DOCUMENTAZIONE

PARTE 1

1. creare basi dati CREATE SCHEMA 'RoomArtist'
2. fase di creazione progetto nel framework Laravel:

```
composer create-project laravel/laravel --prefer-dist The_Room_Artist
php artisan config:cache
php artisan serve
```

   A questo punto visitando il link http://127.0.0.1:8000
3. prendiamo il file .env andando a cambiare i attributi:valore, con i rispettivi valori della nostra password e del database
   DB_CONNECTION=mysql
   DB_HOST=127.0.0.1
   DB_PORT=3306
   DB_DATABASE=The_Room_Artist
   DB_USERNAME=root
   DB_PASSWORD=*****
4. comandi per creare il login, ovvero l'autenticazione.
   composer require laravel/ui
   php artisan ui bootstrap --auth
   npm install --save-dev vite laravel-vite-plugin
   npm install --save-dev @vitejs/plugin-vue
   npm install && npm run build
   php artisan migrate
5. andiamo in migration, in particolare nella migrazione che gestisce gli utenti e modifichiamo la funzione up

```
public function up()

{

Schema::create('user', function (Blueprint $table) { //crea tabella user che avrà i seguenti campi

$table->id();

$table->timestamps(); });

}
```

6. Laravel ha già inserito nei due metodi le azioni necessarie a creare una tabella con le colonne che ritiene necessarie o utili ($table->id() per l'id univoco di ogni riga, $table->timestamps() per le colonne created_at e updated_at, che sono sempre 2 colonne aggiuntive che vengono aggiunte...
   Infine, $timestamps indica se la tabella presenta i campi predefiniti di Laravel created_at e updated_at che vengono usati come riferimento cronologico nelle operazioni CRUD.
   Questo comando creerà la classe Product che estende la classe base Model di Laravel. In questa fase la classe creata è vuota, ossia è presente solo la sua struttura base.

7. abbiamo quindi fin ora nel database le tabelle: migrazioni, che tiene traccia delle migrazioni fatte, una tabella degli utenti e una tabella per resettare le password

8. andiamo a creare un seader ovvero una funzione che riempie il database dei dati inziali, permettendo di popolare la base dati...

UTILIZZANDO IL COMANDO, **nome del modello al plurale + Seeder,** lo troviamo nella directory database/Seeder

php artisan make:seeder UsersSeeder

php artisan make:seeder AdminSeeder

php artisan make:seeder RoomsTableSeeder

php artisan db:seed (per caricare tutti i seed)

oppure

[Comando per singoli seed, non tutti:

php artisan db:seed --class=UsersTableSeeder,

php artisan db:seed --class=AdminSeeder

php artisan db:seed --class=RoomTableSeeder

]

9. specifico nella funzione run di UsersSeeder i campi che voglio immettere creare seeder per le tabelle admin, users e rooms:

abbiamo inserito i seguenti dati nella funzione run() dei vari file creati **in app/database/seeders/*.php:**

```
// ADMINS
User::create([
        'name'        => 'Solomon',
        'surname'     => 'Taiwo',
        'email'       => 'solomontaiwo@theartistroom.com',
        'password'    => bcrypt('password'),
        'updated_at'  => date('Y-m-d h:i:s'),
        'created_at'  => date('Y-m-d h:i:s'),
        'is_admin'    => true,
    ]);
User::create([
        'name'        => 'Gaia',
        'surname'     => 'Marzola',
        'email'       => 'gaiamarzola@theartistroom.com',
        'password'    => bcrypt('password'),
        'updated_at'  => date('Y-m-d h:i:s'),
        'created_at'  => date('Y-m-d h:i:s'),
```

```
        'is_admin'     => true,
    ]);

User::create([
        'name'         => 'Giorgia',
        'surname'      => 'Pirelli',
        'email'        => 'giorgiapirelli@theartistroom.com',
        'password'     => bcrypt('password'),
        'updated_at'    => date('Y-m-d h:i:s'),
        'created_at'    => date('Y-m-d h:i:s'),
        'is_admin'     => true,
    ]);

User::create([
        'name'         => 'Luca',
        'surname'      => 'Gaudenzi',
        'email'        => 'lucagaudenzi@theartistroom.com',
        'password'     => bcrypt('password'),
        'updated_at'    => date('Y-m-d h:i:s'),
        'created_at'    => date('Y-m-d h:i:s'),
        'is_admin'     => true,
    ]);

// USERS

DB::table('users')->insert([[
        'name'         => 'Pino',
        'surname'      => 'Pinoli',
        'email'        => 'pinopinoli@theartistroom.com',
        'password'     => bcrypt('password'),
        'updated_at'    => date('Y-m-d h:i:s'),
        'created_at'    => date('Y-m-d h:i:s'),
    ], [
        'name'         => 'Matteo',
```

```
        'surname'      => 'Solo',
        'email'        => 'matteosolo@theartistroom.com',
        'password'     => bcrypt('password'),
        'updated_at'    => date('Y-m-d h:i:s'),
        'created_at'    => date('Y-m-d h:i:s'),
    ], [
        'name'         => 'Harry',
        'surname'      => 'Potter',
        'email'        => 'harrypotter@theartistroom.com',
        'password'     => bcrypt('password'),
        'updated_at'    => date('Y-m-d h:i:s'),
        'created_at'    => date('Y-m-d h:i:s'),
    ], [
        'name'         => 'Ryu',
        'surname'      => 'Hayabusa',
        'email'        => 'ryuhayabusa@theartistroom.com',
        'password'     => bcrypt('password'),
        'updated_at'    => date('Y-m-d h:i:s'),
        'created_at'    => date('Y-m-d h:i:s'),
    ]]);
```

// ROOMS

```
DB::table('rooms')->insert([[
        'name'         => 'Aula Pirelloni',
        'description'   => ‘scultura’,
        'address'      => 'Via Dei Peracottari, 15',
        'size'         => 100, // metri quadri
        'seats'        => 50,
        'updated_at'    => date('Y-m-d h:i:s'),
        'created_at'    => date('Y-m-d h:i:s')
    ], [
        'name'         => 'Aula Gaudemagna',
        'description'   => ‘pittura o disegno’,
        'address'      => 'Via Frassina, 51',
```

```
        'size'        => 20, // metri quadri
        'seats'       => 50,
        'updated_at'    => date('Y-m-d h:i:s'),
        'created_at'    => date('Y-m-d h:i:s')
    ], [
        'name'        => 'Aula Taiwani',
        'description'   =>'fotografia',
        'address'      => 'Via dei Marnoni, 2',
        'size'        => 150, // metri quadri
        'seats'       => 50,
        'updated_at'    => date('Y-m-d h:i:s'),
        'created_at'    => date('Y-m-d h:i:s')
    ], [
        'name'        => 'Aula Marzoletti',
        'description'   =>'scultura',
        'address'      => 'Via Edmondo De Amicis, 66',
        'size'        => 150, // metri quadri
        'seats'       => 50,
        'updated_at'    => date('Y-m-d h:i:s'),
        'created_at'    => date('Y-m-d h:i:s')
    ]]);
```

opzione aggiuntiva

Per vedere se i dati sono stati correttamente inseriti nel database, usare il seguente comando in Mysql Workbench:

```
SELECT * FROM the_artist_room.rooms;
SELECT * FROM the_artist_room.users;
```

Ecc.

