Backend

Tools Used:-

Spring Boot
Spring Data MongoDB
Spring Security
MongoDB

Why Spring?

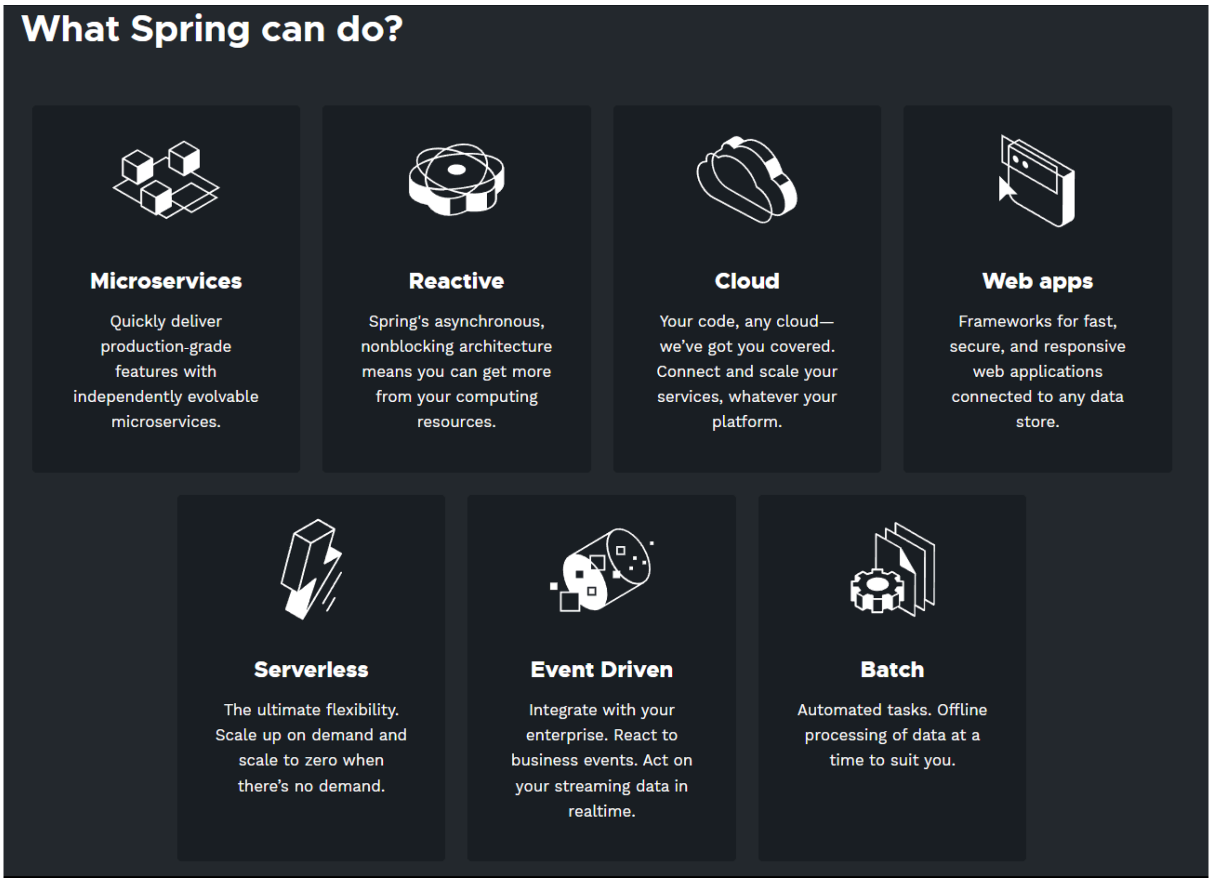
The choice of technology stack, whether it's Spring, Node.js, .NET, or any other framework or language, depends on various factors including:

1. Requirements of the project: Different projects may have different requirements in terms of scalability, performance, security, etc. Certain technologies may be better suited for specific requirements. For instance, Node.js is often favoured for real-time applications due to its event-driven, non-blocking I/O model.
2. Existing expertise: The skills and expertise of the development team can influence the choice of technology. If the team is already proficient in Java and Spring, it might be more efficient to use Spring for backend development.
3. Ecosystem and community support: The availability of libraries, tools, and community support can also impact the decision. A technology with a strong ecosystem and community support can provide better resources and solutions for development challenges.
4. Performance: Different technologies have different performance characteristics. The performance requirements of the project may influence the choice of technology.
5. Scalability: Scalability requirements may also play a role in the choice of technology. Some technologies are better suited for horizontally scaling applications across multiple servers or containers.
6. Cost considerations: Licensing costs, hosting costs, and other financial considerations may influence the choice of technology.

Advantages Of Spring

1. Dependency Injection (DI): Spring provides a powerful dependency injection mechanism that helps manage the dependencies between components in a flexible and loosely coupled manner. DI promotes better testability, maintainability, and scalability of applications.

2. Aspect-Oriented Programming (AOP): Spring supports AOP, which allows developers to modularize cross-cutting concerns such as logging, security, and transaction management. AOP helps in separating concerns and promoting code reusability.
3. Transaction management: Spring provides robust support for transaction management, including declarative transaction management through annotations or XML configuration. This simplifies the implementation of transactional behavior in applications.
4. Integration with other frameworks and technologies: Spring integrates seamlessly with other Java frameworks and technologies, such as Hibernate, JPA, JDBC, JMS, and more. This allows developers to leverage the capabilities of these technologies within Spring applications easily.
5. Security: Spring Security provides comprehensive authentication, authorization, and other security features for securing applications. It offers support for various authentication mechanisms, including HTTP Basic, Digest, OAuth, and JWT.



What Spring can do?

Microservices
Quickly deliver production-grade features with independently evolvable microservices.

Reactive
Spring's asynchronous, nonblocking architecture means you can get more from your computing resources.

Cloud
Your code, any cloud—we've got you covered. Connect and scale your services, whatever your platform.

Web apps
Frameworks for fast, secure, and responsive web applications connected to any data store.

Serverless
The ultimate flexibility. Scale up on demand and scale to zero when there's no demand.

Event Driven
Integrate with your enterprise. React to business events. Act on your streaming data in realtime.

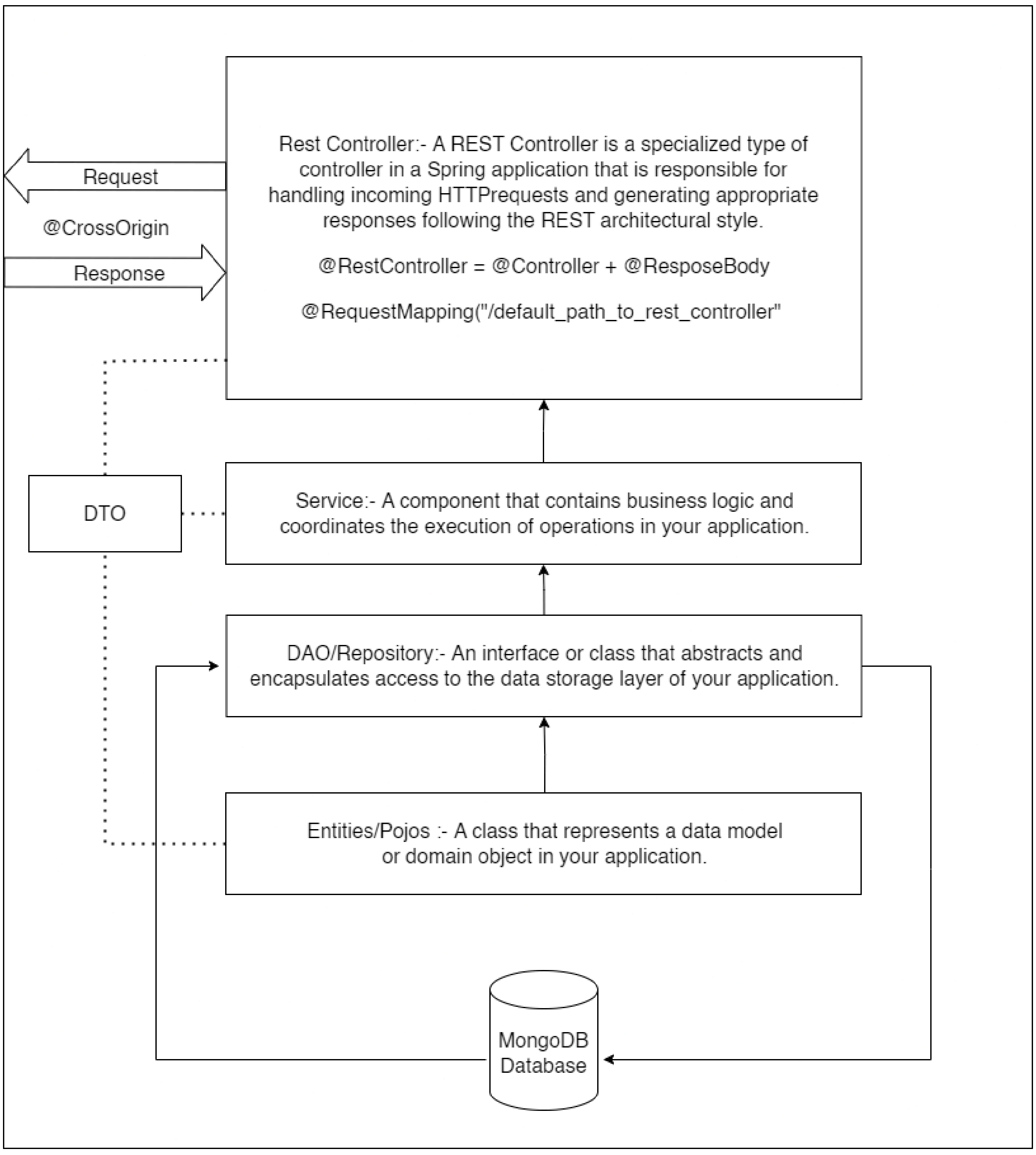
Batch
Automated tasks. Offline processing of data at a time to suit you.

