Event APP

Introduction

This document provides a detailed explanation of the Login component in an Angular project. The component is responsible for handling user login functionality. The code snippet discussed in this documentation focuses on the onSubmit() method within the Login component. The code snippet demonstrates the process of submitting a login form, interacting with an authentication service, and performing actions based on the response.

Project Setup

Before diving into the code explanation, ensure that the project is set up correctly by following these steps:

- Install the necessary dependencies by running the command npm install.
- Launch the development server by executing the command ng serve.

Modules Overview

The Angular project consists of several modules, which are briefly described below:

Auth Module

This module handles authentication-related functionalities.
Features:

- Login: Manages user login functionality.
- Sign Up: Provides user registration functionality.
- Auth Service: Contains methods for user authentication.

Event Module

This module deals with event-related functionalities.

Features:

- Event List: Displays a list of events.
- Event Edit: Enables editing of event details.
- Event Create: Allows the creation of new events.
- Event Calendar: Provides a calendar view of events.

User_layout Module

This module focuses on user-specific layouts and functionalities.

Features:

- Event List for User: Shows a customized event list for the user.
- Register Event: Allows users to register for events.

Login Component

The Login component plays a crucial role in the authentication process. The onSubmit() method, found within this component, handles the form submission and subsequent actions based on the login response. Let's examine the code in more detail:

```
onSubmit() {
  let data = {
    email: this.formg.value.email,
    password: this.formg.value.password,
  };

  this.auth.login(data).subscribe(
    (res: any) => {
      this.user = res.user;
      if (this.user.role_type === 'admin') {
        localStorage.setItem('admin', this.user.role_type);
        localStorage.setItem('user_id', this.user.id);
        this.router.navigate(['events/layout/eventList']);
        localStorage.setItem('username', this.user.first_name);
        this.snackBar.open('Admin Login successful', 'Close', {
          duration: 3000,
        });
      } else {
        this.router.navigate(['userlayout']);
        localStorage.clear();
        localStorage.setItem('user_id', this.user.id);
        localStorage.setItem('username', this.user.first_name);
        this.snackBar.open('user Login success', 'Close', { duration: 3000 });
      }
    },
```

```
    (err: any) => {
      console.log(err);
      this.snackBar.open('Login failed', 'Close', { duration: 3000 });
    }
  );
}
```

Code Explanation

The onSubmit() method in the Login component facilitates the form submission and subsequent login actions. Below is a step-by-step breakdown of the code:

- **Creation of the data object:** The method begins by creating an object called data, which captures the values entered in the email and password fields of the login form.
- **Authentication service invocation:** The this.auth.login(data) function is invoked, assuming it is a method provided by the authentication service. It takes the data object as an argument and returns an observable that can be subscribed to.
- **Subscription to the authentication observable:** The subscribe() function is called on the returned observable. It accepts two callback functions: one for handling the successful response (res) and the other for handling errors (err).
- **Successful response handling:** In the success callback function (res: any) => { ... }, the response object res is received. It is assumed that the response contains a property named user.
- **Admin user handling:**
    - The code checks if the role_type property of the user object is equal to 'admin'. If it is, the following actions are performed:
    - The 'admin' value is stored in the localStorage with the key 'admin'.
    - The 'user_id' value is stored in the localStorage using this.user.id.
    - The Angular router navigates to the ['events/layout/eventList'] route.
    - The 'username' value is stored in the localStorage using this.user.first_name.
    - A snack bar message is displayed indicating a successful admin login.
    - Regular user handling: If the role_type is not 'admin', it assumes the user is a regular user and executes the following actions:
    - The Angular router navigates to the ['userlayout'] route.
    - The localStorage is cleared.
    - The 'user_id' value is stored in the localStorage using this.user.id.
    - The 'username' value is stored in the localStorage using this.user.first_name.
    - A snack bar message is displayed indicating a successful user login.
    - Error handling: In the error callback function (err: any) => { ... }, any errors that occur during the login process are logged to the console, and a snack bar message indicating a login failure is displayed.

```
onSubmit(): void {
  const data = {
    firstname: this.formg.value.firstname,
    lastname: this.formg.value.lastname,
    username: this.formg.value.username,
    email: this.formg.value.email,
    password: this.formg.value.password,
    age: this.formg.value.age,
    nationality: this.formg.value.nationality,
    role_type: this.formg.value.role,
  };

  this.auth.register(data).subscribe(
    (res: any) => {
      this.user = res.user;
      this.router.navigate(['auth/login']);
      this.snackBar.open('Registration successful', 'Close', {
        duration: 3000,
      });
    },
    (err: any) => {
      console.log(err);
      this.snackBar.open('Registration failed', 'Close', { duration: 3000 });
    }
  );
}}
```