[Usare il seguente comando per pulire e ricreare le tabelle: php artisan migrate:fresh]

10. **passiamo in resouces/views/layouts/app.blade.php**
in cui includiamo il bootstrap, aggiungendo nel file (riga 20)

```
<!-- Style -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2HN"
crossorigin="anonymous">
```

PARTE 2: PROCESSO DI SCAFFOLDING

(MODELLO + MIGRAZIONE + CONTROLLER)

php artisan make:model <nome_model_singolare> -m -c --resource

Lo scaffolding è una procedura automatizzata per la creazione delle funzionalità CRUD (Create,

Read, Update, Delete) base di un elemento. Con lo scaffolding è possibile creare, in un colpo solo:

1. un modello con le relative proprietà nel database

2. un controller per la gestione del modello

3. le action base di un sistema CRUD per interagire con il modello

In altre parole, lo scaffolding permette di generare in pochi secondi un'interfaccia completamente funzionante per la gestione di un oggetto (ad esempio un catalogo prodotti). È una soluzione eccellente come punto di partenza per lo sviluppo

Come abbiamo applicato questo concetto?

Almeno una entità con CRUD completo --> abbiamo implementato la gestione delle prenotazioni...

1. Creazione modelli con migrazione e controllo per utenti, stanze e prenotazioni con i seguenti comandi: **nome del modello al singolare**

   php artisan make:model User -m -c --resource

   php artisan make:model Room -m -c --resource

   php artisan make:model Booking -m -c –resource

2. e poi modificare opportunamente i seguenti file appena creati con gli attributi rilevanti:

   /database/migrations/*_create_bookings_table.php
   /database/migrations/*_create_rooms_table.php
   /database/migrations/*_create_users_table.php
   Modificando le funzioni up(): abbiamo inserito i campi per le Prenotazioni, per le stanze e per gli Utenti

3. Provare ad avviare il progetto con il comando
   php artisan serve

PARTE 3: GESTIONE ROTTE

4. Andiamo in **routes/web.php...** e definiamo le rotte. È possibile definire gli URL della propria applicazione con l'aiuto di percorsi. Questi percorsi possono contenere dati variabili e connettersi ai nostri controller. Con php artisan route:list, abbiamo la lista delle rotte
5. Andiamo in **app/Http/Controllers,** definendo i controller
6. Andiamo a creare le cartelle **in resources/views,** in cui definiamo le cartelle: user, rooms, bookings, who-we-are, la pagina home e la barra di navigazione.

VUE.JS E CSS3: PARTE GESTIONE SITO CON TECNOLOGIE AGGIUNTIVE

Vue.js

è un framework JavaScript per la costruzione di interfacce utente. Si basa su standard HTML, CSS e JavaScript e fornisce un modello di programmazione dichiarativo e basato su componenti che ti aiuta a sviluppare in modo efficiente interfacce utente, siano esse semplici o complesse.

Uno dei pregi di Vue.js è proprio la leggibilità del codice, **infatti {{ mostra.descrizione }}** mi permette di accedere all'attributo descrizione di mostra.

1. Per lavorare con Vue.js è necessario istanziare un oggetto **Vue** (una istanza di Vue globale), grazie al costruttore esposto dal framework, cui passiamo una serie di parametri che permettono di configurare il comportamento della nostra app. Bisogna anzitutto creare una istanza di Vue. Possiamo passare al suo costruttore un oggetto che ci permette di definire alcuni parametri dell'applicazione, come dati e comportamenti, nel nostro caso:

Parametro	**Descrizione**
el	il riferimento all'elemento div definito nel documento HTML
data	Definisce un set di dati

Appena l'oggetto viene creato, Vue acquisisce il div **con id=app** e si occupa di sostituire il con i valori corrispondenti alla descrizione delle mostre.

`{{ mostra.nome }}`	`{{ mostra.descrizione }}`
Riga 59	Riga 60

```
new Vue({
            el: '#app'
```

2. Abbiamo definito la variabile mostre, questa sorta di struct, o tipo di collezioni di campi di tipo diverso.
3. I cui campi verranno visualizzati mediante il ciclo for definito a riga 56: **v-for="(mostra, index) in mostre".**
4. **v-show="mostra.mostraDettagli"** rappresenta una proprietà dati Vue booleana con "true" o "false" come valore della proprietà. Se è 'true' viene mostrato il tag div, mentre se è 'false' il tag non viene mostrato. Essa viene legata all'evento click, in riga 62,(**@click="toggleDettagli(index)")** il quale rende true tale proprietà dati e quindi mostra la descrizione della mostra.
5. Quindi la descrizione della mostra sarà inizialmente nascosta. Quando il pulsante **"Mostra Dettagli"** viene cliccato, Vue.js aggiorna dinamicamente il modello e mostra o nasconde la descrizione.