Why Use MongoDB

Using MongoDB with Spring Boot offers several advantages, particularly in scenarios where a flexible and scalable NoSQL database is preferred. Here are some reasons why you might choose to use MongoDB with Spring Boot:

1. Schema flexibility: MongoDB is a document-oriented database, which means it stores data in flexible, JSON-like documents. This allows for dynamic and evolving schemas, making it suitable for scenarios where data structures may change frequently or vary between different documents.
2. Scalability: MongoDB is designed to scale horizontally by distributing data across multiple servers or clusters. This makes it well-suited for handling large volumes of data and supporting high-traffic applications.
3. Performance: MongoDB's flexible data model and support for embedded documents and arrays can lead to improved query performance, especially for use cases that involve complex data structures or nested relationships.
4. Developer productivity: Spring Boot provides seamless integration with MongoDB through the Spring Data MongoDB project. This integration abstracts away many of the complexities of working with MongoDB, allowing developers to focus on writing application code rather than low-level database interactions.
5. Rich querying capabilities: MongoDB supports powerful querying capabilities, including rich query expressions, aggregation pipelines, and secondary indexes. This enables developers to perform complex queries and analytics operations efficiently.
6. Document-oriented storage: MongoDB's document-oriented storage model aligns well with object-oriented programming paradigms, making it a natural fit for Java-based applications built with Spring Boot.
7. JSON support: Both MongoDB and Spring Boot use JSON as their native data interchange format, facilitating seamless integration and data exchange between the database and application layers.
8. Community and ecosystem: MongoDB has a large and active community of users, developers, and contributors. It offers extensive documentation, tutorials, and resources, as well as a rich ecosystem of third-party libraries and tools that integrate with Spring Boot.

REST API server as A Backend : Layered Architecture


Request
@CrossOrigin
Response
Rest Controller:- A REST Controller is a specialized type of controller in a Spring application that is responsible for handling incoming HTTPrequests and generating appropriate responses following the REST architectural style.
@RestController = @Controller + @ResposeBody
@RequestMapping("/default_path_to_rest_controller"
DTO
Service:- A component that contains business logic and coordinates the execution of operations in your application.
DAO/Repository:- An interface or class that abstracts and encapsulates access to the data storage layer of your application.
Entities/Pojos :- A class that represents a data model or domain object in your application.
MongoDB Database

Developing complex web applications requires careful organisation and structure to ensure maintainability, scalability, and efficient development. One of the most widely adopted approaches to achieve these goals is layered architecture. This concept involves dividing the application into distinct layers, each with specific responsibilities, that interact with each other in a well-defined way.

Benefits of Layered Architecture:

- Separation of Concerns: Each layer handles a specific set of tasks, making the code easier to understand, develop, and maintain. Changes in one layer are less likely to impact other parts of the application.
- Reusability: Components like services and data access layers can be reused across different parts of the application, reducing code duplication and promoting efficiency.
- Scalability: Layers can be independently scaled based on their specific needs. For example, the database layer might require more resources than the presentation layer during peak traffic.
- Testability: Each layer can be tested in isolation, simplifying the testing process and improving overall code quality.

Base Dependencies

```xml
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-devtools</artifactId>
   <scope>runtime</scope>
   <optional>true</optional>
</dependency>
<dependency>
   <groupId>org.projectlombok</groupId>
   <artifactId>lombok</artifactId>
   <optional>true</optional>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
</dependency>
```

spring-boot-starter-validation:

- This dependency provides support for bean validation in Spring Boot applications.
- It includes libraries such as Hibernate Validator, which allows you to use annotations like @NotNull, @Size, @Email, etc., for validating request payloads or model objects.
- Bean validation ensures that the data sent by clients meets specified criteria before processing it further in the application.

spring-boot-starter-web:

- This dependency is used to build web applications with Spring MVC (Model-View-Controller) framework in Spring Boot.
- It includes libraries for handling HTTP requests, managing controllers, processing request payloads, and generating responses.
- With this dependency, you can create RESTful APIs, web services, or traditional web applications using Spring MVC annotations like @RestController, @RequestMapping, etc.

spring-boot-devtools:

- This dependency provides development-time features to improve the developer experience.
- It includes tools such as automatic application restarts, live reload of resources, and enhanced error reporting for faster development iterations.
- This dependency is typically included as a development-only dependency to avoid affecting production deployments.

lombok:

- Lombok is a library that helps reduce boilerplate code in Java classes by generating common methods at compile time.
- With Lombok, you can annotate your Java classes with annotations like @Data, @Getter, @Setter, @NoArgsConstructor, etc., to automatically generate these methods.
- Lombok makes code more concise and readable by eliminating repetitive code patterns.

spring-boot-starter-test:

- This dependency includes libraries and utilities for testing Spring Boot applications.
- It provides support for writing unit tests, integration tests, and end-to-end tests for Spring Boot components.

- Libraries such as JUnit, Spring Test, Mockito, and AssertJ are included to facilitate writing and running tests efficiently.

Components Of Architecture

1. MongoDB Database

Databases are used to store data. MongoDB is a NoSql database which stores data in JSON format (BSON internally).

- The database is a structured collection of data organised for efficient storage, retrieval, and manipulation.
- In Spring applications, databases are typically used to persist application data, and various persistence technologies such as relational databases (e.g., MySQL, PostgreSQL) or NoSQL databases (e.g., MongoDB, Cassandra) can be used depending on the requirements of the application.
- Spring provides support for working with databases through various modules such as Spring Data JPA, Spring JDBC, Spring Data MongoDB, etc.
- We have Used Spring Data MongoDB

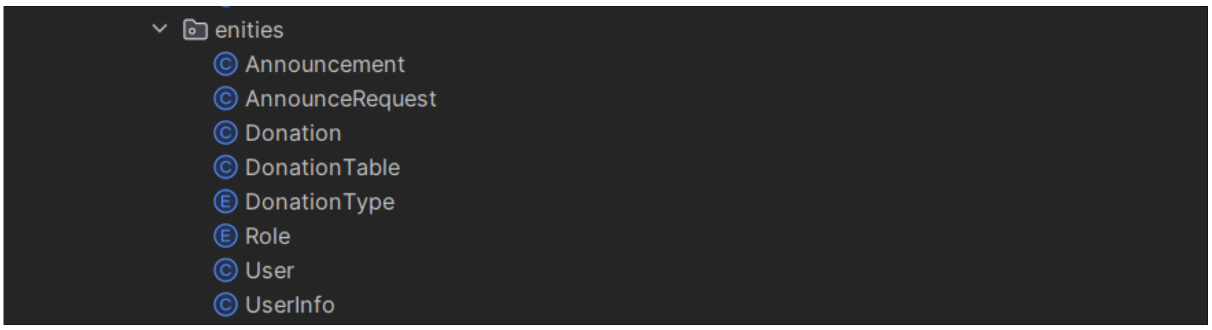
Maven dependency

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

2. Entities/POJOs

- In the context of MongoDB and Spring Data MongoDB, an entity or POJO represents a domain object or document in your MongoDB database.
- It is a Java class that typically maps directly to a MongoDB document, with each field representing a property in the document.
- Annotating the class with `@Document` in Spring Data MongoDB indicates that it's a MongoDB document and specifies the collection name if it differs from the class name.
- We have to use `@Id` for defining a ID

Our Entities


enities
Announcement
AnnounceRequest
Donation
DonationTable
DonationType
Role
User
UserInfo

We have a total of 6 Entities.

1. User:- User class defines a MongoDB document model for storing user information. It includes fields for the user's ID, first name, last name, email, password, city, role, and a reference to additional user information. Annotations such as @NotBlank, @Email, and @DBRef are used for validation and mapping purposes. Lombok annotations help reduce boilerplate code by automatically generating constructors, getter/setter methods, and a toString() method.
2. UserInfo:- It contains UserInfo like Aadhaar Card(List of Aadhaar for NGO), PAN Card(List of Aadhaar for NGO), Registration No(Only For NGOs). This is also the entity that is mapped with Our User entity via @DBRef.
3. Donation:- Donation class defines a MongoDB document model for storing information about donations. It includes fields for the donation's ID, donor name, donor email, type of donation, and quantity. Annotations such as @NotBlank are used for validation purposes to ensure that required fields are not empty or null.
4. DonationTable:- DonationTable class defines a MongoDB document model for storing information about donation tables. It includes fields for the donation table's ID, the user's ID, and a list of donation IDs associated with the user.
5. Announcement:- Announcement class defines a MongoDB document model for storing information about announcements. It includes fields for the announcement's ID, NGO name, requirement, and date. Here all the announcements approved by the Admin will stay.
6. AnnounceRequest:- It has the same exact structure as Announcement but here all the announcement requests made by NGO will stay. If the admin approves an announcement, the announcement request will be transferred from here to Announcement.

Enum:-

1. Role:-

- The Role enum defines a set of roles that users can have in the system.
- Enum Constants: The enum constants defined within the Role enum are:
    - ROLE_DONOR: Represents the role of a donor.
    - ROLE_ADMIN: Represents the role of an administrator.
    - ROLE_NGO: Represents the role of an NGO (Non-Governmental Organisation).

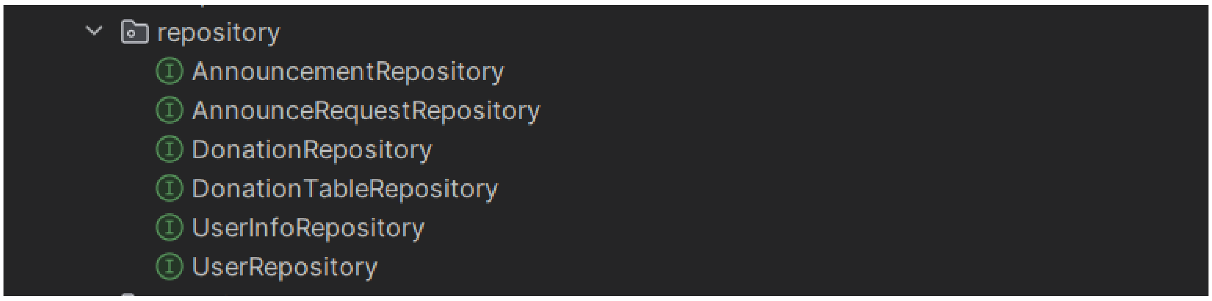
2. DonationType

- The DonationType enum defines the type of donations a Donor can make.
- Enum Constants: The enum constants defined within the Role enum are:
    - MONEY
    - STATIONARY
    - CLOTHES

3. DAO/Repository

- In Spring, a repository is a high-level abstraction over MongoDB data access operations, providing methods for common CRUD (Create, Read, Update, Delete) operations and custom query methods.
- Spring Data MongoDB repositories are interfaces that extend MongoRepository or ReactiveMongoRepository and are typically parameterized with the entity type and its primary key type.
- Repositories allow developers to interact with MongoDB collections using simple method signatures, without writing boilerplate code for data access operations.