Code Explanation

The onSubmit() method in the Registration component facilitates the submission of a registration form and subsequent registration actions. Let's examine the code in detail:

Form Data Preparation

The onSubmit() method starts by creating an object called data which captures the values entered in the registration form fields. The properties of the data object include:

- firstname: The value from the firstname field of the form.
- lastname: The value from the lastname field of the form.
- username: The value from the username field of the form.
- email: The value from the email field of the form.
- password: The value from the password field of the form.
- age: The value from the age field of the form.
- nationality: The value from the nationality field of the form.
- role_type: The value from the role field of the form.

Registration Service Invocation

The code invokes the register(data) function provided by the auth service. It passes the data object as an argument to the function. The register(data) function is assumed to return an observable that can be subscribed to.

Subscription to the Registration Observable

The subscribe() function is called on the observable returned by the register(data) function. It accepts two callback functions: one for handling the successful response (res) and the other for handling errors (err).

Successful Response Handling

In the success callback function (res: any) => { ... }, the response object res is received. It is assumed that the response contains a property named user.

- The user object is assigned to the this.user property of the component.
- The Angular router navigates to the ['auth/login'] route, which redirects the user to the login page.
- A snack bar message is displayed indicating a successful registration.

Event List

```
  constructor(
    private auth: AuthService,
    private router: Router,
    private snackBar: MatSnackBar
  ) {}
  url: string = '../assets/img1.jpg';
  public isAdmin: any;
  public events: any = [];
  imageChange(event: any) {
    this.url = event.target.src;
  }
  ngOnInit(): void {
    this.getEvents();
    this.isAdmin = localStorage.getItem('admin');
  }
  getEvents() {
    this.auth.getEvents().subscribe(
      (res: any) => {
        this.events = res;
        console.log(this.events);
      },
      (err: any) => {}
    );
  }
  edit(item: any) {
    this.router.navigate(['events/layout/edit/' +
item.id], {
      queryParams: { id: item.id },
    });
  }
  del(item: any) {
    this.auth.deleteEvent(item.id).subscribe(
      (res: any) => {
```

```
          this.snackBar.open('event deleted successfully', 'Close', {
            duration: 3000,
          });
          this.getEvents();
        },
        (err: any) => {
          console.log('error in deleting');
          this.snackBar.open('event not deleted', 'Close', { duration: 3000 });
        }
      );
    }

    registerEvent(event: any): void {
      this.auth.registerEvent(event).subscribe(
        (response: any) => {
          // Handle the success response, e.g., show a success message
          console.log('Event registered successfully');
          this.router.navigate(['eventList']);
        },
        (error: any) => {
          // Handle the error response, e.g., show an error message
          console.error('Failed to register event:', error);
        }
      );
    }
```

This code snippet can be **summarized** as follows:

- The constructor initializes the **AuthService**, Router, and **MatSnackBar** dependencies.
- The **url** variable is set to the path of an image.
- The **isAdmin** variable is used to store the value of the 'admin' item in the **localStorage**.
- The events variable is initialized as an empty array.
- The **imageChange**() function is triggered when the image changes and updates the **url** variable.
- In the **ngOnInit**() method, the **getEvents**() function is called to retrieve the events, and the value of isAdmin is obtained from the localStorage.
- The **getEvents**() function retrieves the events by calling the **getEvents**() method from the **auth** service and assigns the response to the events variable.
- The **edit**() function navigates to the edit page for a specific event.
- The **del**() function deletes an event by calling the **deleteEvent**() method from the **auth** service and then refreshing the event list.