Animazioni aggiunte:

1. Abbiamo scelto di usare le animazioni offerte da CSS3.
   L'animazione consente di far cambiare gradualmente stile ad un elemento della pagina.
   In particolare, per ciascun cambiamento di stile bisogna specificare un keyframe.
   Il keyframe determina il tipo di stile da applicare all'elemento ad un determinato istante temporale, e fondamentalmente rappresenta il "codice" di esecuzione dell'animazione.
2. Abbiamo usato la funzione CSS che riposiziona un elemento nelle direzioni orizzontale e/o verticale, ovvero transalte(); questa trasformazione è caratterizzata da un vettore bidimensionale [tx, ty]. Le sue coordinate definiscono quanto l'elemento si muove in ciascuna direzione.

3. È anche possibile utilizzare la percentuale. Utilizzando la percentuale, puoi aggiungere tutte le modifiche di stile che desideri. Il nostro esempio ci dice che la funzione CSS verrà applicata alll'elemento quando l'animazione è completa al 25%, al 50% e ancora quando l'animazione è completa al 100% e così via...
I fotogrammi chiave ci aiutano a contenere gli stili che l'elemento avrà in determinati momenti.
4. L'associazione tra l'elemento a cui vogliamo applicare la funzione lo facciamo nel foglio di stile. Quindi definiamo la classe dell'elemento da animare, **<h2 class="funky-title">Prossime Mostre</h2>** e poi nello **<style> .funky-title { animation: bounce 2s infinite; },** viene definita ogni 2 secondi e eseguita all'infinito.

JQUERY

Per usare jQuery abbiamo richiamato la libreria ufficiale nella sezione **<head>** del documento html.

In resources/views/home.blade.php:

Abbiamo aggiunto animazioni sia per le citazioni che per le card, in cui si descrivono le tipologie delle stanze.

Tipo di animazioni:

1. gli elementi vengono prima nascosti con il **metodo hide();**
2. dissolvenza in entrata con **fadeIn().**

```
<script>
  $(document).ready(function() {
    $('#quote-text').hide().fadeIn(2000);
  });
</script>
```

Lo script seleziona gli elementi con classe: quote-text e card, a cui applica le animazioni

```
<script>
  $(document).ready(function() {
    $('.card').hide().fadeIn(1500);
  });
</script>
```

Stesse animazioni definite anche in **/resources/views/who-are-we/index.blade.php**

Piccole precisazioni sull'uso di alcune direttive
asset() usato per generare un URL per una risorsa dell'applicazione.
Footer e navbar che è presente in tutte le pagine lo troviamo in: **resources/views/layouts/app.blade.php** Ecco perché in ogni file in cui vogliamo sia il footer, sia la nav-bar andiamo a inserire **@extends('layouts.app')**
Per il formato dalle date: <html lang="{{ str_replace('_', '-', app()->getLocale()) }}"> sta dicendo ai browser che la lingua per il documento corrente è "en" o qualsiasi altra lingua tu abbia impostato come locale (config/app.php). La lingua predefinita per la tua applicazione è memorizzata nel config/app.phpfile di configurazione. È possibile modificare questo valore per adattarlo alle esigenze della propria applicazione.
Laravel genera automaticamente un "token" CSRF per ogni sessione utente attiva gestita dall'applicazione. Questo token viene utilizzato per verificare che l'utente autenticato sia la persona che effettivamente effettua le richieste all'applicazione. Poiché questo token è archiviato nella sessione dell'utente e cambia ogni volta che la sessione viene rigenerata, un'applicazione dannosa non è in grado di accedervi.

