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.

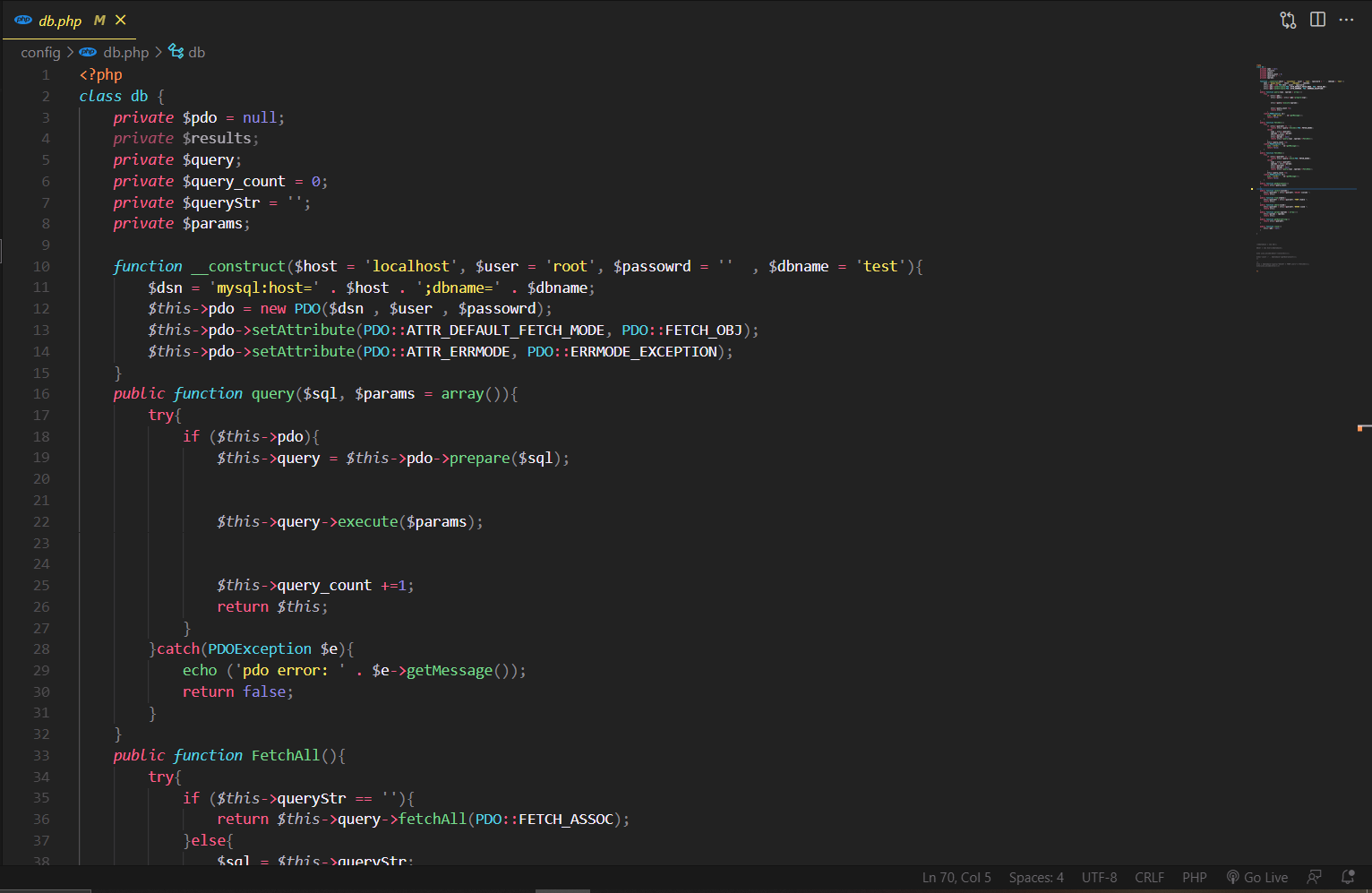
Verranno eliminati tutti i dati dell’utente da tutte le tabelle del database tramite l’api fornita apposta per

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.