The **registerEvent**() function registers a new event by calling the **registerEvent**() method from the **auth** service and handles the success and error responses accordingly.

Event CRUD

The provided code snippets can be summarized as follows:

Code Snippet 1:

Defines form fields and initializes a **FormGroup** object using **FormBuilder**.
Retrieves the item ID from the query parameters and fetches the corresponding item data using the ID.
Updates the form fields with the fetched data.
Implements the **formatDate**() function to format dates.
Handles the form submission by sending an update request to the server using the **updateEvent**() method from the auth service.

Code Snippet 2:

Declares an array variable events and a boolean variable notfound.
Retrieves a list of events from the server using the **getRegEvents**() method from the eventService.
Populates the events array with the received data and checks if the array is empty.
If no events are found, sets the notfound variable to true; otherwise, sets it to false.

Code Snippet 3:

Declares a FormGroup object and an object variable user.
Initializes the form fields using FormBuilder.
Handles form submission by sending a create event request to the server using the **createEvent**() method from the auth service.

Laravel Event Code:

```php
<?php


namespace App\Http\Controllers;

use App\Models\Event;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;
use Validator;
  class EventsController extends Controller
{
    public function index()
    {
        $events = Event::all();

        return response()->json($events);
    }

    public function eventsCalendar()
    {
        $events = DB::table('reg_events')->get();
        $events =  DB::table('reg_events')
        ->join('users', 'reg_events.user_id', '=', 'users.id')
        ->select('reg_events.*', 'users.*')
        ->get();

        return response()->json(['message' => 'Event registered
successfully', 'events' => $events], 201);
    }


    public function registerEvent(Request $request){
```

```
        $validator = Validator::make($request->all(), [
            'event_title' => 'required',
             'event_description' => 'required',
             'start_date'  => 'required',
             'end_date'  => 'required',
             'user_id'  => 'required',

         ]);

         if ($validator->fails()) {
             return response()->json($validator->errors()->toJson(), 400);
         }
            $events = DB::table('reg_events')->insert($request->all());

        return response()->json(['message' => 'Event registered successfully'], 200);


        // $event = $request->only('id', 'event_title', 'event_description', 'start_date', 'end_date', 'user_id');

        // // Fetch the email from the `users` table based on the user_id using a join
        // $email = DB::table('reg_events')
        //     ->join('users', 'reg_events.user_id', '=', 'users.id')
        //     ->where('reg_events.user_id', $event['user_id'])
        //     ->value('users.email');

        // // Add the fetched email to the event data
        // $event['email'] = $email;

        // // Insert the event data into the `reg_events` table using DB
```

```
        // DB::table('reg_events')->insert($event);

        return response()->json(['message' => 'Event registered
successfully'], 201);
        // $event = $request->only('id', 'event_title',
'event_description', 'start_date', 'end_date', 'email');

        // // Insert the event data into the `reg_events` table
using DB
        // DB::table('reg_events')->insert($event);

        // return response()->json(['message' => 'Event
registered successfully'], 201);
    }
    public function store(Request $request)
    {
        $data = $request->validate([
            'event_title' => 'required',
            'event_description' => 'required',
            'event_image' => 'required',
            'event_created' => 'required',
            'start_date' => 'required',
            'end_date' => 'required',
            'event_badge' => 'required',
            'event_ticket' => 'required',
            'waiting_list' => 'required',
            'event_status' => 'required',
            'price' => 'required',
         ]);

        $event = Event::create($data);
        return response()->json(['id' => $event->id], 201);
    }

    public function show($id)
    {
```

```
        $event = Event::findOrFail($id);
        return response()->json($event);
    }

    public function update(Request $request, $id)
    {
        $data = $request->validate([
            'event_title' => 'required',
            'event_description' => 'required',
            'event_image' => 'required',
            'event_created' => 'required',
            'start_date' => 'required',
            'end_date' => 'required',
            'event_badge' => 'required',
            'event_ticket' => 'required',
            'waiting_list' => 'required',
            'event_status' => 'required',
            'price' => 'required',
        ]);

        $event = DB::table('events')->where('id', $id)-
>update($data);

        return response()->json(['message' => 'Event updated
successfully']);
    }

    public function destroy($id)
    {
        $event = Event::findOrFail($id);
        $event->delete();

        return response()->json(['message' => 'Event deleted
successfully']);
    }
}
```