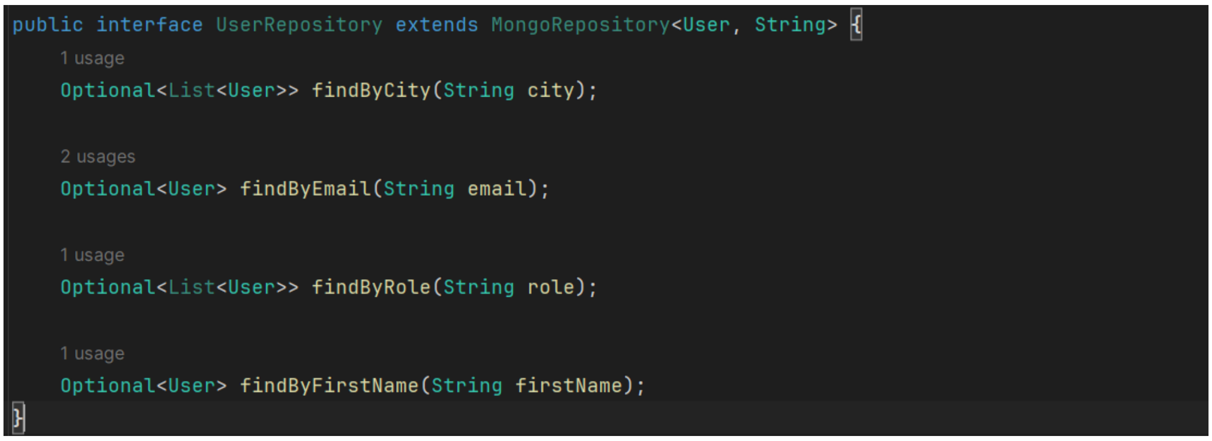
Our Repositories:-



```
repository
    AnnouncementRepository
    AnnounceRequestRepository
    DonationRepository
    DonationTableRepository
    UserInfoRepository
    UserRepository
```

Usually in DAO/Repository we don't need to write much code, we just need to autowire them to our service layer to use common methods provided by MongoRepository<> like find(), findAll(), findByID(), save(). But if we need some special function or use a custom query we can write them inside our DAO/Repository.

Here's an example of our UserRepository:-



```
public interface UserRepository extends MongoRepository<User, String> {
    1 usage
    Optional<List<User>> findByCity(String city);

    2 usages
    Optional<User> findByEmail(String email);

    1 usage
    Optional<List<User>> findByRole(String role);

    1 usage
    Optional<User> findByFirstName(String firstName);
}
```

As you can see, Some of the custom methods are defined above to fetch by a specific field.
As a design pattern. We could use Optional<> to fetch specific fields for better error handling.

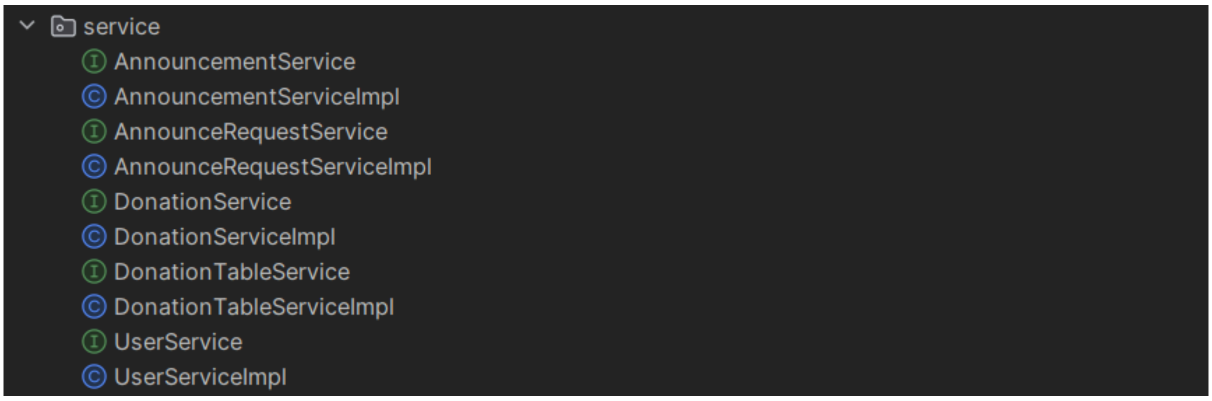
MongoRepository<>
MongoRepository is an interface provided by Spring Data MongoDB, which offers a high-level abstraction for interacting with MongoDB databases in Spring applications. It simplifies data access and reduces boilerplate code by providing ready-to-use CRUD (Create, Read, Update, Delete) operations and custom query methods.

4. Service

- Business Logic: The service layer encapsulates the business logic of the application. It performs operations such as data validation, transformation, aggregation, and orchestration of multiple repository methods.
- Transaction Management: Services manage transactions, ensuring that multiple repository operations are performed atomically. This helps maintain data consistency and integrity.
- Data Transformation: Services transform data between entity objects (representing database records) and DTOs (Data Transfer Objects) for communication with the controller layer.
- Error Handling: Services handle exceptions and errors gracefully, translating database-related exceptions into meaningful error messages or appropriate HTTP status codes for the controller layer.
- Security and Authorization: Services enforce security and authorization rules, ensuring that only authorised users can perform specific actions or access certain data.

Here we usually use Autowire Repository, PasswordEncoder, ModalMappers etc. @Autowired is used to inject dependencies (ModelMapper and PasswordEncoder in this case) into Spring-managed components, such as controllers or services. This enables the component to use these dependencies without manually creating instances or managing their lifecycle.

Our Services



```
service
    AnnouncementService
    AnnouncementServiceImpl
    AnnounceRequestService
    AnnounceRequestServiceImpl
    DonationService
    DonationServiceImpl
    DonationTableService
    DonationTableServiceImpl
    UserService
    UserServiceImpl
```

We use @Service and @Transactional above the Service Implementation. We separate Service and Service Implementation to provide loose coupling to the Spring IOC Container and so that we @Autowired it in the controller and rest things can be taken care of by the Controller.

@Service:

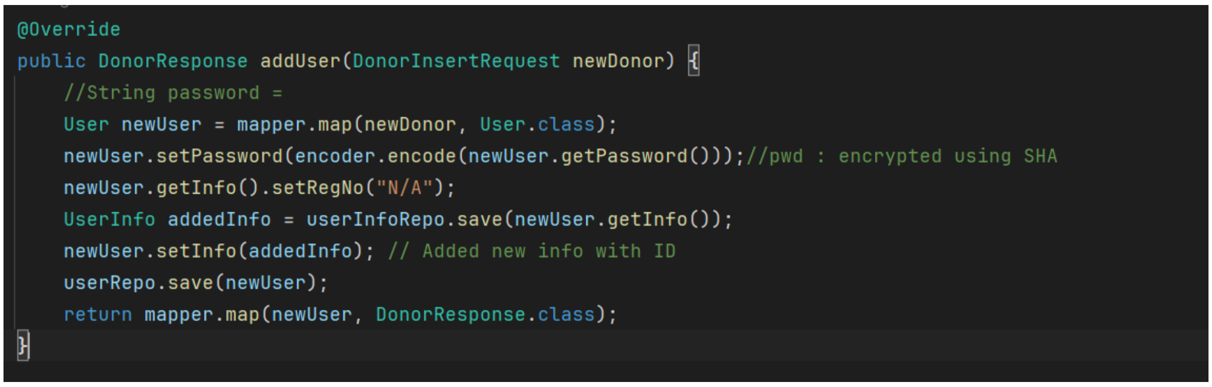
- @Service is a specialisation of the @Component annotation in Spring Framework.
- It is used to indicate that a particular class is a service component in the Spring application context.
- The @Service annotation serves as a marker for Spring to automatically detect and register the annotated class as a bean during component scanning.
- By using @Service, you can leverage Spring's dependency injection mechanism to inject service components into other Spring-managed beans, such as controllers or other services.

@Transactional:

- @Transactional is an annotation provided by Spring Framework that is used to mark a method, indicating that it should be executed within a database transaction.
- It starts a new database transaction before the method is executed and commits (or rolls back) the transaction after the method completes.

- @Transactional can be applied at both the class level (to apply to all methods in the class) or at the method level.
- It helps ensure data consistency and integrity by guaranteeing that multiple database operations within the annotated method are treated as a single atomic unit.
- @Transactional also provides options for specifying transaction propagation behaviour, isolation level, rollback rules, and more.

Example of a service to add a donor in database:-

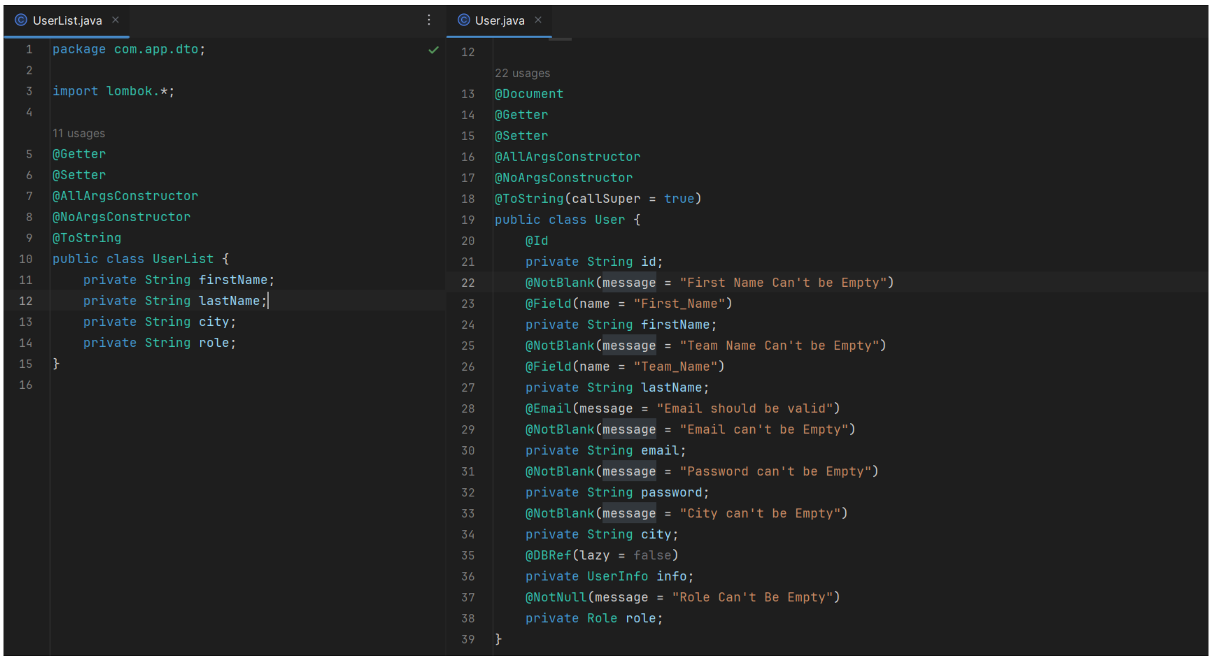


```java
@Override
public DonorResponse addUser(DonorInsertRequest newDonor) {
    //String password =
    User newUser = mapper.map(newDonor, User.class);
    newUser.setPassword(encoder.encode(newUser.getPassword()));//pwd : encrypted using SHA
    newUser.getInfo().setRegNo("N/A");
    UserInfo addedInfo = userInfoRepo.save(newUser.getInfo());
    newUser.setInfo(addedInfo); // Added new info with ID
    userRepo.save(newUser);
    return mapper.map(newUser, DonorResponse.class);
}
```

<u>DTO(Data Transfer Objects)</u>

- Definition: DTOs are plain Java objects used to transfer data between different layers of an application, such as between the service layer and the controller layer, or between microservices in a distributed system.
- Purpose: DTOs serve as lightweight data containers that encapsulate data required for specific use cases or interactions. They allow for decoupling between the domain model (entities) and the presentation layer (controllers), preventing overexposure of internal data structures.
- Mapping: In the service layer, entity objects retrieved from the database (via repositories) are mapped to DTOs before being passed to the controller layer. This mapping process involves copying relevant data from entities to DTOs, often using libraries like ModelMapper or manually implemented mapping methods.
- Data Validation: DTOs can also include validation annotations (e.g., @NotBlank, @Email, etc.) to enforce data integrity and consistency. These annotations can be validated in the service layer before processing the data further.
- Security: DTOs can be tailored to include only the necessary fields required by the client, thereby minimising the risk of exposing sensitive information. This helps enhance security by practising the principle of least privilege.

