Login code with CSRF token in Angular 14, Spring Boot and PostgreSQL

To implement a login system with CSRF token using Angular 14, Spring Boot, and PostgreSQL, you can follow the steps outlined below:

1. Create a new Angular project using the Angular CLI:

```arduino
ng new angular-login-app
```

1. Install the `@angular/material`, `@angular/flex-layout`, and `@angular/forms` packages:

```bash
npm install @angular/material @angular/flex-layout @angular/forms
```

1. Create a login component:

```
ng generate component login
```

1. In the login component, create a form with email and password fields:

```html
<form [formGroup]="loginForm" (ngSubmit)="onSubmit()">
  <mat-form-field>
    <input matInput placeholder="Email" formControlName="email">
  </mat-form-field>

  <mat-form-field>
    <input matInput placeholder="Password" type="password" formControlName="
  </mat-form-field>

  <button mat-raised-button color="primary" type="submit">Login</button>
</form>
```

1. In the login component class, create a `FormGroup` for the login form:

```typescript
import { Component, OnInit } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {
  loginForm: FormGroup;

  ngOnInit() {
    this.loginForm = new FormGroup({
      email: new FormControl('', [Validators.required, Validators.email]),
      password: new FormControl('', Validators.required)
    });
  }

  onSubmit() {
    // handle login form submission
  }
}
```

1. Install the `ngx-cookie-service` package to manage cookies:

```
npm install ngx-cookie-service
```

1. In the login component class, inject the `CookieService`:

```typescript
import { CookieService } from 'ngx-cookie-service';

constructor(private cookieService: CookieService) { }
```

1. When the login form is submitted, use the `HttpClient` to send a POST request to the Spring Boot backend:

```typescript
import { HttpClient } from '@angular/common/http';

constructor(private http: HttpClient) { }

onSubmit() {
  const credentials = {
    email: this.loginForm.get('email').value,
    password: this.loginForm.get('password').value
  };

  this.http.post('/api/login', credentials, { withCredentials: true }).subsc
    // handle successful login
  });
}
```

1. In the Spring Boot backend, create a `UserController` with a `/login` endpoint that generates a CSRF token:

```java
@RestController
public class UserController {
  @Autowired
  private CsrfTokenRepository csrfTokenRepository;

  @PostMapping("/login")
  public ResponseEntity<?> login(@RequestBody Credentials credentials, HttpS
    // authenticate user
    // generate CSRF token
    CsrfToken csrfToken = this.csrfTokenRepository.generateToken(request);
    this.csrfTokenRepository.saveToken(csrfToken, request, response);

    return ResponseEntity.ok().build();
  }
}
```

1. In the Angular app, use the `HttpClient` interceptor to add the CSRF token to all outgoing requests:

```typescript
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from '@angul
import { Observable } from 'rxjs';
import { CookieService } from 'ngx-cookie-service';

@Injectable()
export class CsrfInterceptor implements HttpInterceptor {
  constructor(private cookieService: CookieService) { }

  intercept(request: HttpRequest<any>, next: HttpHandler
```

How to connect Angular 14 with PostgreSQL database with springboot

To connect Angular 14 with a PostgreSQL database using Spring Boot, you can follow these general steps:

1. Create a new Spring Boot project with Spring Initializr and select the following dependencies:

- Spring Web
- Spring Data JPA
- PostgreSQL Driver

1. Create a database in PostgreSQL and configure the connection properties in the `application.properties` file of your Spring Boot project:

```arduino
spring.datasource.url=jdbc:postgresql://localhost:5432/mydatabase
spring.datasource.username=myuser
spring.datasource.password=mypassword
spring.jpa.hibernate.ddl-auto=update
```

2. Create a new entity class representing a table in your PostgreSQL database. For example, suppose we have a table called `customer` with `id`, `name`, and `email` columns. Then, the corresponding entity class would look like this:

```java
@Entity
@Table(name = "customer")
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "email")
    private String email;

    // constructors, getters, and setters
}
```

3. Create a repository interface that extends the `JpaRepository` interface, which provides basic CRUD operations for the entity. For example:

```java
public interface CustomerRepository extends JpaRepository<Customer, Long> {
}
```

