`),  
  ADD CONSTRAINT `tickets_ibfk_4` FOREIGN KEY (`id_e`) REFERENCES `events` (`id`);  
COMMIT;
```

CHECK VARI

Visto che ci sono parecchie problematiche per inserire i check con phpMyAdmin li indicherò qua sotto:

Events:

- date => CHECK date > CURRENT_DATE() + 30 (Si deve pubblicare un evento con almeno circa un mese di anticipo)
- ticket_price => CHECK ticket_price > 0
- tot_tickets => CHECK tot_tickets > 50 (più di 50 biglietti totali)
- discounted => CHECK discounted IN (0, 1)

prefer_events:

- caso => CHECK caso IN (0,1)

tickets:

- date => CHECK date <= CURRENT_DATE() (Anche se un po' inutile poichè il record si crea quando l'utente compra il biglietto)

users:

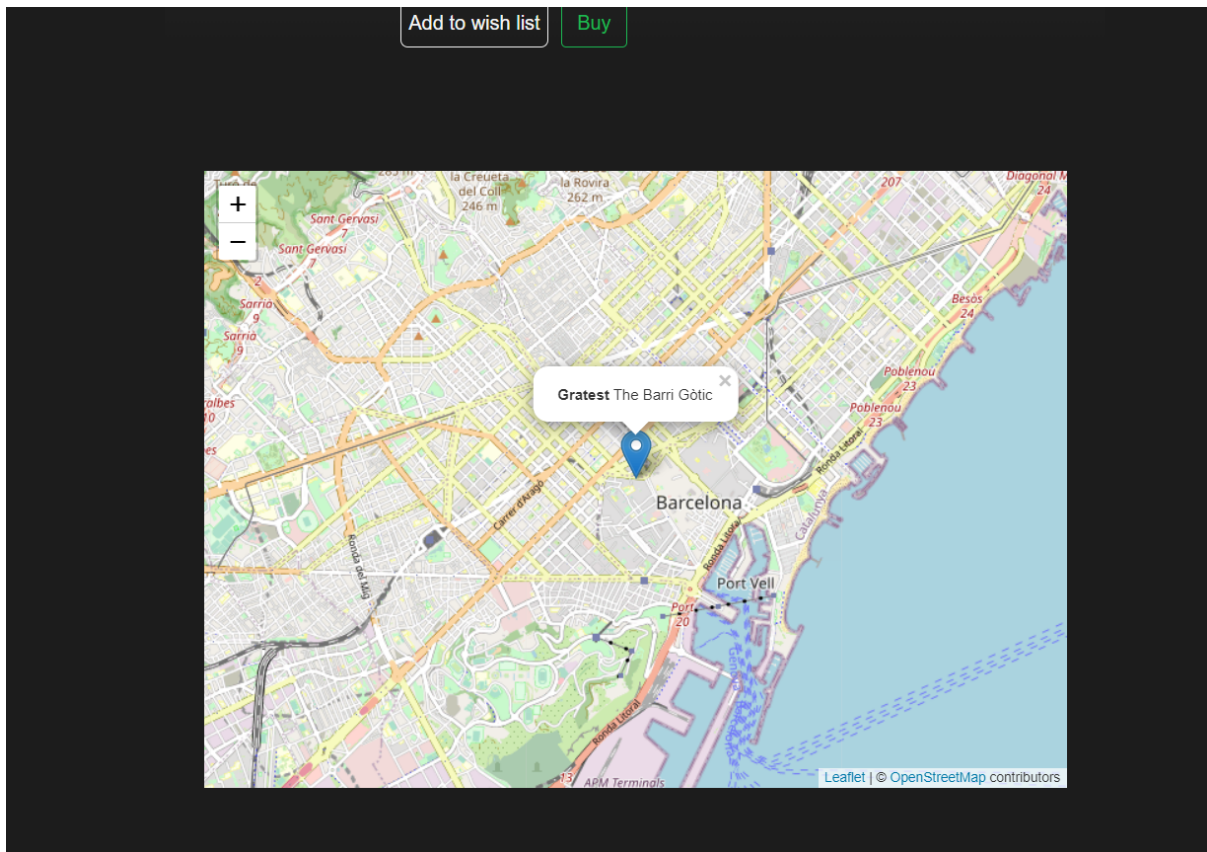
- email => CHECK email LIKE("%@%.%")
- tel => CHECK tel LIKE("_____")
- admin=> CHECK admin IN (0, 1) (0 -> utente normale, 1-> admin)
- aviable_balance => CHECK aviable_balance > 0 (anche questo un po' superfluo perchè ho assegnato la proprietà UNSIGNED)

API esterne usate (Javascript)

- CanvasJS
- Leaflet

Leaflet:

Obiettivo:



Codice sorgente:

```
var map = L.map('mapid').setView([lat, lng], 13);

L.tileLayer(
  'https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png'
).addTo(map);

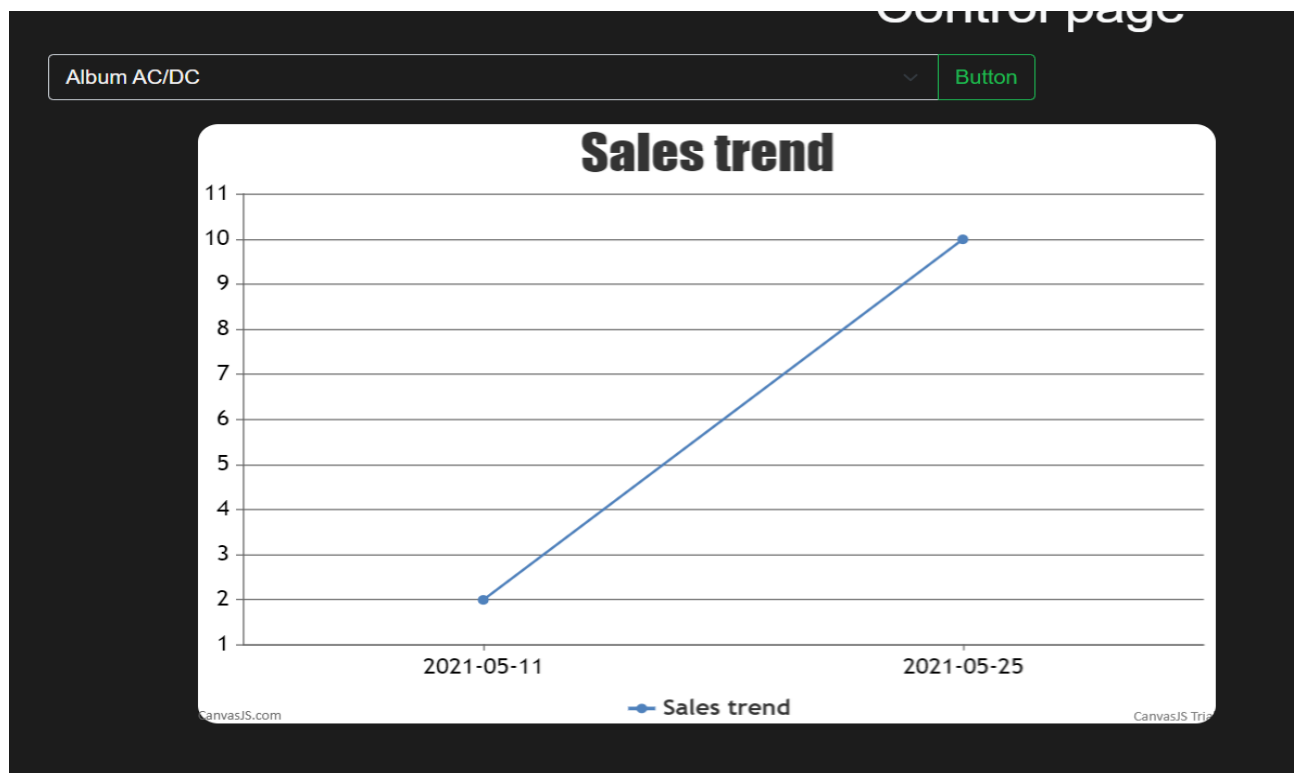
L.marker([lat, lng])
  .addTo(map)
  .bindPopup(`<b>${title}</b>\n${place}`)
  .openPopup();
```

Questo codice istanzia questa mappa in un div che ho chiamato "mapid" e inizializza la posizione alle coordinate *lat* e *lng* (latitudine e longitudine) (*riga 1*), poi sfrutta la API messe a disposizione per disegnare la mappa (righe 3,4,5) e infine aggiunge un marker alla stessa latitudine e longitudine (7,8,9,10).

Ovviamente al posto di *lat* e *lng* uso le coordinate del luogo dove si terrà l'evento prese dall'api che ho create che restituisce le informazioni del singolo evento

CanvasJS:

Obiettivo:



Si vuole dunque generare un grafico fornendo una lista di tuple [(x,y),(x1,y1),(x2,y2),(xn,yn)]

Questo è il pezzo di codice che genera il grafico:

```
var chart = new CanvasJS.Chart('graphic', {
  theme: 'light1', // "light2", "dark1", "dark2"
  animationEnabled: true, // change to true
  title: {
    text: 'Sales trend',
  },
  data: [
    {
      showInLegend: true,
      name: 'Sales trend',
      // Change type to "bar", "area", "spline", "pie", etc.
      type: 'line',
      dataPoints: dataPoints,
    },
  ],
});
chart.render();
```

La parte su cui concentrarsi è dataPoints, come valore si aspetta un oggetto di questo tipo:

```
[
  {label : x1, y: y1},
  {label : x2, y: y2},
  {label : x3, y: y3},
  //ECC...
]
```

Di conseguenza l'obiettivo è quello di avere al posto di ogni x le date in cui sono stati comprati biglietti di quell'evento e al posto delle y la quantità di biglietti acquistati in quelle date. Per questo ho sviluppato un API apposta che ritorna un risultato con questo format:

```
{
  "status": "success",
  "data": [
    {
      "COUNT(*)": "2",
      "date": "2021-05-11"
    },
    {
      "COUNT(*)": "10",
      "date": "2021-05-25"
    }
  ]
}
```

A questo punto non è bastato fare altro che effettuare il parse di questa risposta per trasformarla in un oggetto Javascript, sostituire la chiave “COUNT(*)” con “y” e “date” con “label” e creare l’oggetto Javascript dataPoints assegnandogli quell’array di oggetti:

```
const Res = await fetch(
  `${url.origin}/esercizi/tesina/app/api/ticket/howMany.php?title=${title}`
);
const res = await Res.json();
if (res.status === 'success') {
  res.data.forEach((el, idx) => {
    el['label'] = el.date;
    delete el.date;
    el['y'] = parseInt(el['COUNT(*)']);
    delete el['COUNT(*)'];
  });
  dataPoints = res.data;
}
```

Async/await:

La dicitura che ho utilizzato per chiamare la funzione fetch “await” è un'altra maniera per programmare in maniera asincrona utilizzando le promise. In modo tale che sembri il classico stile di programmazione sequenziale.

Mi spiego meglio: scrivere il codice qua sopra è come scrivere questo codice

```
fetch(`${url.origin}/esercizi/tesina/app/api/ticket/howMany.php?title=${title}`)
  .then((Res) => Res.json())
  .then((res) => {
    if (res.status === 'success') {
      res.data.forEach((el, idx) => {
        el['label'] = el.date;
        delete el.date;
        el['y'] = parseInt(el['COUNT(*)']);
        delete el['COUNT(*)'];
      });
      console.log(res.data);
      dataPoints = res.data;
    }
    ECC... (IL RESTO DEL CODICE CHE UTILIZZA LA RISPOSTA)
  });
```

Solamente che la procedura nel primo caso è possibile solamente nelle funzioni asincrone, infatti la funzione nella quale chiamo la fetch API e creo il grafico è descritta nella seguente maniera:

```
$('#button').click(async () => {
    const title = $('.form-select').val();
    $('#graphic').empty();
    try {
        const Res = await fetch(
`${url.origin}/esercizi/tesina/app/api/ticket/howMany.php?title=${title}`
        );
        ECC...
    });
});
```

N.b: alla dichiarazione della funzione prima delle parentesi c'è scritto "async" per indicare appunto che la funzione è asincrona

Tutto ciò perchè alcune funzioni e alcuni metodi (in realtà molti se si parla di JS) non ritornano un valore proprio, ma ritornano una *promise* (letteralmente "promessa"), quindi per poter accedere ad un valore di una promise bisogna "aspettarlo". E in una funzione sincrona bisogna specificare che la risorsa potrà essere usata solo quando si sarà "aspettato" il valore. Mentre in una funzione asincrona (che essa stessa ritorna una *promise*) è sottinteso che verranno utilizzate le risorse non appena saranno pronte.

So che questo tipo di discorso che introduce i concetti principali della programmazione asincrona può sembrare un po' ostico, mi sono infatti limitato a sintetizzare e ad essere chiaro il più possibile perchè altrimenti ci sarebbero davvero troppe basi da spiegare (Event loop, thread disponibili, ecc...)

Librerie PHP usate

DomPdf

DomPdf è una libreria di PHP che permette di creare degli output e dei file pdf utilizzando il linguaggio HTML come corpo del file. Ecco la funzione nella quale ho usato la libreria:

```
function preparePdf($fileName , $html, $username){

    $domPdf = new Dompdf();

    $domPdf->load_html($html);
    $domPdf->setPaper("A4" , "landscape");
```

```
$domPdf->render();

$file = $domPdf->output();

$fp = fopen("../..pdf/" . md5($username) . "/" . $fileName ,
"a");

fwrite($fp , $file);
fclose($fp);

//file_put_contents($fileName , $file, FILE_USE_INCLUDE_PATH);

return "../..pdf/" . md5($username) . "/" . $fileName;
}
```

Come parametri accetta il nome del file da creare, il corpo HTML del pdf e il nome dell'utente che sta creando il pdf (perchè questo pdf, che è la ricevuta del pagamento, verrà salvato nella cartella che ha lo stesso nome dell'utente hashato tramite MD5) e come valore ritorna il percorso del file pdf appena creato.

PHPMailer

PHPMailer è una libreria di PHP usata per inviare mail da un mittente ad un destinatario attraverso un certo server mail.

```
function sendMail($userFrom, $psw , $to , $subject , $body , $file=
null){

    //Instantiation and passing `true` enables exceptions
    $mail = new PHPMailer(true);

    try {
        //Server settings
        $mail->isSMTP();
        //Send using SMTP

        $mail->Mailer = "smtp";
```

```
        // $mail->SMTPDebug = 1;                                // Enable
verbose debug output
        $mail->Host      = 'smtp.gmail.com';
// Set the SMTP server to send through
        $mail->SMTPAuth  = true;
// Enable SMTP authentication
        $mail->Username  = $userFrom;                            // SMTP
username
        $mail->Password  = $psw;
// SMTP password
        $mail->SMTPSecure = PHPMailer::ENCRYPTION_STARTTLS;
// Enable TLS encryption; `PHPMailer::ENCRYPTION_SMTPS` encouraged
        $mail->Port      = 587;

        $mail->isHTML(true);                                    // Set
email format to HTML

        // Recipients
        $mail->addAddress($to, 'bassi');

        $mail->setFrom($userFrom);
                                // Name is optional
        $mail->From = $userFrom;

        // Attachments
        if($file) $mail->addAttachment($file);                  // Add
attachments
                                // Optional name

        // Content
        $mail->Subject = $subject;
        $mail->Body    = $body;
        $mail->AltBody = 'This is the body in plain text for
non-HTML mail clients';

        $mail->send();
        return true;
    } catch (Exception $e) {
        echo "Mailer Error: {$mail->ErrorInfo}";
        return false;
    }
}
```

```
}
```

Questa è la funzione che ho creato per semplificarne l'utilizzo per il mio elaborato.

Questa mail come parametri ha bisogno della mail del mittente, la password del mittente, la mail del destinatario, l'oggetto della mail, il corpo della mail e il percorso locale del file da allegare (opzionale) e restituisce *true* se la mail è stata inviata correttamente o *false* se c'è stato un errore;

Come si può vedere si devono settare degli attributi all'istanza della classe PHPMailer tra cui la mail del destinatario, la mail del mittente, l'allegato e il corpo della mail (dopo aver specificato che è di tipo html).

Ecco dove uso entrambe le librerie:

```
$receipt = preparePdf($pdfName . ".pdf" , $body, $username);  
  
$email_val = sendMail("tesina.bassi@gmail.com", $mailPsw,  
$_SESSION["email"] , "Payment receipt" , "no-reply" , $receipt);
```

Da come si può intuire qui il codice crea la ricevuta pdf (la variabile *\$body* corrisponde al codice html della ricevuta) e successivamente invio la mail che ha come allegato quel pdf alla mail dell'utente che ha effettuato l'acquisto.