Example of User Entity and UserList which is sent to frontend with only part of the information. All other information is not sent to frontend.



```java
package com.app.dto;

import lombok.*;

11 usages
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class UserList {
    private String firstName;
    private String lastName;
    private String city;
    private String role;
}
```

```java

22 usages
@Document
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@ToString(callSuper = true)
public class User {
    @Id
    private String id;
    @NotBlank(message = "First Name Can't be Empty")
    @Field(name = "First_Name")
    private String firstName;
    @NotBlank(message = "Team Name Can't be Empty")
    @Field(name = "Team_Name")
    private String lastName;
    @Email(message = "Email should be valid")
    @NotBlank(message = "Email can't be Empty")
    private String email;
    @NotBlank(message = "Password can't be Empty")
    private String password;
    @NotBlank(message = "City can't be Empty")
    private String city;
    @DBRef(lazy = false)
    private UserInfo info;
    @NotNull(message = "Role Can't Be Empty")
    private Role role;
}
```

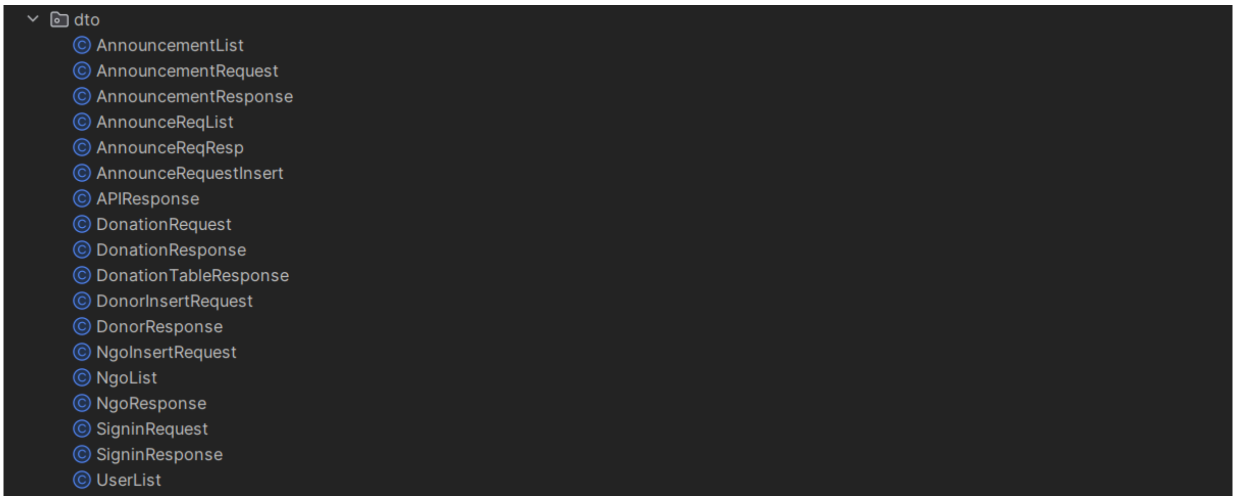
Here we Use ModalMapper.
ModelMapper is a library used for mapping one object to another, typically used for converting DTOs (Data Transfer Objects) to entity objects and vice versa.
To AutoWire ModalMapper in any class. We need first configure it as bean in our application.java

```java
@Bean
public ModelMapper mapper() {
ModelMapper modelMapper = new ModelMapper();
modelMapper.getConfiguration().setMatchingStrategy(MatchingStrategie
s.STRICT);
return modelMapper;
}
```

Using ModalMapper for mapping List<User> to List<UserList>

```java
@Autowired
ModelMapper mapper;
@Override
public List<UserList> getUserList() {
     List<User> userList = userRepo.findAll();
     List<UserList> userListResp = userList.stream()
                 .map(u -> mapper.map(u, UserList.class))
                 .collect(Collectors.toList());
     return userListResp;
}
```

Our DTOs:-



```
dto
  AnnouncementList
  AnnouncementRequest
  AnnouncementResponse
  AnnounceReqList
  AnnounceReqResp
  AnnounceRequestInsert
  APIResponse
  DonationRequest
  DonationResponse
  DonationTableResponse
  DonorInsertRequest
  DonorResponse
  NgoInsertRequest
  NgoList
  NgoResponse
  SigninRequest
  SigninResponse
  UserList
```

5. Controller

A controller in the context of web development, particularly in frameworks like Spring MVC, is a Java class responsible for handling incoming HTTP requests, processing them, and generating appropriate responses.

What Is RestController?

- @RestController is a specialised version of the @Controller annotation in Spring MVC framework.
- It's used to indicate that a particular class is a RESTful web service controller.
- It combines @Controller and @ResponseBody annotations, making it convenient to create RESTful APIs that return data directly in the response body.

Key Features of a RestController:

- Routing: Controllers use mappings (URL patterns) to define which methods should handle which types of requests. These mappings are typically defined using annotations such as @RequestMapping, @GetMapping, @PostMapping, etc.
- Request Processing: Controllers contain methods that are annotated to handle specific types of HTTP requests (GET, POST, PUT, DELETE, etc.). These methods process the incoming requests, perform necessary operations, and prepare responses.
- Data Binding: Controllers often use data binding mechanisms to extract data from request parameters, headers, or payloads and pass them to the business logic layer for further processing.

- View Resolution: In web applications that use server-side rendering, controllers are responsible for determining which view (HTML template) should be rendered as part of the response. They typically return the name of the view to be rendered.
- Response Generation: Controllers generate responses to be sent back to clients. These responses can include HTML content, JSON data, file downloads, redirections, or error messages, depending on the nature of the request and the application's requirements.

Various Annotation Used for that:-
The above features can be used by using following annotations:-
1. @RequestMapping:
    - @RequestMapping is a versatile annotation used to map HTTP requests to handler methods in Spring MVC controllers or Spring REST controllers
    - It can be used at both the class and method levels to specify the URI path, HTTP method, request parameters, headers, and other conditions for mapping.
    - @RequestMapping is flexible and supports a wide range of mappings, including simple path mappings, regular expressions, ant-style patterns, and more.
    - More specific annotations like @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, etc., are preferred for better readability and type safety.
2. @ResponseBody:
    - @ResponseBody is an annotation used to indicate that the return value of a method should be serialized directly into the HTTP response body.
    - It's typically used in combination with @Controller or @RestController annotations to return data directly in the response body without involving view resolution.
    - This annotation is commonly used when building RESTful APIs to return JSON, XML, or other data formats.
3. @RequestBody:
    - @RequestBody is an annotation used to bind the HTTP request body to a method parameter in Spring REST controller.
    - It can be used with POST, PUT, PATCH, and other HTTP methods that have a request body.
    - When a method parameter is annotated with @RequestBody, Spring automatically converts the request body (which may be JSON, XML, or other formats) into a Java object.

4. @PathVariable:

- @PathVariable is an annotation used to extract values from URI templates in Spring Controller.
- It binds the value of a URI template variable to a method parameter.
- It's commonly used in combination with @RequestMapping to capture path variables from the URI path.
- Path variables are placeholders in the URI path enclosed within curly braces ({}), and @PathVariable allows you to access these placeholders in your controller methods.

```
@RestController
@RequestMapping("/user")
public class UserController {

    @Autowired
    UserService userService;

    @PostMapping("/add/ngo")
    ResponseEntity<?> addNgo(@RequestBody @Valid NgoInsertRequest
newNgo) {
        //Check if role is equal to ROLE_ADMIN
        if (newNgo.getRole().name().equals("ROLE_ADMIN")) {
            return ResponseEntity.status(HttpStatus.BAD_REQUEST)
                    .body(new APIResponse("Can't Add Admin"));
        } else {
            NgoResponse resp = userService.addNgo(newNgo);
            return
ResponseEntity.status(HttpStatus.CREATED).body(resp);
        }
    }
}
```

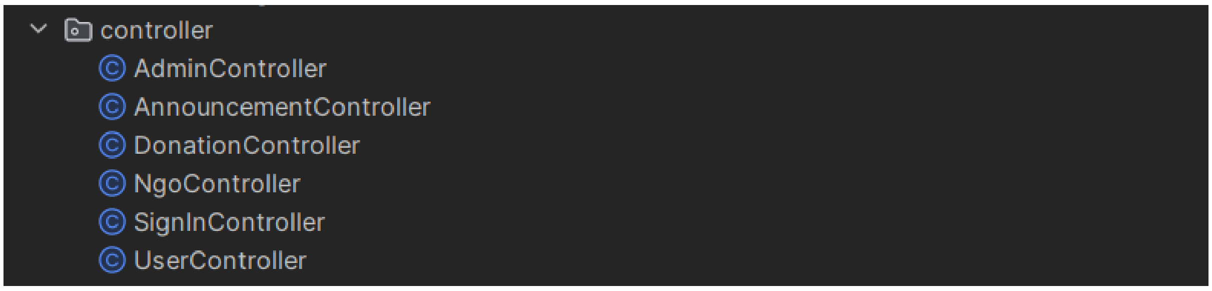
`ResponseEntity<?>:-`

- ResponseEntity<> is a generic class provided by Spring Framework that represents the entire HTTP response.
- It allows you to control the HTTP response status code, headers, and body returned by your controller methods.
- By using ResponseEntity<>, you can customize the response entity, including the status code, headers, and response body, based on your application's logic and requirements.

