1. program

searchForm.jsp – Home page JSP name

```
    <form:form modelAttribute="category">
        <p>
            <label>Select a category: </label>
            <form:select path="id"
items="${categories}" itemLabel="name"
itemValue="id"/>

        </p>
        <p id="buttons">
            <input id="submit" type="submit"
tabindex="5" value="Search">
        </p>    </form:form></div>
```

result.jsp – Result page JSP name

```
<c:forEach items="${books}" var="book">
    <tr>
        <td>${book.id}</td>
        <td>${book.title}</td>
        <td>${book.publishDate}</td>
    </tr>
</c:forEach>
       </tbody>
    </table>
</div>
```

Write Controller, Service, Repository classes below to implement this functionality.

1. You need two Controllers: classic Controller, REST Controller
2. Service class: You can omit it if there's no need.
3. For the server side(REST controller), REST URLs – you're free to create anything you want.
4. Any method you used to retrieve info from DB, you must declare in Repository or built in.

```
  @Controller
  public class BookController {

     @Autowired
     BookService bookService;

     @GetMapping("/")
     public String searchPage(@ModelAttribute
  Category category, Model model) {
         List<Category> categoryList =
  bookService.getAllCategories();
         categoryList.add(new Category(-1, "Please
  select a category"));
         Collections.reverse(categoryList);
         model.addAttribute("categories",
  categoryList);
         return "searchForm";
     }

     @PostMapping("/")
     public String getBooksByCategory(Category
  category, RedirectAttributes redirectAttributes) {

  redirectAttributes.addFlashAttribute("categoryId",
  category.getId());
         return "redirect:/result";
     }

     @GetMapping("/result")
     public String displayResult(Model model) {
         Integer categoryId = (Integer)
 model.asMap().get("categoryId");
         model.addAttribute("books",
 bookService.getAllBooksByCategoryId(categoryId
 ));
         return "result";
     }
 }

@Service
public class BookServiceImpl implements
BookService {

   @Autowired
   private RestTemplate restTemplate;

   private final String serviceUrl =
"http://localhost:8080/";

   @Override
   public Book get(Long id) {
      return null;
   }

   @Override
   public List<Category> getAllCategories() {
      ResponseEntity<List<Category>> response =
            restTemplate.exchange(serviceUrl +
"categories", HttpMethod.GET, null,
            new
ParameterizedTypeReference<List<Category>>() {
            });
      return response.getBody();
   }

   @Override
   public List<Book>
getAllBooksByCategoryId(Integer id) {
      ResponseEntity<List<Book>> response =
            restTemplate.exchange(serviceUrl +
"categories/"+id, HttpMethod.GET, null,
            new
ParameterizedTypeReference<List<Book>>() {
            });
      return response.getBody();
   }
}

@RestController
public class BookRestController {

   @Autowired
   BookService bookService;

   @GetMapping("/categories")
   public List<Category> getCategories(){
      return bookService.getAllCategories();
   }

   @GetMapping("/categories/{id}")
   public List<Book>
getBooksByCategoryId(@PathVariable Integer id){
      System.out.println(id);
      return
bookService.getAllBooksByCategoryId(id);
   }

}

@Repository
public interface BookRepository extends
JpaRepository<Book, Long> {

   List<Book> findBooksByCategoriesIn(Category
category);

   @Query("select b from Book b join
b.categories c where c.id = :id order by
b.publishDate desc ")
   List<Book> findByCategoriesIn(Integer id);
}

@Repository
public interface CategoryRepository extends
JpaRepository<Category, Integer> {
}
```

2. Spring Boot

Spring Boot is an open source Java-based framework used to create a micro Service. It is used to build stand-alone and production ready spring applications.

- Create a full application in one executable JAR
- Make it quick and easy to create such applications

Version Numbers

● Best not to give a version to dependencies

● Spring Boot has will automatically select the best version to work with (based on parent)

Make a class with a main() your @Configuation

– Spring boot prefers Java Config

– Spring boot requires a main() method

@SpringBootApplication is a composite of: @Configuration, @ComponentScan, @EnableAutoConfiguration

Auto-configuration can't do everything: Needs certain values such as DB user / password

These values can be stored in: application.properties

Spring Boot has profile support

Dev Tools Benefits

● Automatically restarts when it senses a change

● Option to configure resources

Spring Boot uses convention over configuration

Provides dependency management through starters

Uses a @SprinBootApplication with a main() method

Has support for multiple configuration profiles, that can be activated through external configuration

Multiple profiles can be active, indicated by: spring.profiles.active=dev,mysql

Here are just a few of the features in Spring Boot:

- Opinionated 'starter' dependencies to simplify build and application configuration.
- Embedded server to avoid complexity in application deployment.
- Metrics, Helth check, and externalized configuration.
- Automatic config for **Spring** functionality – whenever possible.

```
spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp
@SpringBootApplication
@EnableWebSecurity
Public classApplication {
    Public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
    }}
```

3. Spring Security

Spring Security is a framework that focuses on providing both authentication and authorization (or "access- control") to Java web application and SOAP/RESTful web services

▯ Authentication: Confirming truth of Credentials, Who are you?