L'annotazione **@guest** in Laravel viene utilizzata per verificare se l'utente corrente è un ospite (non autenticato). È spesso utilizzata all'interno di direttive di controllo del flusso per mostrare o nascondere determinati contenuti a seconda dello stato di autenticazione dell'utente.
Questa istruzione utilizza Carbon per analizzare la data di arrivo ($booking->arrival_date) e quindi formattarla secondo il formato desiderato **(nel caso di esempio, 'd/m/Y').**
Questione validazione: nei controller

Come abbiamo svolto la possibilità di visualizzare "Rieccoti", una volta che l'utente si logga.

1. Auth::check() chiama Auth::user(), ne ottiene il risultato e quindi controlla se l'utente esiste. Infatti Auth::user() recupera l'utente attualmente autenticato...
2. Auth carica il record quando l'utente è autenticato;
3. Per determinare se l'utente che effettua la richiesta HTTP in entrata è autenticato, è possibile utilizzare il metodo-check. Questo metodo restituirà true se l'utente è autenticato;

Come abbiamo gestito la questione delle citazioni? Esse cambiano dinamicamente.

1. Abbiamo aggiunto un modello nuovo: **Quote.php** e nel controller Home abbiamo aggiunto la funzione per generare casualmente la citazione

```
public function index()
{
// Per avere una citazione casuale in home
$quote = Quote::inRandomOrder()->first(); //passa alla variabile quote il valore casuale, la variabile quote poi viene usata nella home per visualizzarla
return view('home', compact('quote'));
}
```

> **Quote::inRandomOrder()**: Questo metodo ordina casualmente le righe della tabella "quotes". In altre parole, recupera i record in un ordine casuale ogni volta che viene eseguita la query.
> **->first()**: Una volta che le righe sono state ordinate casualmente, questo metodo restituisce il primo record risultante. Poiché le righe sono in un ordine casuale, otterrai un record casuale ogni volta che esegui questa query.

2. **create quote table**, abbiamo creato una migrazione per le citazioni, la cui tabella nel Database è stata popolata da **/seeders/QuotesTableSeeder.php**, per popolare la struttura dati, con le varie citazioni
3. la frase che è gia stata decisa dal controller della home ogni volta che la home è caricata

abbiamo aggiunto anche il Middleware: in app/Http/Middleware/AdminMiddleware.php

descrizione del Middleware

1. l middleware fornisce un meccanismo pratico per ispezionare e filtrare le richieste HTTP in ingresso nell'applicazione
   php artisan make:middleware AdminMiddleware
2. La funzione fornita è parte di un middleware in un'applicazione Laravel. Questo middleware, chiamato AdminMiddleware, controlla se l'utente autenticato è un amministratore.

Ecco una spiegazione più dettagliata della funzione:

- La funzione handle è il punto di ingresso del middleware. Prende due parametri: $request, che rappresenta la richiesta HTTP in arrivo, e $next, che è una chiusura (closure) rappresentante il middleware successivo o l'handler finale della richiesta.
- La funzione inizia controllando se c'è un utente autenticato usando auth()->check(). Se c'è un utente autenticato, controlla se la proprietà is_admin dell'utente è impostata su true.
- Se l'utente è un amministratore, il middleware consente alla richiesta di proseguire chiamando $next($request). In caso contrario, se l'utente non è un amministratore, reindirizza l'utente alla pagina principale (home) utilizzando redirect()->route('home') e include un messaggio di errore tramite with('error', 'Non autorizzato.').
- In sostanza, questo middleware garantisce che solo gli utenti autenticati con privilegi di amministratore possano accedere a determinate rotte o controller. Se un utente non amministratore tenta di accedere a tali rotte, verrà reindirizzato alla pagina principale con un messaggio di errore.

Non si può procedere alla prenotazione se l'utente non ha inserito tutti i campi o se ha inserito un numero di posti sbagliato!!!!

1. Abbiamo aggiunto la possibilità di non procedere alla prenotazione se l'utente inserisce un numero di posti superiore alla capienza dell'aula
2. Controlla se il numero di posti disponibili nella stanza ($room->available_seats) è inferiore al numero di persone richiesto ($request->input('people')).