Some Basic Http status code that can be used with `ResponseEntity<?>`

- 200 OK: The request has succeeded. The response body contains the requested resource.
- 201 Created: The request has been fulfilled, and a new resource has been created.
- 204 No Content: The server successfully processed the request but does not need to return an entity body, and might want to return updated metadata.
- 400 Bad Request: The server could not understand the request due to invalid syntax, missing parameters, or other client-side errors.
- 401 Unauthorized: The request requires user authentication. The client must authenticate itself to get the requested response.
- 403 Forbidden: The server understood the request, but it refuses to authorize it. The client does not have access rights to the content.
- 404 Not Found: The server can't find the requested resource. It might be temporarily unavailable or may have been removed.
- 500 Internal Server Error: A generic error message indicating that something went wrong on the server-side. It's often caused by unexpected conditions that the server doesn't know how to handle.

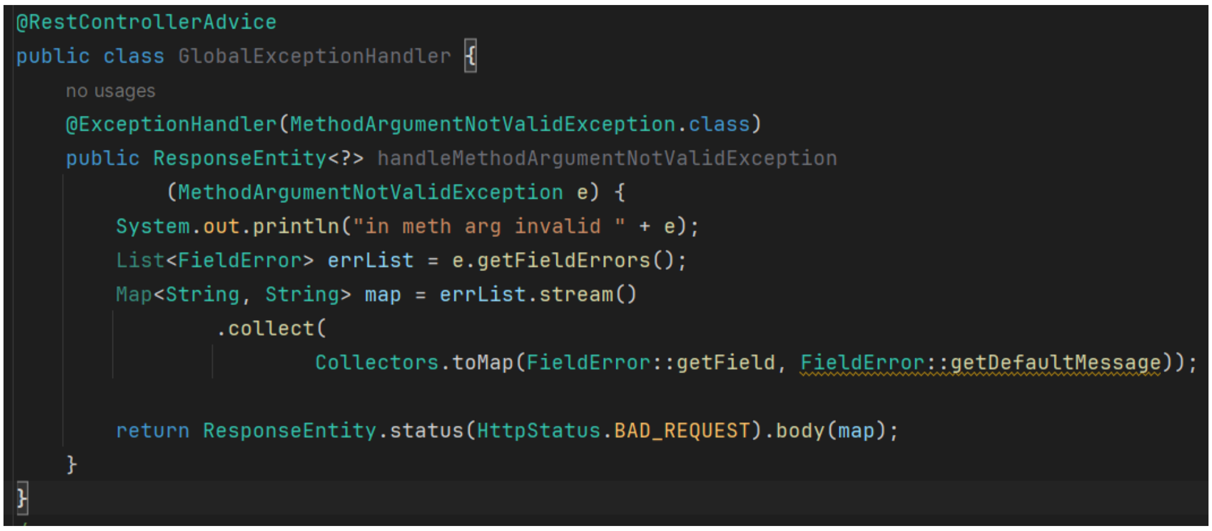
Our Controllers:-



```
controller
    AdminController
    AnnouncementController
    DonationController
    NgoController
    SignInController
    UserController
```

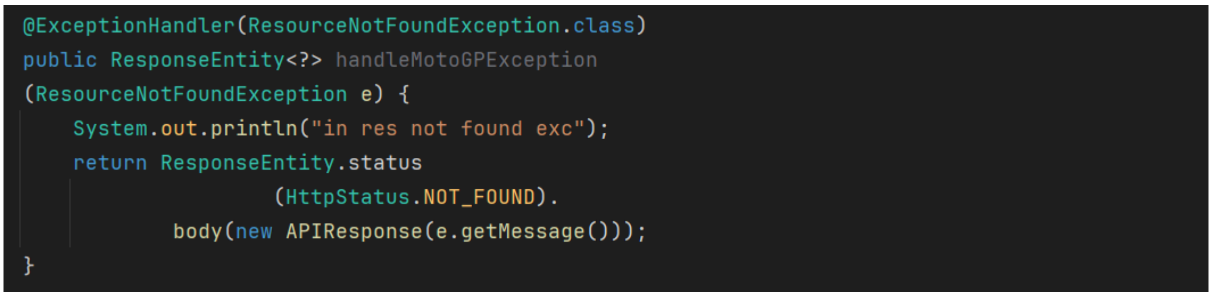
Global Exception Handling in Spring

Global exception handling in Spring Boot involves capturing and handling exceptions that occur throughout the application, providing a centralised mechanism to handle errors and return appropriate responses to clients.
For Creating A Global Exception Handler, We Can Use
@RestControllerAdvice:- It tells Spring Container that the following is a centralized custom exception handler ,it is used to provide COMMON ADVICE to all rest controllers regarding exception handling.
@ExceptionHandler:- It is used to define methods that handle specific types of exceptions that may occur during the execution of a controller's handler methods.

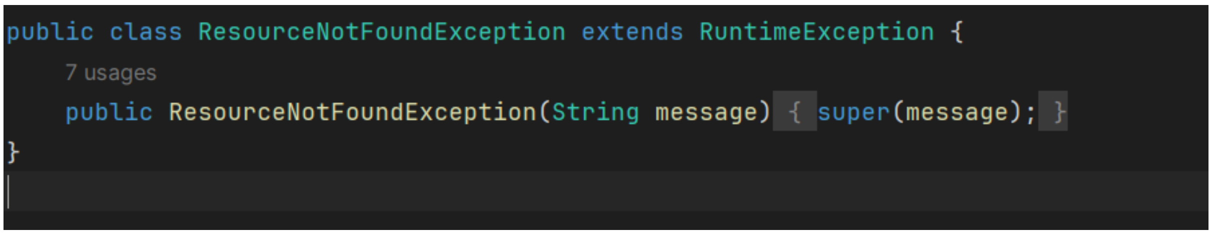


```
@RestControllerAdvice
public class GlobalExceptionHandler {
    no usages
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<?> handleMethodArgumentNotValidException
            (MethodArgumentNotValidException e) {
        System.out.println("in meth arg invalid " + e);
        List<FieldError> errList = e.getFieldErrors();
        Map<String, String> map = errList.stream()
                .collect(
                        Collectors.toMap(FieldError::getField, FieldError::getDefaultMessage));

        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(map);
    }
}
```

We can also add Custom Exception here:-



```
@ExceptionHandler(ResourceNotFoundException.class)
public ResponseEntity<?> handleMotoGPException
(ResourceNotFoundException e) {
    System.out.println("in res not found exc");
    return ResponseEntity.status
                (HttpStatus.NOT_FOUND).
            body(new APIResponse(e.getMessage()));
}
```



```
public class ResourceNotFoundException extends RuntimeException {
    7 usages
    public ResourceNotFoundException(String message) { super(message); }
}
```

CORS Configuration in Spring

CORS or Cross Origin Resource Sharing allows specific origins to access the application's resources, enables certain HTTP methods, and sets other CORS-related options. This configuration is essential for enabling cross-origin communication between different web applications or servers.

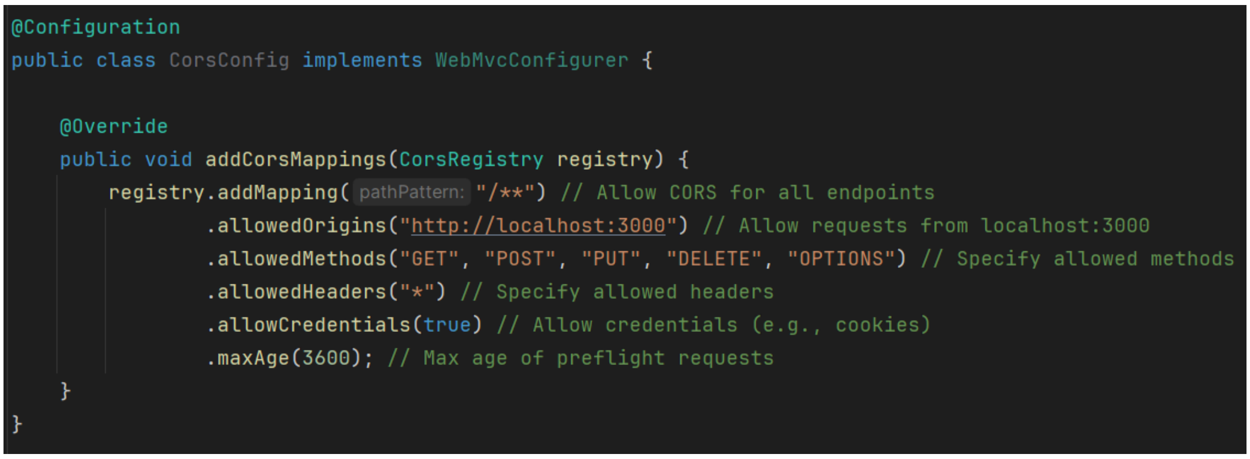
It can be done in three ways:-

1. Method Level
2. Class Level
3. Central CORS Configuration

We can implement Central CORS config by:-

1. Class Configuration: @Configuration: This annotation marks the class as a Spring configuration class, indicating that it contains bean definitions and configuration settings for the application.

2. Implementing WebMvcConfigurer: implements WebMvcConfigurer: This interface allows you to customize the behavior of Spring MVC, including CORS configuration.
3. Overriding addCorsMappings: Overrider public void addCorsMappings(CorsRegistry registry): This method is used to configure CORS settings within Spring MVC.



```java
@Configuration
public class CorsConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping( pathPattern: "/**") // Allow CORS for all endpoints
                .allowedOrigins("http://localhost:3000") // Allow requests from localhost:3000
                .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS") // Specify allowed methods
                .allowedHeaders("*") // Specify allowed headers
                .allowCredentials(true) // Allow credentials (e.g., cookies)
                .maxAge(3600); // Max age of preflight requests
    }
}
```

Spring Security

Spring Security is a powerful and widely used framework for implementing authentication, authorization, and other security features in Spring applications. It provides a comprehensive set of features and functionalities to secure your web applications and APIs.
Key Features of Spring Security:

- Authentication: Spring Security provides various mechanisms for user authentication, including form-based login, basic authentication, token-based authentication (e.g., JWT).
- Authorization: Once users are authenticated, Spring Security allows you to define authorization rules to control access to specific resources and functionalities based on user roles, permissions, or other criteria.
- Session Management: Spring Security manages user sessions, including session creation, invalidation, and security against attacks like session fixation.
- CSRF Protection: Spring Security helps prevent Cross-Site Request Forgery (CSRF) attacks by adding security tokens to forms and validating them during requests.
- Method Security: Spring Security allows you to secure individual methods within your controllers using annotations, enabling fine-grained access control on specific functionalities.