▯ Authorization: Define access policy for principal, What can you do?

▯ Principal: User that performs the action, Currently logged in User

▯ GrantedAuthority: Application permission granted to a principal

▯ Roles: coarse-grained permission

▯ SecurityContext: Hold the authentication and other security information

▯ SecurityContextHolder: Provides access to SecurityContext

▯ AuthenticationManager: Controller in the authentication process

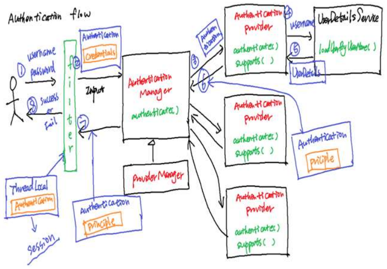
▯ AuthenticationProvider: Interface that maps to a data store which stores your user data.

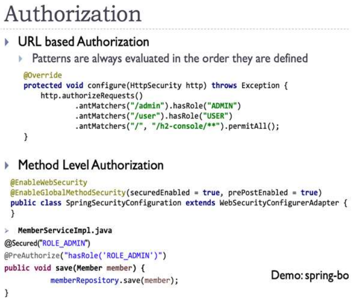
▯ Authentication Object: Object is created upon authentication, which holds the login credentials.

▯ UserDetails: Data object which contains the user credentials, but also the Roles of the user.

▯ UserDetailsService: Collects the user credentials, authorities(roles) and build an UserDetails object.

Spring security architecture





Authorization

- URL based Authorization
  - Patterns are always evaluated in the order they are defined

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
            .antMatchers("/admin").hasRole("ADMIN")
            .antMatchers("/user").hasRole("USER")
            .antMatchers("/", "/h2-console/**").permitAll();
}
```

- Method Level Authorization

```
@EnableWebSecurity
@EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled = true)
public class SpringSecurityConfiguration extends WebSecurityConfigurerAdapter {
}
```

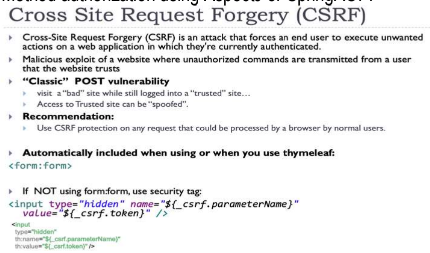
- MemberServiceImpl.java

```
@Secured("ROLE_ADMIN")
@PreAuthorize("hasRole('ROLE_ADMIN')")
public void save(Member member) {
        memberRepository.save(member);
}
```

Demo: spring-bo

Web request authorization using interceptors.

Method authorization using AspectJ or SpringAOP.



Cross Site Request Forgery (CSRF)

- Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.
- Malicious exploit of a website where unauthorized commands are transmitted from a user that the website trusts
- **"Classic" POST vulnerability**
  - visit a "bad" site while still logged into a "trusted" site…
  - Access to Trusted site can be "spoofed".
- **Recommendation:**
  - Use CSRF protection on any request that could be processed by a browser by normal users.
- **Automatically included when using or when you use thymeleaf:**

```
<form:form>
```

- If NOT using form:form, use security tag:

```
<input type="hidden" name="${_csrf.parameterName}"
  value="${_csrf.token}" />
```

```
<input
 type="hidden"
 th:name="${_csrf.parameterName}"
 th:value="${_csrf.token}" />
```

Remember Me

▯ AKA persistent-login authentication

▯ Able to remember the identity of a principal between sessions.

▯ Based on a permanent cookie [default expiration of 2 weeks.]

Remember Me Configuration

▯ Simple Hash-Based Token Approach

- It uses hashing to preserve the security of cookie-based tokens.
- This approach has security issue and is commonly not recommended.
- stores hashed user password in "remember me" cookie – easy to hack.

▯ Persistent Token Approach

- Uses database to store the generated tokens
- Uses combination of randomly generated series and token are persisted, making a brute force attack very unlikely.
- Requires table persistent_logins in database

▯ token-validity defaults to 14 days