CodeExplanation:

The provided code snippet is a PHP code segment from an EventsController class. Here's a
The index method retrieves all events from the Event model and returns them as a JSON response.
The eventsCalendar method fetches events and user data from the database using a join operation and returns the result as a JSON response.
The registerEvent method handles the registration of events. It validates the incoming request data and inserts it into the reg_events table using DB. It returns a JSON response with a success message.
The store method validates the request data and creates a new event record in the Event model. It returns a JSON response with the ID of the created event.
The show method retrieves a specific event based on its ID and returns it as a JSON response.
The update method validates the request data and updates the corresponding event in the events table based on the provided ID. It returns a JSON response with a success message.
The destroy method deletes a specific event based on its ID from the Event model. It returns a JSON response with a success message.

User Controller:

```php
<?php


namespace App\Http\Controllers;

use App\Models\user;
use Illuminate\Database\Eloquent\Relations\BelongsTo;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Validation\Rule;
use Illuminate\Support\Facades\Auth;

class UserController extends Controller
{
    /**
     * Display a listing of the resource.
     *
```

```
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
    }
    public function create()
    {
        return view('register');
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    // public function store(Request $request) //this is the
post route for the register form  in the register.blade.php
    // {
    //     // $formFields=$request->validate([
    //     //     'firstname'=>['required'],
    //     //     'lastname'=>['required'],
    //     //     'username'=>['required'],
    //     //     'email'=>['required','email',Rule::unique('us
ers','email')],
    //     //     'password'=>'required',
    //     //     'age'=>['required'],
    //     //     'nationality'=>['required']
    //     // ]);
```

```
    //     // $user=User::create([
    //     //     'first_name'=>$formFields['firstname'],
    //     //     'last_name'=>$formFields['lastname'],
    //     //     'username'=>$formFields['username'],
    //     //     'email'=>$formFields['email'],
    //     //     'password'=>Hash::make($formFields['password'
]),
    //     //     'age'=>$formFields['age'],
    //     //     'nationality'=>$formFields['nationality']
    //     // ]);

    //     // auth()->login($user);

    //     // return redirect('/')->with('success','Your
account has been created');

    //     $formFields = $request->validate([
    //         'firstname' => ['required'],
    //         'lastname' => ['required'],
    //         'username' => ['required'],
    //         'email' => ['required', 'email',
Rule::unique('users', 'email')],
    //         'password' => 'required',
    //         'age' => ['required'],
    //         'nationality' => ['required']
    //     ]);

    //     $user = User::create([
    //         'first_name' => $formFields['firstname'],
    //         'last_name' => $formFields['lastname'],
    //         'username' => $formFields['username'],
    //         'email' => $formFields['email'],
    //         'password' =>
Hash::make($formFields['password']),
    //         'age' => $formFields['age'],
```

```
    //         'nationality' => $formFields['nationality']
    //     ]);

    //     return response()->json(['message' => 'Your account
has been created'], 201);

    // }


    public function store(Request $request)
{
    $formFields = $request->validate([
        'firstname' => ['required'],
        'lastname' => ['required'],
        'username' => ['required'],
        'email' => ['required', 'email', Rule::unique('users',
'email')],
        'password' => 'required',
        'age' => ['required'],
        'nationality' => ['required'],
        'role_type' => ['required'] // Add role field
    ]);

    $user = User::create([
        'first_name' => $formFields['firstname'],
        'last_name' => $formFields['lastname'],
        'username' => $formFields['username'],
        'email' => $formFields['email'],