Dependencies for Spring Security and JWT

```xml
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.security</groupId>
   <artifactId>spring-security-test</artifactId>
   <scope>test</scope>
</dependency>
<dependency>
   <groupId>io.jsonwebtoken</groupId>
   <artifactId>jjwt-api</artifactId>
   <version>0.11.5</version>
</dependency>
<dependency>
   <groupId>io.jsonwebtoken</groupId>
   <artifactId>jjwt-impl</artifactId>
   <version>0.11.5</version>
   <scope>runtime</scope>
</dependency>
<dependency>
   <groupId>io.jsonwebtoken</groupId>
   <artifactId>jjwt-jackson</artifactId>
   <version>0.11.5</version>
   <scope>runtime</scope>
</dependency>
```

<u>SecurityConfig File:-</u>

We can add SecurityConfig File that configures Spring Security to:

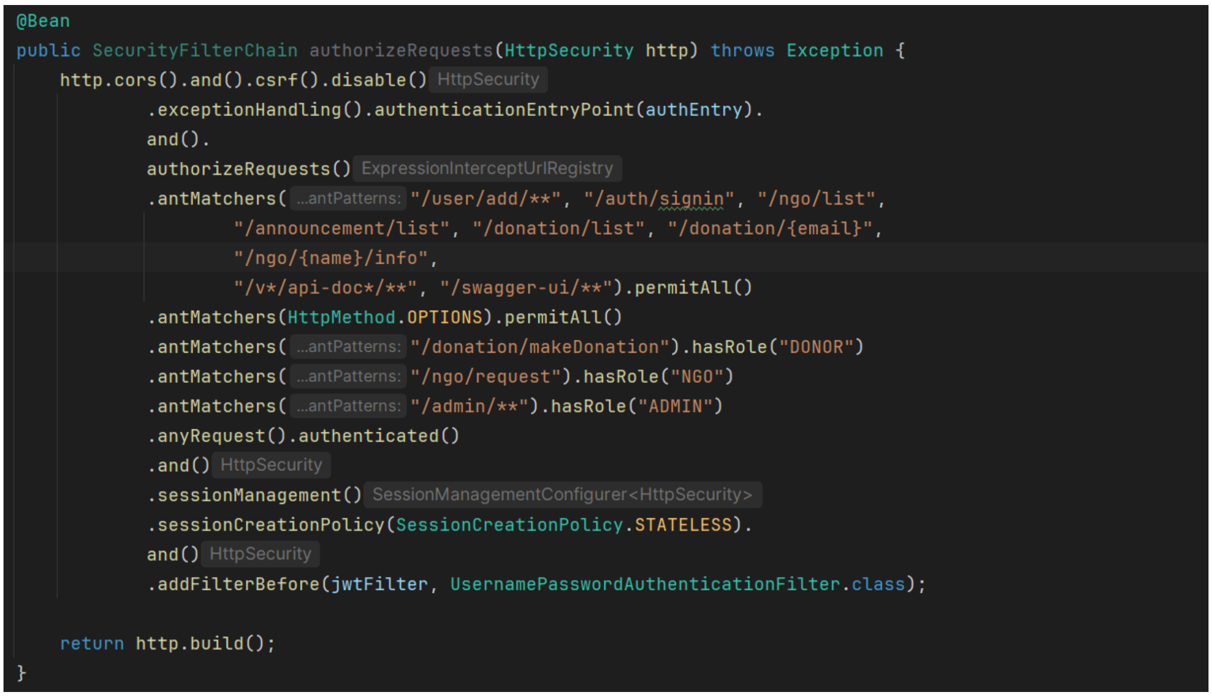
- Enable web security, method security, and CORS.
- Customise authentication and authorization rules.
- Use a custom authentication entry point.
- Disable CSRF protection (security implications should be considered).
- Employ stateless authentication with JWT.
- Define an AuthenticationManager bean.

We can use:

1. @EnableWebSecurity: Enables Spring Security's web security features.
2. @Configuration: Marks the class as a Spring configuration class.

Now we can configure SecurityFilterChain.

SecurityFilterChain:
It is a central component responsible for coordinating the various security filters involved in the authentication and authorization process. It represents a chain of filters that are applied sequentially to incoming requests, ultimately determining whether a user is authorised to access a specific resource.

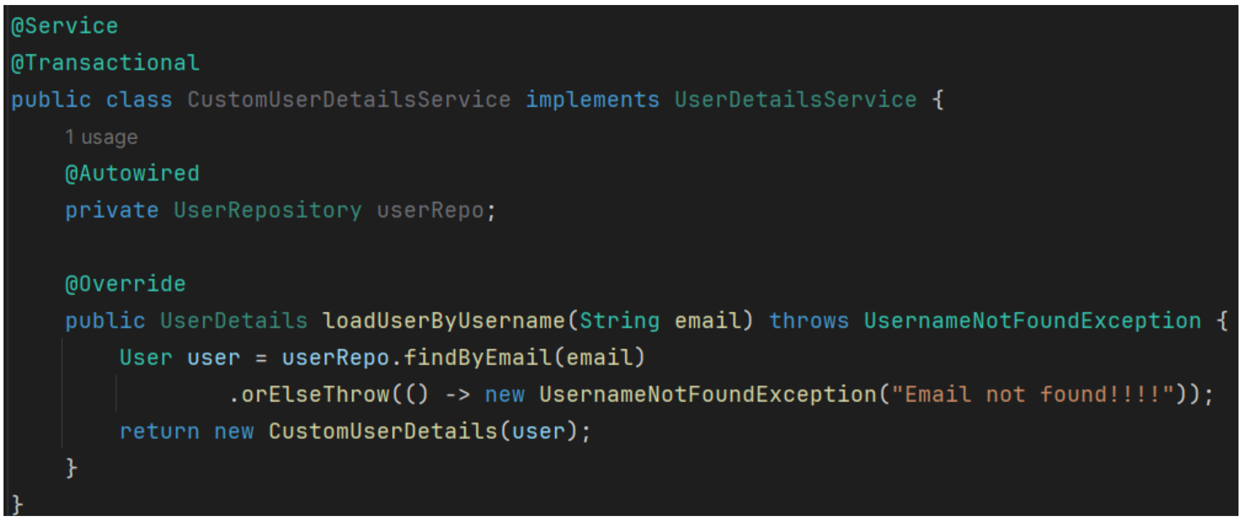


```
@Bean
public SecurityFilterChain authorizeRequests(HttpSecurity http) throws Exception {
    http.cors().and().csrf().disable() HttpSecurity
            .exceptionHandling().authenticationEntryPoint(authEntry).
            and().
            authorizeRequests() ExpressionInterceptUrlRegistry
            .antMatchers( ...antPatterns: "/user/add/**", "/auth/signin", "/ngo/list",
                    "/announcement/list", "/donation/list", "/donation/{email}",
                    "/ngo/{name}/info",
                    "/v*/api-doc*/**", "/swagger-ui/**").permitAll()
            .antMatchers(HttpMethod.OPTIONS).permitAll()
            .antMatchers( ...antPatterns: "/donation/makeDonation").hasRole("DONOR")
            .antMatchers( ...antPatterns: "/ngo/request").hasRole("NGO")
            .antMatchers( ...antPatterns: "/admin/**").hasRole("ADMIN")
            .anyRequest().authenticated()
            .and() HttpSecurity
            .sessionManagement() SessionManagementConfigurer<HttpSecurity>
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS).
            and() HttpSecurity
            .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

1. http.cors().and().csrf().disable(): Enables CORS and disables CSRF (Cross-Site Request Forgery) protection (consider the implications for security).
2. .exceptionHandling().authenticationEntryPoint(authEntry): Sets the custom authentication entry point for handling unauthorised requests.
3. .authorizeRequests(): Starts the configuring authorization rules.
   - .antMatchers(...).permitAll(): Allows specific endpoints to be accessed without authentication.
   - .antMatchers(HttpMethod.OPTIONS).permitAll(): Allows OPTIONS requests for CORS preflight checks.
   - .antMatchers(...).hasRole(...): Requires specific roles for certain endpoints.
   - .anyRequest().authenticated(): Requires authentication for all other requests.
4. .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS): Disables session creation for stateless authentication (often used with JWT).
5. .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class): Adds the JWT filter before the default authentication filter to handle JWT-based authentication.

DAO Layer Auth using Email And Password

After securing the endpoint we can do authentication in DAO Layer, Using username and password. For that we can do a custom implementation of UserDetailsService for user authentication in Spring Security using email and password.



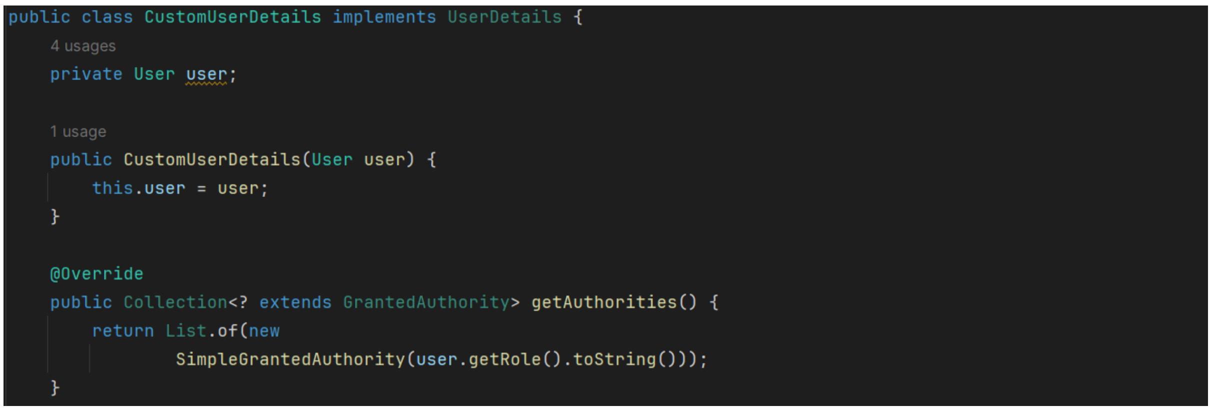
```java
@Service
@Transactional
public class CustomUserDetailsService implements UserDetailsService {
    1 usage
    @Autowired
    private UserRepository userRepo;