Il db.php

Primo sguardo a questa classe:



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>"]);

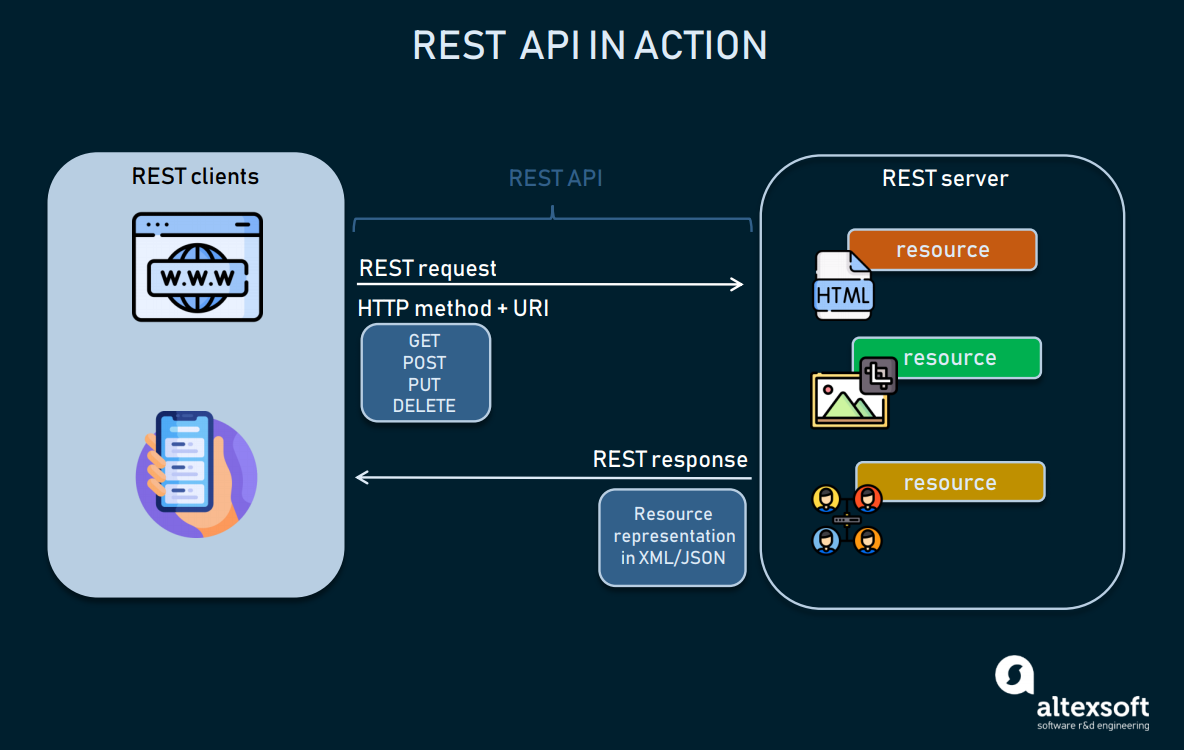
$elenco\_record = $query->fetchAll(PDO::FETCH\_ASSOC);

Mentre invece usando la classe db posso invece scrivere:

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

$res = $conn->query("SELECT \* FROM <TABELLA>" , ["<EVENTUALE ARRAY PARAMETRI>"])->FetchAll();

REST Api App:



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

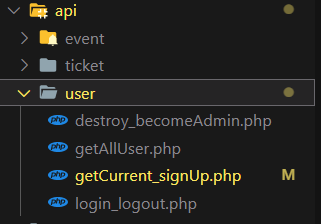
Seguendo questo schema architetturale esiste sempre un client che richiede al server la risorsa e lo stato di quella risorsa. La risorsa può essere in formato JSON, XML, Py, Ecc…  
Ma solitamente viene usato il formato JSON perché è universale per tutti i linguaggi e facilmente leggibile (user-friendly).