```
<security:remember-me data-source-ref="dataSource" token- validity-seconds="86400" remember-me-parameter="keepMe"/>
```

4. JWT

Traditional Authentication System: client - server
User logs in, server checks credentials ->
Session stored in sever, cookie created <-
Send session data to access endpoints ->

Issues with Traditional Systems

 Sessions: Record needs to be stored on server.
 Scalability: With sessions in memory, load increases drastically in a distributed system.
 CORS: When using multiple devices grabbing data via AJAX requests, may run into forbidden requests.
 CRSF: Riding session data to send requests to server from a browser that is trusted via session.

Token-Based Authentication/Authrization Systems

User logs in, server checks credentials ->
Token generated, store in storage/cookie <-
Provide token in headers for all requests ->

Token-based Authorization System

 Stateless: self containded, Scalability: no need to store session in memory CSRF: no session being used, Digitally-signed, Mobile-ready, Decoupled

What is JSON Web Token?

 JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.
 JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA.
 This information can be verified and trusted because it is digitally signed.
 Compact: Because of their smaller size, JWTs can be sent through a URL, POST parameter, or inside an HTTP header. Additionally, the smaller size means transmission is fast. JWT is simply a string in the format of header.payload.signature
 Self-contained: The payload contains all the required information about the user, avoiding the need to query the database more than once.

JSON Web Token Structure: JSON Web Tokens consist of three parts separated by dots (.), which are: header payload signature

Headers are in JSON format encoded in Base64Url. They consists of 2parts: the type of token and the hashing algorithm being used e.g.

{"alg": "HS256", "typ": "JWT" }

Symmetric key crypto (HMAC SHA256) uses single private key (good between trusted parties) and are faster than asymmetric key crypto (RSA SHA256) which uses public/private keys (good between untrusted parties).

JWT payload contains the claims encoded in Base64Url. Claims are statement about the user and additional metadata. 3 types of claims:

Reserved: The JWT specification defines seven reserved claims that are not required, but are recommended to allow interoperability with third-party applications.

Public: These can be defined at will by those using JWTs. But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.

Private: These are the custom claims created to share information between parties that agree on using them.

JWT Signature

This takes the encoded header, payload, a secret and the algo specified in the head and sign it. Ths signature is used to verify authenticity of the sender and the integrity of the message

How does JWT work?

 In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned and must be saved locally (typically in local storage, but cookies can be also used).
 Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema. The content of the header should look like the following: Authorization: Beare <token>

5. OAuth

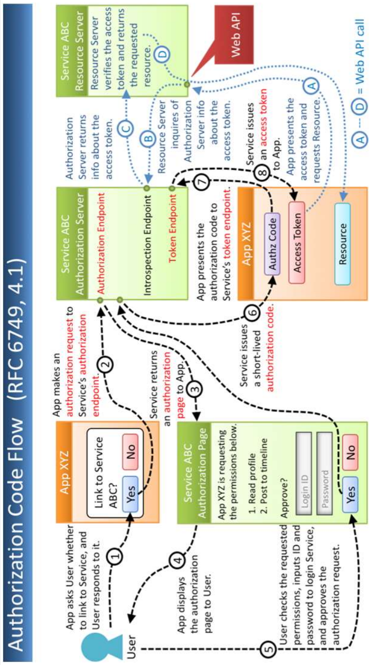
OAuth is an open-standard authorization protocol or framework that provides applications the ability for "secure designated access." It allows an end user's account information to be used by 3rd party services. For example, you can tell Facebook that it's OK for ESPN.com to access your profile or post updates to your timeline without having to give ESPN your Facebook password. This minimizes risk in a major way: In the event ESPN suffers a breach, your Facebook password remains safe

Terminologies

 Resource Owner: The User (who wants to sing up on Quora)
 Resource Server: The API (Google)
 Authorization Server: The resource server hosts the protected user accounts, and the authorization server verifies the identity of the user then issues the access tokens to the application. Often resource server and authorization server are the same.
 Client: The third-party application (Quora)

OAuth 2.0 Grant Flows

 Authentication code grant flow: most commonly used because it is optimized for *server-side applications*, where source code is not publicly exposed, and *Client Secret* confidentiality can be maintained. This is a redirection-based flow, which means that the application must be capable of interacting with the *user-agent* (i.e. the user's web browser) and receiving API authorization codes that are routed through the user-agent.
 Implicit grant flow: The **implicit** grant type is used for mobile apps and web applications (i.e. applications that run in a web browser), where the *client secret* confidentiality is not guaranteed. The implicit grant type is also a redirection-based flow but the access token is given to the user-agent to forward to the application, so it may be exposed to the user and other applications on the user's device. Also, this flow does not authenticate the identity of the application, and relies on the redirect URI (that was registered with the service) to serve this purpose. The implicit grant type does not support refresh tokens.