    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        User user = userRepo.findByEmail(email)
                .orElseThrow(() -> new UsernameNotFoundException("Email not found!!!!"));
        return new CustomUserDetails(user);
    }
}
```

When we implement UserDetailsService here, we need to override a method named loadUserByUsername(String).

This method is the core implementation that Spring Security calls during the authentication process.

- It takes the user's email address as input.
- It retrieves the user object from the UserRepository using the findByEmail(email) method.
- If the user is found, it creates a new CustomUserDetails object and returns it.

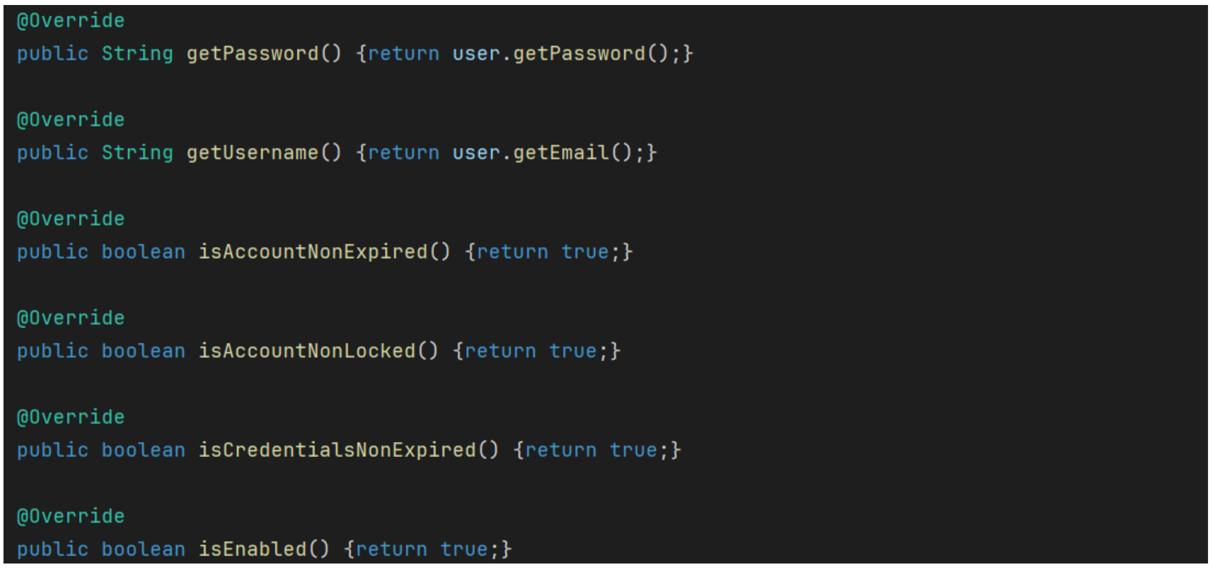
Now as you can see we are returning a CustomUserDetail object. So we need to create a class which implements UserDetails. Its purpose is to encapsulate user information retrieved from the database and provide it in a format that Spring Security can understand and use for authentication and authorization.



```java
public class CustomUserDetails implements UserDetails {
    4 usages
    private User user;

    1 usage
    public CustomUserDetails(User user) {
        this.user = user;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return List.of(new
                SimpleGrantedAuthority(user.getRole().toString()));
    }
```

After Implementing class we have to override many methods:-

1. getAuthorities():
   - Returns a collection of GrantedAuthority objects representing the user's roles or permissions.
   - In this implementation, it creates a list containing a single SimpleGrantedAuthority based on the user's role property.
2. getPassword(): Returns the user's password (retrieved from the User object).
3. getUsername(): Returns the user's unique identifier (in this case, the email address).
4. isAccountNonExpired(): Indicates whether the user's account is active and not expired. This implementation always returns true (no account expiration).
5. isAccountNonLocked(): Indicates whether the user's account is not locked due to too many failed login attempts. This implementation always returns true (no account locking).
6. isCredentialsNonExpired(): Indicates whether the user's credentials (password) are not expired and need to be reset. This implementation always returns true (no password expiration).
7. isEnabled(): Indicates whether the user's account is enabled and allowed to log in. This implementation always returns true (all accounts are enabled).



```java
@Override
public String getPassword() {return user.getPassword();}

@Override
public String getUsername() {return user.getEmail();}

@Override
public boolean isAccountNonExpired() {return true;}

@Override
public boolean isAccountNonLocked() {return true;}

@Override
public boolean isCredentialsNonExpired() {return true;}

@Override
public boolean isEnabled() {return true;}
```

JWT

JSON Web Token (JWT) is an open standard that defines a compact and self-contained way for securely transmitting information between two parties, typically a server and a client. It consists of three parts:

- Header: Contains information about the token itself, such as the signing algorithm used.
- Payload: Contains the actual claims, which are pieces of information about the user or the application context. These claims can be customised based on your needs.
- Signature: Ensures the integrity and authenticity of the token. It is created by signing the header and payload using a cryptographic algorithm and a secret key.

Benefits of using JWT:

- Stateless Authentication: JWTs eliminate the need for server-side session management, as they contain all the necessary information about the user within the token itself. This simplifies server architecture and improves scalability.
- Security: JWTs are digitally signed, making them tamper-proof. Any modification to the header or payload will invalidate the signature, ensuring data integrity.
- Compactness: JWTs are relatively small in size, making them suitable for transmission over HTTP headers or URL parameters.
- Flexibility: JWTs can be customised to carry various types of information and can be used for different purposes beyond authentication, such as data exchange and authorization.'

JWT in Spring

Inside application.properties add following properties:-

```
SECRET_KEY=1uORMccTFcTWQvjupAGxGapKFtcSRb4IPAaUJpfxbQoXuHutABXbNdt74
yAg4SvofyvqSY6MXcbwoTXp7gk2Q3jIjOjsaUAUz4xA+7OHU4w=
#JWT expiration timeout in msec : 24*3600*1000
EXP_TIMEOUT=86400000
```

Now inside JWT utils fetch both these things using @Value

1. Generating JWT Token: generateJwtToken(Authentication authentication):

- Takes an Authentication object as input, containing user information after successful authentication.
- Extracts the CustomUserDetails object from the Authentication object.
- Builds a JWT using the Jwts library:
    - Sets the subject (user email) and issues a timestamp.
    - Sets the expiration time based on the configured jwtExpirationMs.
    - Adds a custom claim named "authorities" containing a comma-separated String of user authorities.

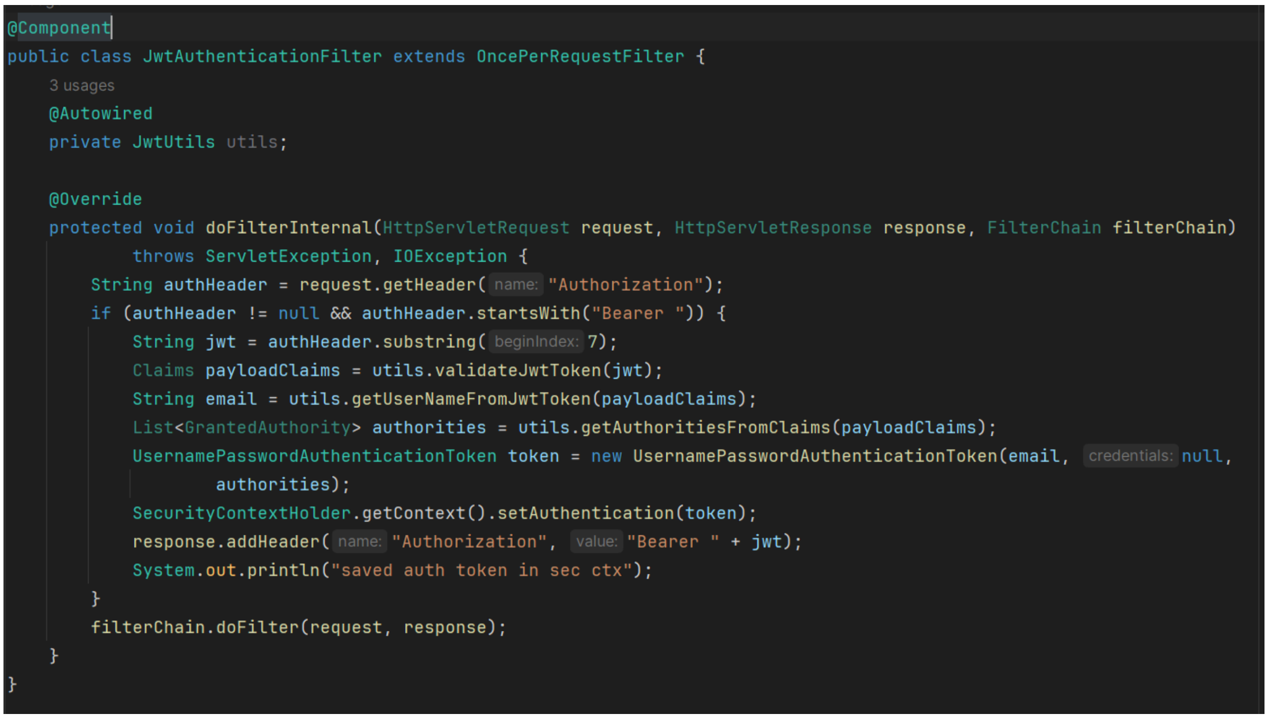
    - Signs the token using the secret key and the HS512 algorithm.
- Returns the compact JWT string.

2. Extracting Information from JWT:
   - getUserNameFromJwtToken(Claims claims): Extracts the username from the subject claim of the parsed JWT.
   - getAuthoritiesFromClaims(Claims claims): Extracts the user authorities from the custom "authorities" claim and converts them into a list of GrantedAuthority objects.
3. Validation:
   - validateJwtToken(String jwtToken): Parses the JWT string, verifies its signature using the secret key, and returns the claims if valid.

Adding JWT Auth Filter

- The filter intercepts requests, extracts and validates JWTs if present, and establishes authenticated sessions for valid tokens.
- It integrates JWT authentication into Spring Security's filter-based architecture.
- It simplifies authorization checks for subsequent resources within the application.

1. Filter Class:
   - JwtAuthenticationFilter: This class extends OncePerRequestFilter, ensuring it executes only once per request. It's responsible for handling JWT authentication within Spring Security's filter chain.

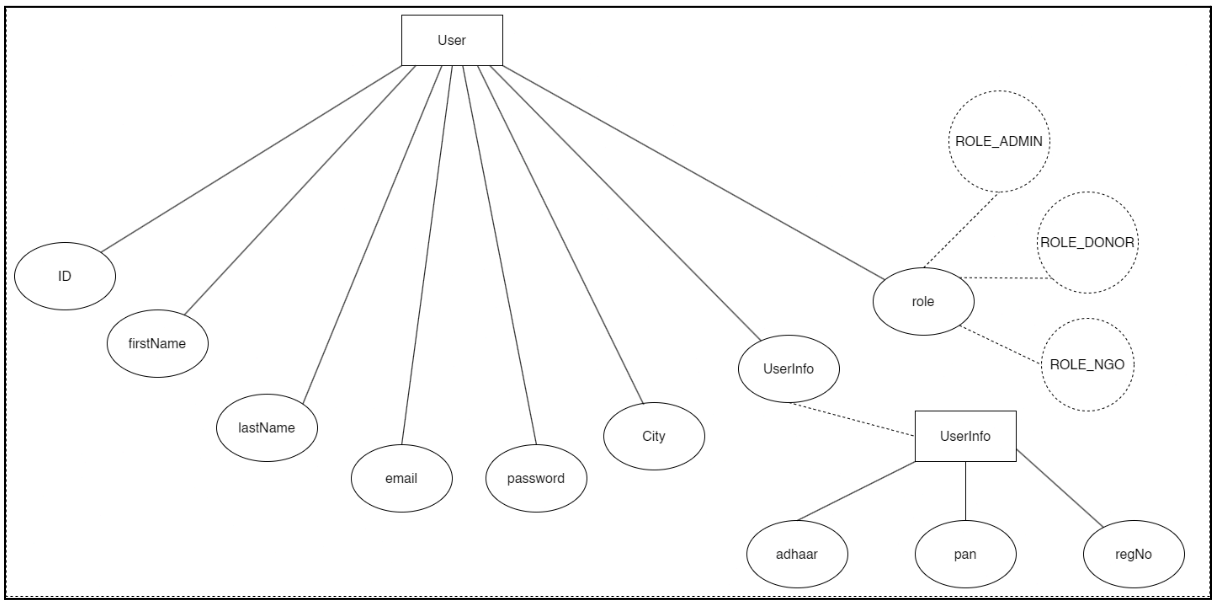