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.

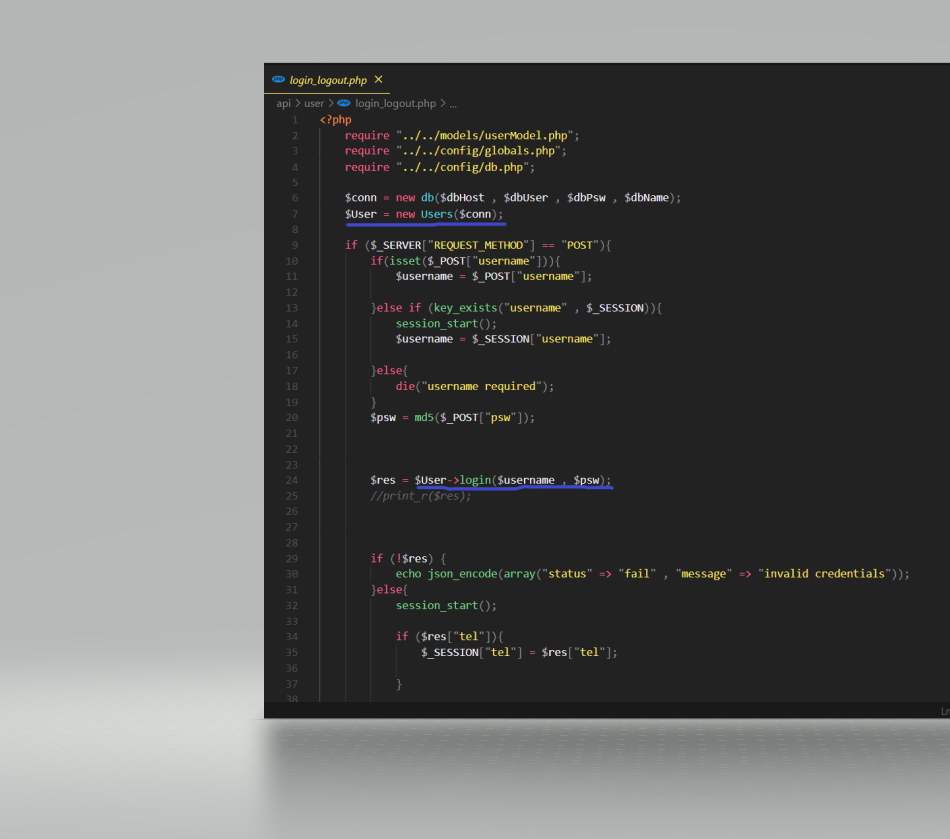
Ognuna di queste entità è suddivisa per ogni entità relazionale nel db.

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.

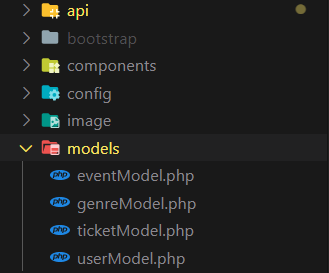


Come si può vedere ci sono diverse cartelle nelle quali sono contenute le api necessarie per ogni entità. Per esempio, se volessi effettuare il login e avere un riscontro in formato JSON di come è andata l’operazione dovrei chiamare l’api con url “localhost/[...]/api/user/login\_logout.php” con metodo POST.  
Invece se chiamo la stessa api col metodo GET effettuerò il logout della sessione.