        'password' => Hash::make($formFields['password']),
        'age' => $formFields['age'],
        'nationality' => $formFields['nationality'],
        'role_type' => $formFields['role_type'] // Save the
role in the database
    ]);

    return response()->json([
```

```
        'message' => 'Your account has been created',
        'user'=>$user,
    ], 201);
}


    /**
     * Display the specified resource.
     *
     * @param  \App\Models\user  $user
     * @return \Illuminate\Http\Response
     */
    public function show(user $user)
    {
        //
    }

    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \App\Models\user  $user
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, user $user)
    {
        //
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  \App\Models\user  $user
     * @return \Illuminate\Http\Response
     */
    public function destroy(user $user)
```

```
    {
        //
    }

    public function login()
    {
        return view('login');
    }

// public function authenticate(Request $request){
//     // $formFields=$request->validate([
//     //     'email'=>['required','email'],
//     //     'password'=>['required']
//     // ]);

//     // if(auth()->attempt($formFields)){

//     //     $request->session()->regenerate();//this is to
regenerate the session id for the user

//     //     return redirect('/')->with('success','You have
been logged in');
//     // }

//     // return back()->withErrors([
//     //     'email'=>'The provided credentials do not match
our records'
//     // ])->onlyInput('email');

//     $formFields = $request->validate([
//         'email' => ['required', 'email'],
//         'password' => ['required'],
//     ]);

//     if (Auth::attempt($formFields)) {
```

```
//         return response()->json(['message' => 'You have been
logged in']);
//     }

//     return response()->json(['error' => 'The provided
credentials do not match our records'], 401);
// }

public function authenticate(Request $request)
{
    $formFields = $request->validate([
        'email' => ['required', 'email'],
        'password' => ['required'],
    ]);

    if (Auth::attempt($formFields)) {
        $user = Auth::user();
        if ($user->role === 'admin') {
            return response()->json(['message' => 'Logged in as
admin',  'user'=>$user,]);
        } else {
            return response()->json(['message' => 'Logged in as
user',   'user'=>$user,]);
        }
    }
    return response()->json(['error' => 'The provided
credentials do not match our records'], 401);
}


    public function logout(Request $request)
    {
        // auth()->logout();
```

```
        // $request->session()->invalidate();//this is to
destroy the session which is the cookie that is stored in the
browser used to authenticate the user
        // $request->session()->regenerateToken();//this is to
regenerate the token for the session
        // return redirect('/')->with('success','You have been
logged out');
        Auth::logout();

        $request->session()->invalidate();
        $request->session()->regenerateToken();

        return response()->json(['message' => 'You have been
logged out']);
    }



    public function announcement()
    {
        return $this -> HasMany(Announcement::class); //this is
the relationship between the user and the announcement and
$this is the user

    }
}
```

Code Explanation:

The provided code snippet is a PHP code segment from a UserController class. Here's a concise summary:

- The create method returns a view for the registration form.
- The store method validates the registration form data, creates a new user record in the users table, and returns a JSON response with a success message.
- The show, update, and destroy methods are empty and not implemented.
- The login method returns a view for the login form.
- The authenticate method validates the login form data and attempts to authenticate the user. If successful, it returns a JSON response indicating the user's role.
- The logout method logs out the authenticated user, invalidates the session, regenerates the session token, and returns a JSON response with a success message.
- The announcement method defines a relationship between the user and the announcement models (assuming there is a hasMany relationship).