```
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {
    3 usages
    @Autowired
    private JwtUtils utils;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {
        String authHeader = request.getHeader( name: "Authorization");
        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            String jwt = authHeader.substring( beginIndex: 7);
            Claims payloadClaims = utils.validateJwtToken(jwt);
            String email = utils.getUserNameFromJwtToken(payloadClaims);
            List<GrantedAuthority> authorities = utils.getAuthoritiesFromClaims(payloadClaims);
            UsernamePasswordAuthenticationToken token = new UsernamePasswordAuthenticationToken(email,  credentials: null,
                    authorities);
            SecurityContextHolder.getContext().setAuthentication(token);
            response.addHeader( name: "Authorization",  value: "Bearer " + jwt);
            System.out.println("saved auth token in sec ctx");
        }
        filterChain.doFilter(request, response);
    }
}
```

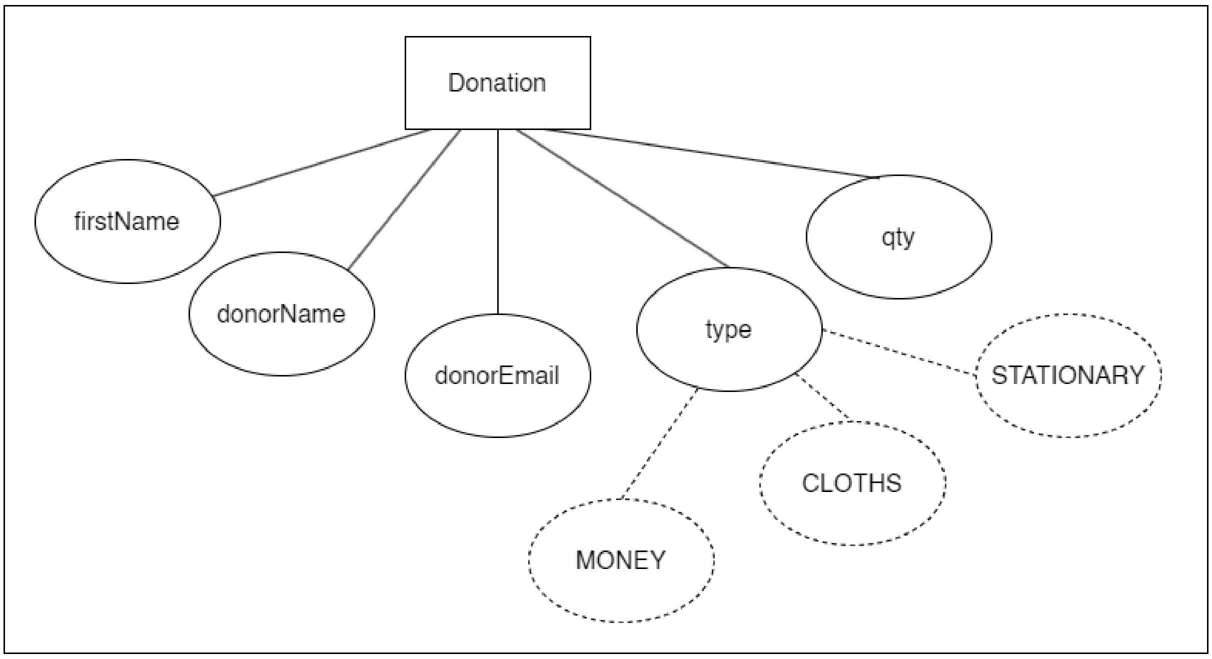
doFilterInternal: This method is invoked for each incoming request. It performs the following actions:

1. Retrieves Authorization Header: Checks for an "Authorization" header in the request, expecting a JWT in the format "Bearer <token>".
2. Extracts JWT (if present): If the header exists and starts with "Bearer ", it extracts the JWT token from the header value.
3. Validates JWT: Calls utils.validateJwtToken to verify the token's signature and integrity.
4. Extracts User Information: If the token is valid, it extracts:
    - Username from the subject claim (getUserNameFromJwtToken)
    - Authorities from the "authorities" claim (getAuthoritiesFromClaims)
5. Creates Authentication Token: Builds a UsernamePasswordAuthenticationToken using the extracted username and authorities, representing an authenticated user within Spring Security.
6. Sets Authentication in Context: Places the authentication token in the SecurityContextHolder, making it available for subsequent authorization checks within the application.
7. Adds JWT to Response Header (optional): Optionally adds the JWT back to the response header for convenience in subsequent requests.
8. Proceeds with Filter Chain: Calls filterChain.doFilter to continue processing the request with the remaining filters in the Spring Security chain.

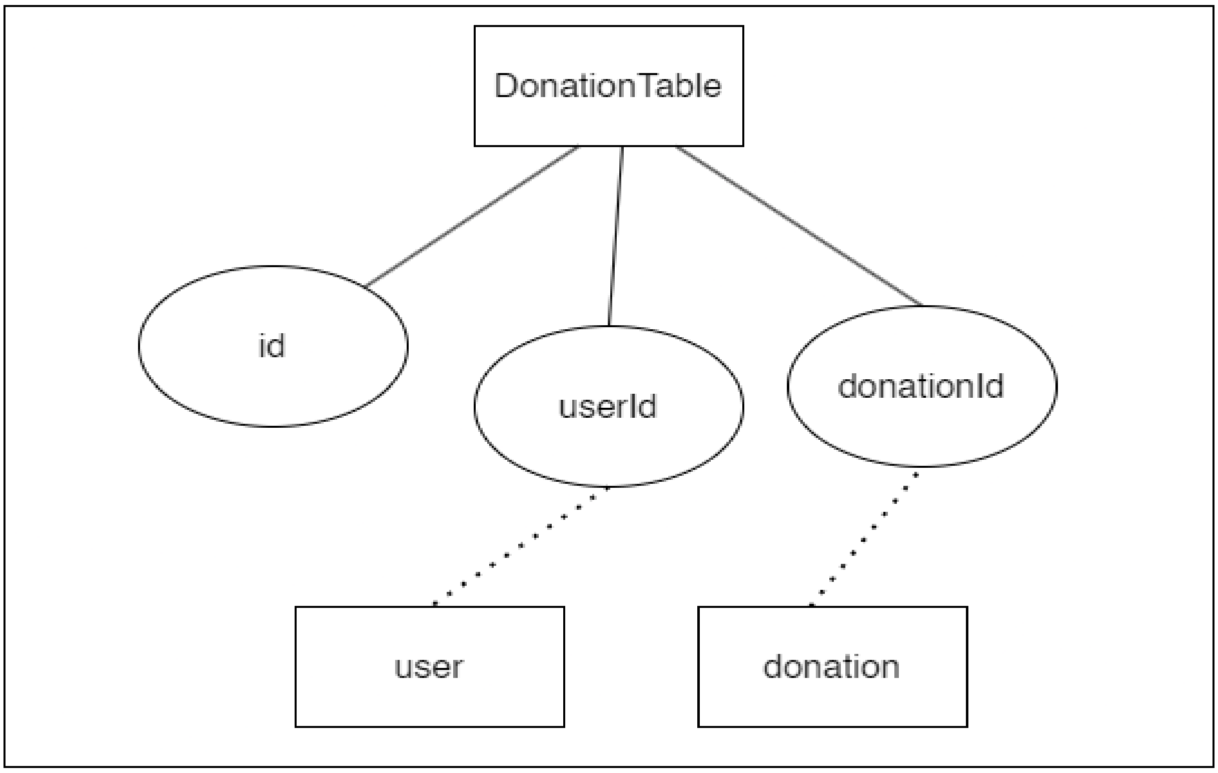
Collection Diagram



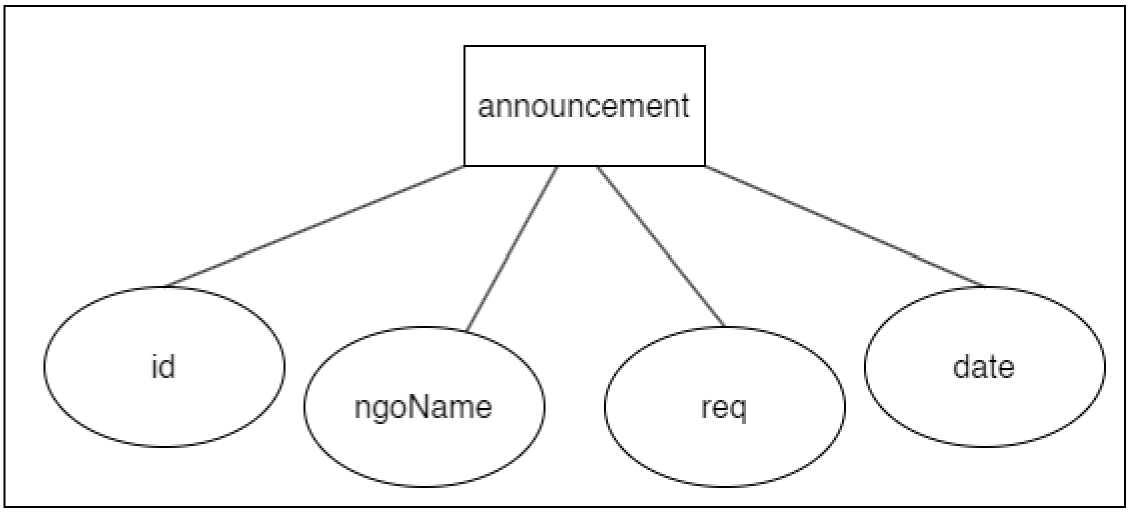
User and UserInfo



Donation Collection



DonationTable Collection



announcement (also for announceRequest) collection