3. Se la condizione è vera (cioè se non ci sono abbastanza posti disponibili), viene eseguito un reindirizzamento alla pagina precedente.
4. Con withInput(), i valori di input della richiesta vengono inclusi nel reindirizzamento. Questo significa che quando l'utente viene reindirizzato alla pagina precedente, i campi del form conterranno ancora i valori che l'utente ha inserito, semplificando così la correzione degli errori.

resources/views/rooms/create.blade.php troviamo **da riga 60:**

lo script per rendere il pulsante di creazione non cliccabile se tutti gli altri campi non sono compilati

In breve, quando viene inserito del testo in uno qualsiasi degli input all'interno del form con l'id createRoomForm, il codice verifica se tutti gli input obbligatori ([required]) sono stati compilati. Se almeno uno di questi input è vuoto, il bottone con l'id createRoomButton viene disabilitato. Altrimenti, se tutti gli input obbligatori sono compilati, il bottone viene abilitato.

Come abbiamo gestito la questione transazioni database?

Troviamo in riga 56-84-98 di **app/Http/Controllers/BookingController.php**

Una transazione sul database è un insieme di operazioni che puoi eseguire in modo sicuro all'interno della struttura del database della tua applicazione, come ad esempio le query SQL per modificare i dati (es. aggiornamenti, cancellazioni e inserimenti)

- crea una transazione: utilizza il **DB::beginTransaction();**comando per avviare una transazione.
- Rollback di una transazione: utilizza il **DB::rollBack();**comando se desideri apportare modifiche o annullare azioni.
- Effettua una transazione: se tutto è andato come previsto, utilizza il **DB::commit();**comando.

La funzione getRoomInfo($id), che troviamo in app/Http/Controllers/RoomController.php

1. **$room = Room::findOrFail($id);:** Utilizza il modello Eloquent Room per cercare una stanza nel database con l'ID specificato. Se la stanza non viene trovata, viene generata un'eccezione di tipo ModelNotFoundException.

2. **return response()->json([...]);:** Restituisce una risposta JSON. In questo caso, viene restituito un array JSON contenente le informazioni sulla stanza.
3. In sintesi, questa funzione accetta l'ID di una stanza, cerca la stanza nel database, e restituisce le informazioni sulla stanza sotto forma di risposta JSON, includendo il numero di posti disponibili.

// Funzione per il pulsante di promozione utente normale ad admin

```
public function promoteToAdmin(User $user)
{
    $user->update(['is_admin' => true]);
    return response()->json([
        'success' => true,
        'data' => $user,
    ], 200);
}
```

Restituisce una risposta JSON indicando che l'operazione di promozione è stata completata con successo.

1. 'success' => true,: Indica che l'operazione è avvenuta con successo.
2. 'data' => $user,: Restituisce l'oggetto dell'utente aggiornato come parte dei dati nella risposta JSON. Questo può essere utile per ottenere ulteriori dettagli sull'utente dopo la promozione.
3. In Laravel, JSON (JavaScript Object Notation) è comunemente utilizzato per gestire dati strutturati e trasmetterli tra il back-end e il front-end o tra diversi servizi.

app/Providers/AppServiceProvider.php

Abbiamo aggiunto direttive blade personalizzate, in modo tale da renderle visibili solo quando l'utente loggato è admin.

due direttive personalizzate per Blade, **chiamate @admin e @endadmin**. Le direttive Blade personalizzate ti consentono di estendere la sintassi dei template Blade con nuove istruzioni personalizzate.