 Resource owner password credentials grant flow - Username/ password access - With the **resource owner password credentials** grant type, the user provides their service credentials (username and password) directly to the application, which uses the credentials to obtain an access token from the service. This grant type should only be enabled on the authorization server if other flows are not viable. Also, it should only be used if the application is trusted by the user (e.g. it is owned by the service, or the user's desktop OS).
 Client credentials grant flow: The **client credentials** grant type provides an application a way to access its own service account. Examples of when this might be useful include if an application wants to update its registered description or redirect URI, or access other data stored in its service account via the API.



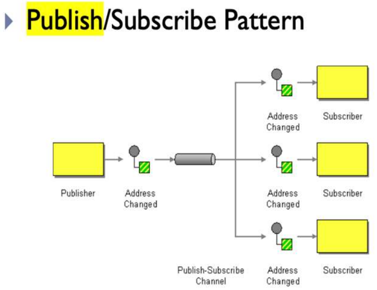
Authorization Code Flow (RFC 6749, 4.1)
Service ABC
Resource Server
Resource Server verifies the access token and returns the requested resource. (D)
Web API
(A) --- (D) = Web API call
Authorization Server returns info about the access token. (C)
(B) Resource Server inquiries of Authorization Server info about the access token.
Service issues an access token to App. (8)
App presents the access token and requests Resource. (A)
Service ABC
Authorization Server
Authorization Endpoint
Introspection Endpoint
Token Endpoint
App presents the authorization code to Service's token endpoint. (7)
App XYZ
Authz Code
Access Token
Resource
App makes an authorization request to Service's authorization endpoint. (2)
Service returns an authorization page to App. (3)
Service issues a short-lived authorization code. (6)
App XYZ
Link to Service ABC?
Yes
No
Service ABC
Authorization Page
App XYZ is requesting the permissions below.
1. Read profile
2. Post to timeline
Approve?
Login ID
Password
Yes
No
App asks User whether to link to Service, and User responds to it. (1)
App displays the authorization page to User. (4)
User checks the requested permissions, inputs ID and password to login Service, and approves the authorization request. (5)
User

OAuth – Authorization Code Flow

 To use this flow, the client application (Quora, Postman, etc) must first register with authorization server.
 The client application is assigned a client ID and a client secret(password) by the authorization server.
 The client ID and secret is unique to the client application on the authorization server.

6. Messaging

Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.



Publish–subscribe is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be. Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

In the publish-subscribe model, subscribers typically receive only a subset of the total messages published. The process of selecting messages for reception and processing is called *filtering*. There are two common forms of filtering: topic-based and content-based.

In a **topic-based** system, messages are published to "topics" or named logical channels. Subscribers in a topic-based system will receive all messages published to the topics to which they subscribe. The publisher is responsible for defining the topics to which subscribers can subscribe.

Advantages: loose coupling and scalability

Disadvantages: message delivery issues

Point-to-Point (P2P) channels are used to establish 1-to-1 communication lines between systems or components. One component publishes a message to the channel so another can pick it up. There can be only one component at each end of the channel.

RabbitMQ

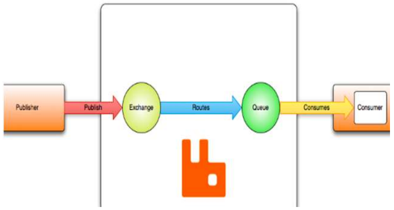
● RabbitMQ is a popular message-oriented middleware server using the AMQP protocol.
Plugin support for STOMP, MQTT, and many others

AMQP Terminology

● Producers: send messages
● Consumers: receive messages
● Broker: the middleware server
● Queue: where messages are stored on the broker
● Exchange: what receives the messages on the broker and routes them to queues

The AMQP specification is defined in several layers: (i) a type system, (ii) a symmetric, asynchronous protocol for the transfer of messages from one process to another, (iii) a standard, extensible message format and (iv) a set of standardised but extensible 'messaging capabilities.'

Basic flow



Bindings: Each Queue should bind with an Exchange with a *routing key.* A link between a queue and an exchange.

Routing key: String of characters, a key that the exchange looks at to decide how to route the message to queues. like an *address* for the message.

Exchange

Direct: Routes messages with a routing key equal to the routing key declared by the binding queue.

Fanout: Routes messages to all bound queues indiscriminately. If a routing key is provided, it will simply be ignored.

Topic: Routes messages to queues whose routing key matches all, or a portion of a routing key

RabbitMQ config

● Exchanges and Queues are not configured through a config file on the broker. They are created with the Broker API by producer/consumer
● Only need to create a queue
– There is a default Exchange (does direct delivery)
– Give the name of the queue and it send it there

Messaging: Messaging protocols are Asynchronous
●Receiving can be Synchronous / Asynchronous
●Messages can be routed in different ways ●Spring RabbitTemplate makes it easy to send and receive AMQP messages