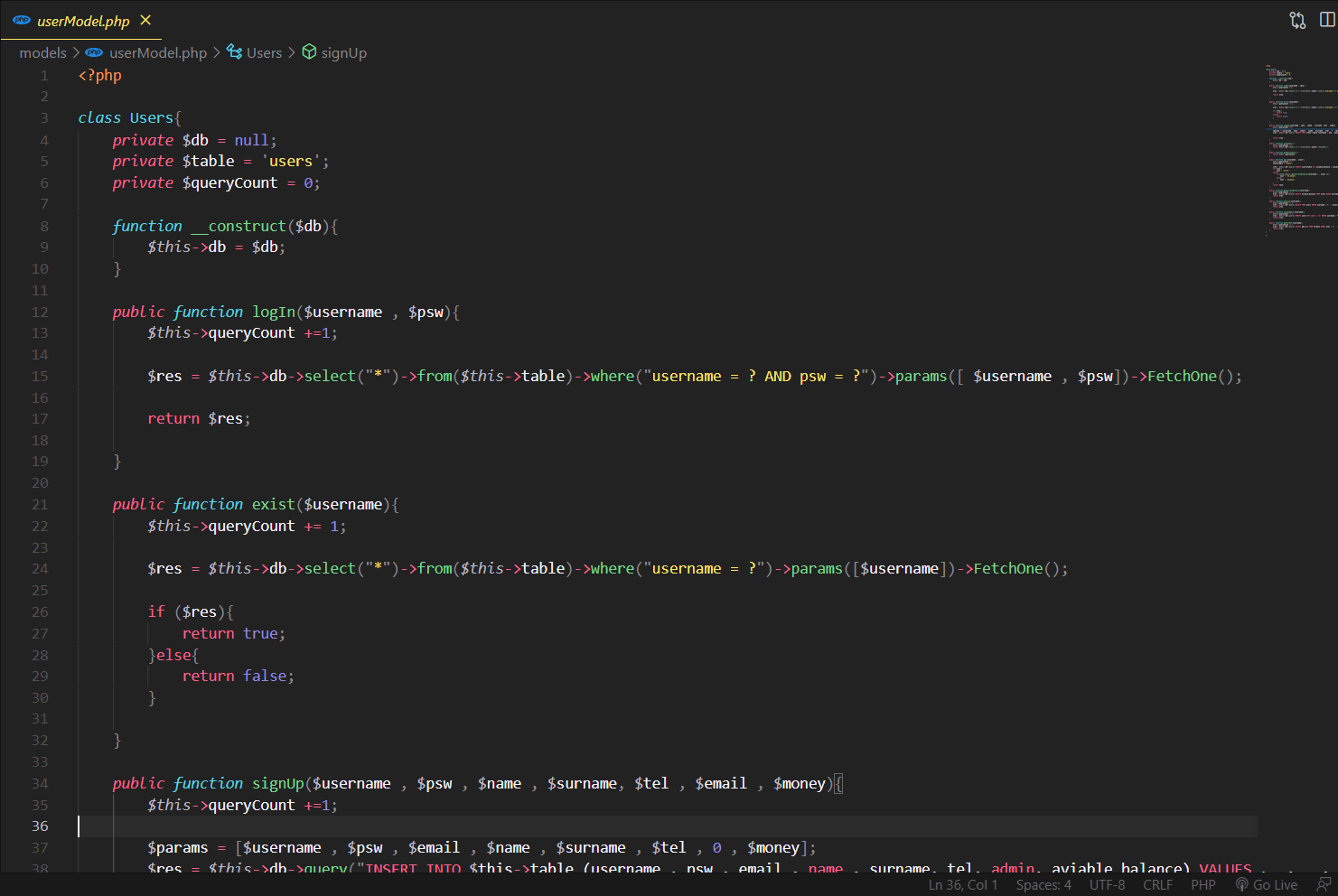
Guardando bene le parti sottolineate...



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”.