1. **Blade::directive('admin', function () { ... }):** Questo codice registra una nuova direttiva Blade chiamata @admin. Quando questa direttiva è utilizzata nei tuoi template Blade, il suo contenuto sarà eseguito solo se l'utente è autenticato (auth()->check()) e se l'utente attuale ha il ruolo di amministratore (auth()->user()->is_admin restituisce true).
2. **Blade::directive('endadmin', function () { ... }):** Questo codice registra una direttiva Blade chiamata @endadmin, che segna la fine del blocco controllato dalla direttiva @admin.
3. resources/views/layouts/app.blade.php TROVIAMO QUESTE DIRETTIVE APPLICATE ALLE PAGINE

In **public/images** troviamo le immagini del nostro progetto

Nel contesto di Laravel, la cartella **resources/views/auth** contiene i file di vista specifici per le operazioni di autenticazione dell'applicazione. Questi file di vista vengono utilizzati per mostrare le pagine associate alle azioni di registrazione, accesso (login), reset della password e altre operazioni correlate all'autenticazione degli utenti.

1. Login.blade.php: La vista per la pagina di accesso (login).
2. register.blade.php: La vista per la pagina di registrazione.
3. verify.blade.php: La vista per la pagina di verifica dell'indirizzo email (usata quando l'opzione di verifica dell'email è attivata).
4. passwords/email.blade.php: La vista per la pagina di invio del link di reset della password.

5. passwords/reset.blade.php: La vista per la pagina di reimpostazione della password.

Troviamo le richieste AJAX nei file:

resources/views/rooms/edit.blade.php

resources/views/bookings/edit.blade.php

resources/views/bookings/create.blade.php

resources/views/users/index.blade.php

script per determinare i posti disponibili nella stanza

Ecco una spiegazione più dettagliata del codice:

1. $(document).ready(function() { ... });: Questo assicura che il codice jQuery all'interno di essa verrà eseguito solo quando il documento HTML è completamente caricato.
2. fetchRoomInfo();: Questo chiama la funzione fetchRoomInfo() al caricamento della pagina per ottenere inizialmente le informazioni sull'aula selezionata.
3. $('#room_id').change(function() { ... });: Questa parte del codice attacca un gestore di eventi all'elemento con l'ID room_id. Quando il valore di questo elemento viene cambiato (ad esempio, quando l'utente seleziona un'aula diversa), viene richiamata la funzione fetchRoomInfo() per ottenere le nuove informazioni dell'aula.
4. function fetchRoomInfo() { ... }: Questa funzione è responsabile di effettuare la richiesta AJAX per ottenere le informazioni sull'aula selezionata. Utilizza l'ID dell'aula selezionata, invia una richiesta GET a /api/rooms/{id}, e gestisce le risposte di successo e di errore.
5. Nel blocco success: function(response) { ... }, viene aggiornato l'elemento con l'ID availableSeatsInfo con i dati ottenuti dalla risposta della richiesta AJAX. In questo caso, mostra il numero di posti disponibili nell'aula selezionata.
6. Nel blocco error: function(error) { ... }, viene gestito l'errore nel caso in cui la richiesta AJAX non abbia successo. In tal caso, viene stampato un messaggio di errore nella console e l'elemento availableSeatsInfo viene svuotato.

```
// Codice per promuovere un utente ad admin in riga 79 del file resources/views/users/index.blade.php
```

Eliminazione di un utente:

1. Quando un elemento con la classe .btn.btn-elimina viene cliccato, viene innescata una funzione che impedisce l'azione predefinita e richiede la conferma dell'utente.
2. Se l'utente conferma, viene effettuata una richiesta AJAX di tipo DELETE all'URL /user/{id}, dove {id} è l'identificatore dell'utente da eliminare.
3. Viene inviato il token CSRF ('_token': token) insieme alla richiesta.
4. In caso di successo, viene mostrato un messaggio di successo nel log e la pagina viene ricaricata (location.reload()).
5. In caso di errore, viene mostrato un messaggio di errore nel log.

Promozione di un utente ad amministratore:

1. Quando un elemento con la classe .btn-promuovi viene cliccato, viene innescata una funzione che richiede la conferma dell'utente.
2. Se l'utente conferma, viene effettuata una richiesta AJAX di tipo PUT all'URL /users/{id}/promote, dove {id} è l'identificatore dell'utente da promuovere ad amministratore.