4. Create a Spring REST controller that handles HTTP requests and responses for the `Customer` entity. For example:

```java
@RestController
@RequestMapping("/api/customers")
public class CustomerController {
    @Autowired
    private CustomerRepository customerRepository;

    @GetMapping("")
    public List<Customer> getAllCustomers() {
        return customerRepository.findAll();
    }

    @PostMapping("")
    public Customer createCustomer(@RequestBody Customer customer) {
        return customerRepository.save(customer);
    }

    @GetMapping("/{id}")
    public Customer getCustomerById(@PathVariable Long id) {
        return customerRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Customer n
    }

    @PutMapping("/{id}")
    public Customer updateCustomer(@PathVariable Long id, @RequestBody Custo
        Customer customer = customerRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Customer n
        customer.setName(customerDetails.getName());
        customer.setEmail(customerDetails.getEmail());
        return customerRepository.save(customer);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<?> deleteCustomer(@PathVariable Long id) {
        Customer customer = customerRepository.findById(id)
                .orElseThrow(() -> new ResourceNotFoundException("Customer n
        customerRepository.delete(customer);
        return ResponseEntity.ok().build();
    }
}
```

5. In the Angular app, create a service that interacts with the Spring REST API using the `HttpClient`. For example:

```typescript
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Customer } from './customer';

@Injectable({
  providedIn: 'root'
})
export class CustomerService {
  private baseUrl = '/api/customers';

  constructor(private http: HttpClient) { }

  getAllCustomers(): Observable<Customer[]> {
    return this.http.get<Customer[]>(this.baseUrl);
  }

  getCustomerById(id: number): Observable<Customer> {
    return this.http.get<Customer>(`${this.baseUrl}/${id}`);
  }

  createCustomer(customer: Customer): Observable<Customer> {
    return this.http.post<Customer>(this.baseUrl, customer);
  }

  updateCustomer(id: number, customer: Customer): Observable<Customer> {
    return this.http.put<Customer>(`${this.baseUrl}/${id}`, customer);
  }

  deleteCustomer(id
```

How to connect PostgreSQL database with Angular 14 in Visual basic Studio?

To connect a PostgreSQL database with an Angular 14 application in Visual Studio, you will need to follow these general steps:

1. Install the `pg` library for Node.js, which is the PostgreSQL client for JavaScript. You can install it using the following command in the terminal:

```
npm install pg
```

1. In your Angular project, create a new service that will handle the database connection and SQL queries. For example, let's create a service called `DatabaseService`:

```typescript
import { Injectable } from '@angular/core';
import { Pool, PoolClient } from 'pg';

@Injectable({
  providedIn: 'root'
})
export class DatabaseService {
  private pool: Pool;

  constructor() {
    this.pool = new Pool({
      user: 'myuser',
      host: 'localhost',
      database: 'mydatabase',
      password: 'mypassword',
      port: 5432
    });
  }

  async query(sql: string, params?: any[]): Promise<any> {
    const client: PoolClient = await this.pool.connect();
    try {
      const result = await client.query(sql, params);
      return result.rows;
    } finally {
      client.release();
    }
  }
}
```

This service creates a connection pool to the PostgreSQL database using the connection properties in the constructor. It also provides a `query` method that can be used to execute SQL queries on the database.

1. In your Angular components that need to interact with the database, inject the `DatabaseService` and call its `query` method to execute SQL queries. For example, suppose we want to retrieve all customers from a `customer` table in the database. We can create a component called `CustomerListComponent` and use the `DatabaseService` to retrieve the customers:

```typescript
import { Component, OnInit } from '@angular/core';
import { DatabaseService } from '../database.service';

@Component({
  selector: 'app-customer-list',
  templateUrl: './customer-list.component.html',
  styleUrls: ['./customer-list.component.css']
})
export class CustomerListComponent implements OnInit {
  customers: any[];

  constructor(private db: DatabaseService) { }

  async ngOnInit(): Promise<void> {
    this.customers = await this.db.query('SELECT * FROM customer');
  }
}
```

In this example, the `ngOnInit` method of the `CustomerListComponent` calls the `query` method of the `DatabaseService` to retrieve all customers from the `customer` table in the database.

Note that this example uses the `any` type for the returned rows. For better type safety, you should create interfaces for the tables in your database and use them instead of `any`.