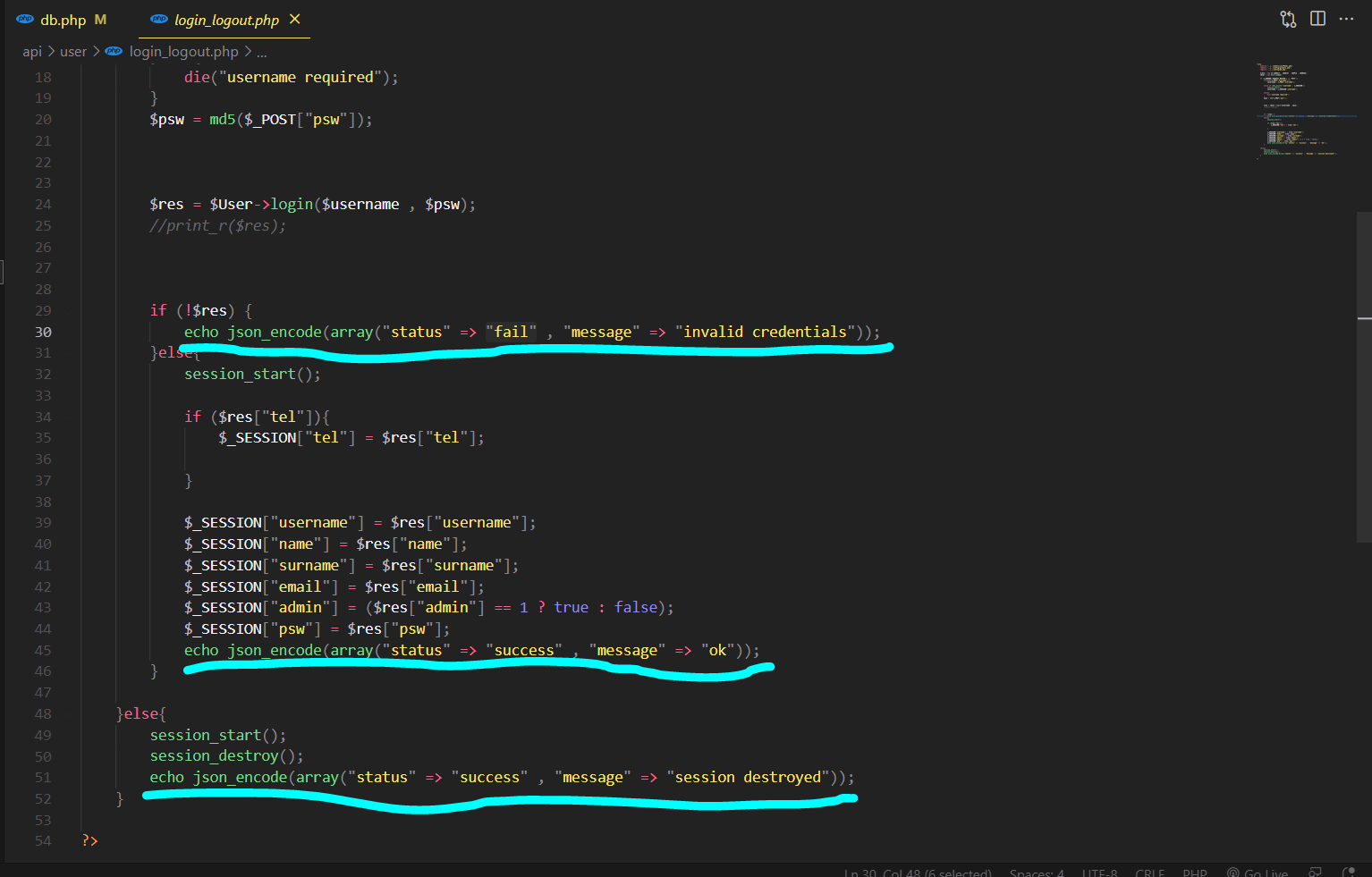
Ecco spiegato il percorso: “..” per spostarsi di un livello indietro (infatti viene usato due volte per tornare alla cartella principale) “/models” per entrare nella cartella models e “userModel.php” per dire da quale file proviene il codice da importare in “login\_logout.php”.