3. Viene inviato il token CSRF (_token: token) insieme alla richiesta.
4. In caso di successo, viene verificato se la risposta contiene il flag success. Se è true, la pagina viene ricaricata (location.reload()).
5. In caso di errore, viene mostrato un messaggio di errore nel log.

event.preventDefault();: Impedisce il comportamento predefinito associato all'evento. Ad esempio, se questo codice è all'interno di un gestore di eventi di un pulsante in un modulo, questa riga impedisce il sottoporsi del modulo.

let id = $(this).attr('data-id');: Recupera l'attributo 'data-id' dell'elemento che ha scatenato l'evento. Questo potrebbe essere l'ID dell'utente che si sta per cancellare.

let token = $('input[name="_token"]').val();: Recupera il valore del campo di input con il nome '_token'. Questo valore potrebbe essere utilizzato come token di autenticazione per confermare l'autenticità della richiesta.

var confirmation = window.confirm("Sei sicuro di voler cancellare questo utente?");: Visualizza una finestra di conferma nel browser con un messaggio. Se l'utente fa clic su "OK", la variabile confirmation sarà true, altrimenti sarà false. Questo può essere utilizzato per gestire la conferma dell'utente prima di procedere con l'azione, come la cancellazione dell'utente.

Riga 45 di resources/views/bookings/edit.blade.php

Abbiamo supposto che l'orario in cui si può prenotare una stanza sia dalle 12.00 in poi, in modo tale da aver tempo di lasciar posto all'artista che l'ha occupata prima di liberare, anche egli entro le 12.00. Quindi gli orari sono predefiniti per implementare questa logica.

Spiegazione sempre nello stesso file di funzione **updateDepartureDateOptions**

Ecco una spiegazione più dettagliata delle funzionalità:

1. updateDepartureDateOptions: Questa funzione è responsabile di aggiornare le opzioni di data di partenza in base alla data di arrivo selezionata. Utilizza un ciclo for per aggiungere opzioni per le due date successive alla data di arrivo. La funzione formatta le date e le aggiunge all'elemento con l'id formatted_departure_date. Inoltre, formatta la data per il campo hidden departure_date e imposta il suo valore.
2. $('#arrival_date').change(function() { ... });: Questo evento viene innescato quando la data di arrivo cambia. Quando ciò accade, chiama la funzione updateDepartureDateOptions per aggiornare dinamicamente le opzioni di data di partenza.
3. updateDepartureDateOptions();: Questa riga chiama la funzione updateDepartureDateOptions durante il caricamento della pagina per assicurarsi che le opzioni di data di partenza siano aggiornate inizialmente.
4. Complessivamente, il codice assicura che le opzioni di data di partenza si aggiornino automaticamente in base alla data di arrivo selezionata nell'interfaccia utente del form.

resources/views/bookings/edit.blade.php nella funzione updateBookingButton:

L'obiettivo principale sembra essere quello di disabilitare il pulsante di prenotazione se il numero di persone che vuole occupare l'aula è superiore al numero di posti disponibili o se mancano informazioni essenziali nei campi del modulo.

1. Raccoglie i valori dai campi del modulo come l'ID dell'aula (room_id), il numero di persone (people), la data e l'orario di arrivo e partenza.
2. Effettua una richiesta AJAX per ottenere i posti disponibili per l'aula specificata.
3. Disabilita il pulsante di prenotazione se il numero di persone supera i posti disponibili o se mancano informazioni essenziali.

4. L'evento input sull'elemento con ID people attiva la funzione updateBookingButton quando il numero di persone viene inserito. Gli eventi change su altri elementi come room_id, arrival_date, ecc., attivano la funzione quando vengono inseriti input in quei campi.
5. L'ultima riga, updateBookingButton();, chiama la funzione una volta al caricamento della pagina per assicurarsi che lo stato del pulsante di prenotazione sia corretto inizialmente.

Nota:

L'URL della richiesta AJAX sembra essere basato su una convenzione di routing, presumibilmente verso un endpoint API che fornisce informazioni sull'aula, compresi i posti disponibili.