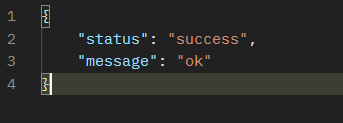
Come si può vedere questa è la parte dei models dedicata alla tabella utenti.  
C’è una classe (Users) che sfrutta un'istanza della classe “db” creata per semplificare la gestione del database col PDO nel “db.php” nella cartella “config”.  
Si possono vedere i primi tre metodi di una lunga serie e uno 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”).

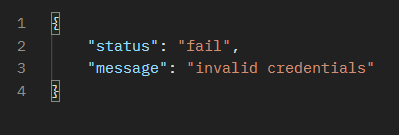
Tornando a “login\_logout.php”:



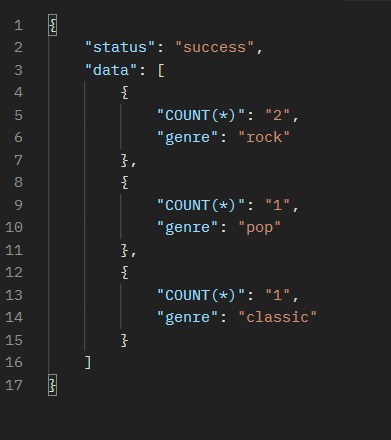
La scrittura alle righe 30, 45, 51 permette al codice di rispondere con un messaggio in questo formato:



Un altro tipo di risposta possibile per esempio è:

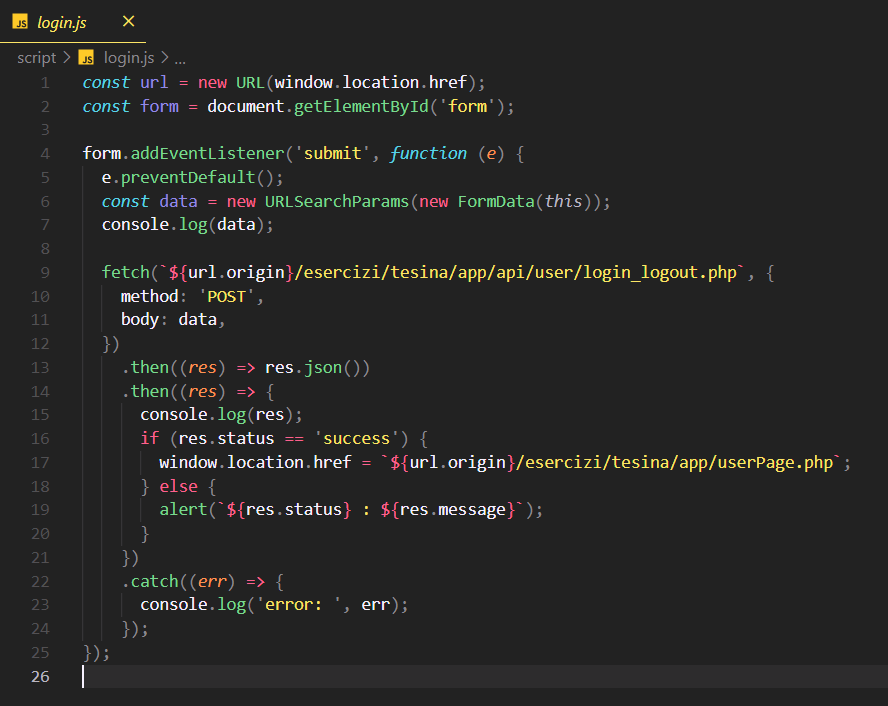


Quando invece un api restituisce un dato o un insieme di dati e non un semplice stato:



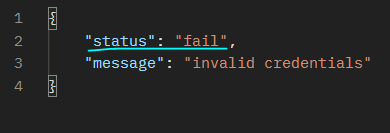
I risultati JSON di queste api vengono sfruttati lato client dagli script delle pagine html e php

per avere delle reazioni nella pagina html in base alla risposta. Ecco un esempio:



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 (



)

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.

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 specify 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

Encrypted function:

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:

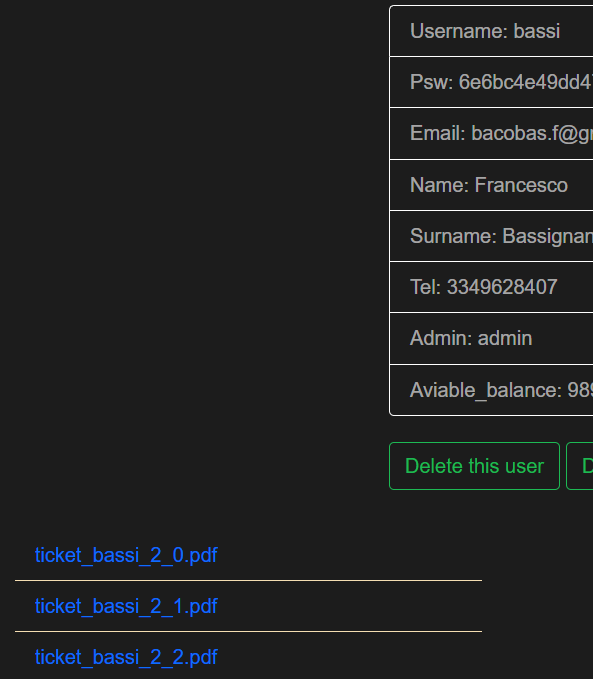
“ciao” => “**6e6bc4e49dd477ebc98ef4046c067b5f”**

“**6e6bc4e49dd477ebc98ef4046c067b5f”** => ??

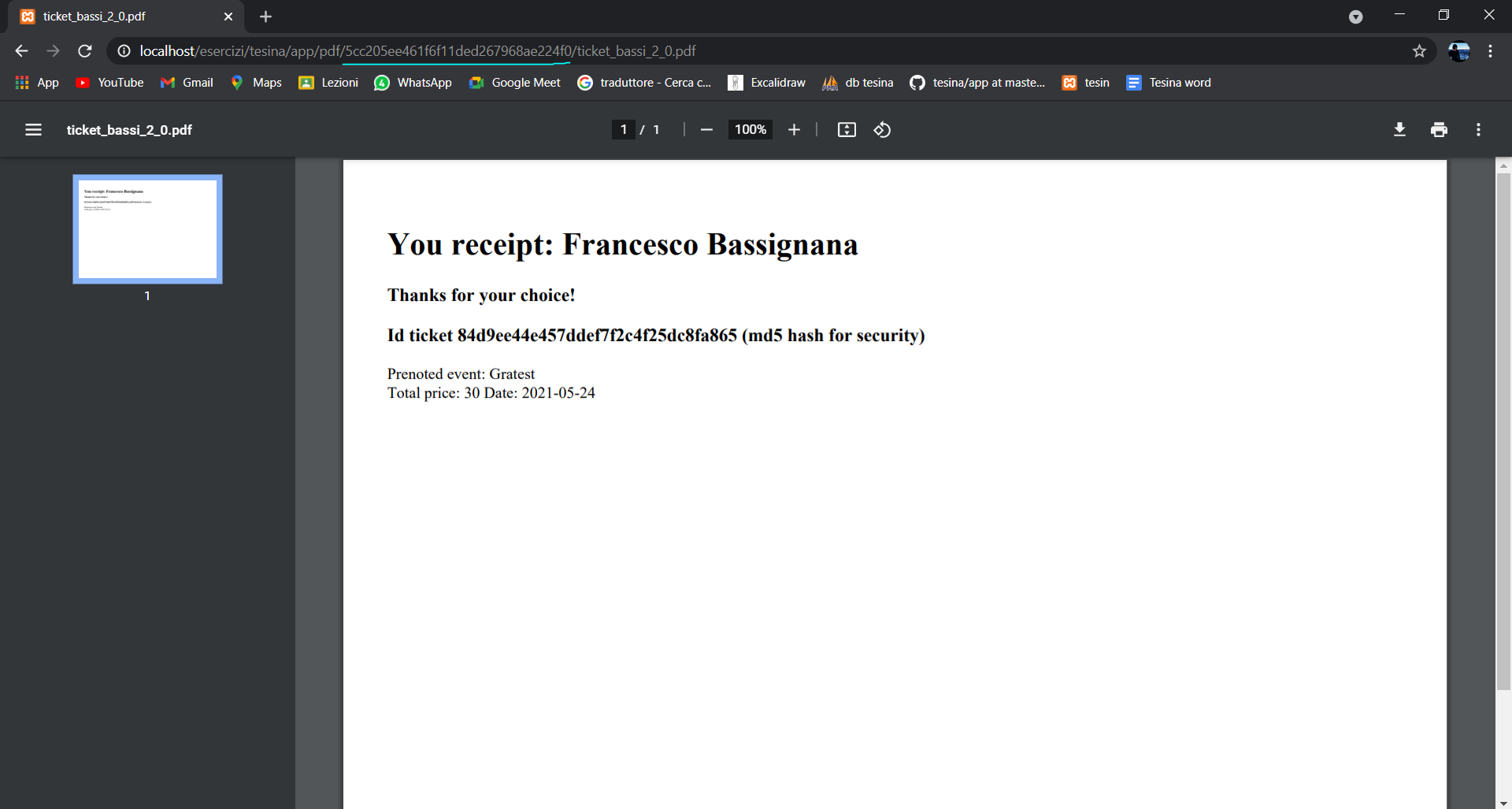
I used hash cryptography in my project mainly for the passwords that are stored in the database and for all the directories, one per user, in which there are the receipts of that user.

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

Es:

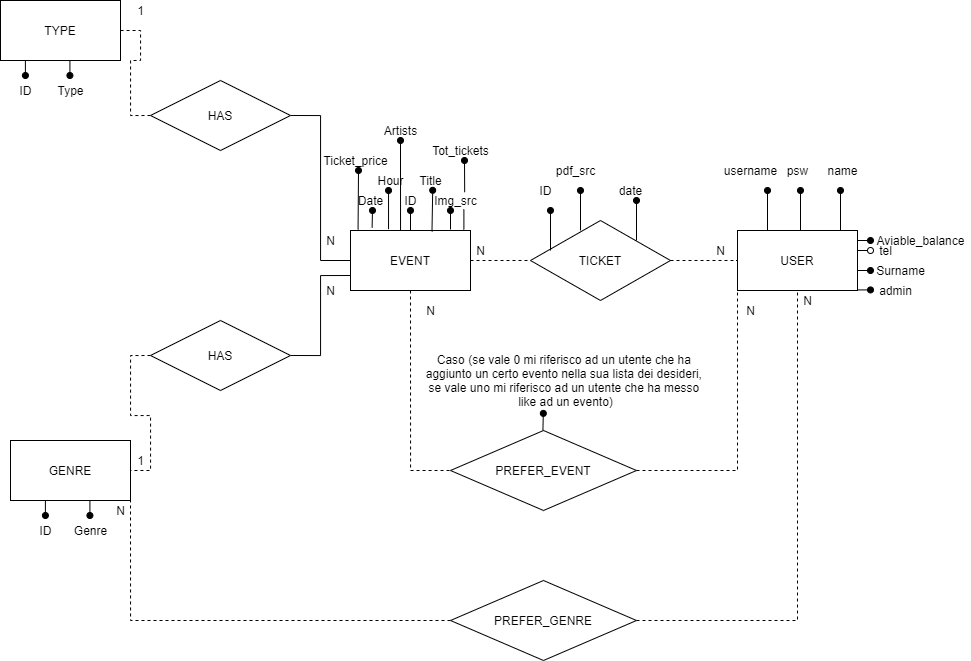


If I click on the first link “ticket\_bassi\_2\_0.pdf”



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, aviable\_balance)

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 (minimo 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)