Advanced Java

Agenda

- Hibernate
  - Life Cycle
  - CRUD operations
  - openSession() vs getCurrentSession()
- Spring Hibernate Integration
- Spring JPA Integration
- Spring Testing
  - Unit testing
  - Integration testing
- Spring Security

Hibernate

Hibernate 5 Bootstrapping

- Bootstrapping refers to the process of building and initializing a SessionFactory.
- Hibernate 5 Bootstrapping (using hibernate.cfg.xml)

```
// Create ServiceRegistry.
ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
            .configure() // auto-read hibernate.cfg.xml from CLASSPATH.
            .build();
// Create Metadata.
Metadata metadata = new MetadataSources(serviceRegistry)
        .buildMetadata();
// Create SessionFactory.
SessionFactory factory = metadata.getSessionFactoryBuilder().build();
```

- Hibernate 5 Bootstrapping (using Java config)

```java
ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
    .applySetting("hibernate.connection.driver_class", "com.mysql.cj.jdbc.Driver")
    .applySetting("hibernate.connection.url", "jdbc:mysql://localhost:3306/mobiles_db")
    .applySetting("hibernate.connection.username", "sunbeam")
    .applySetting("hibernate.connection.password", "sunbeam")
    .applySetting("hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect")
    .applySetting("hibernate.show_sql", "true")
    .build();

Metadata metadata = new MetadataSources(serviceRegistry)
        .addAnnotatedClass(Customer.class)
        .buildMetadata();

SessionFactory factory = metadata.getSessionFactoryBuilder().build();
```

- https://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html

ServiceRegistry

- In Hibernate 5.0, a Service is a type of functionality represented by the interface org.hibernate.service.Service.
- ServiceRegistry is interface and represent a standard to add, manage hibernate services.
- Some implementations are StandardServiceRegistry, BootstrapServiceRegistry, EventListenerRegistry, ...

BootstrapServiceRegistry

- Has no parent and holds three required services i.e. ClassLoaderService (interaction with classloaders as per runtime environment/containers), IntegratorService (integration with third party), StrategySelector (short-naming).

StandardServiceRegistry

- Builds on the BootstrapServiceRegistry and holds additional services like SessionFactory builder, Jndi service, Dialect resolver, Jta platform resolver, etc.
  - hibernate.connection.datasource = java:comp/env/jdbc/myconnpool

Metadata

- Represents application's domain model & its database mapping.

Hibernate: openSession() vs getCurrentSession()

- openSession()
  - Create new hibernate session.
  - It is associated with JDBC connection (autocommit=false).
  - Can be used for DQL (get records), but cannot be used for DML without transaction.
  - Should be closed after its use.
- getCurrentSession()
  - Returns session associated with current context. If no session is available for given context, new session will be created and attached to it.
  - Current session context is configured as hibernate.current_session_context_class.
    - thread: Session is stored in TLS (Thread Local Storage).
    - jta: Session is stored in transaction-context given by JTA providers (like app servers).
    - custom: User implemented context.
  - https://docs.jboss.org/jbossas/javadoc/7.1.2.Final/org/hibernate/context/spi/CurrentSessionContext.html
  - This session is not attached with any JDBC connection (by default).
  - JDBC connection is associated with it, when a transaction is created. The connection is given up, when transaction is completed.
  - The session is automatically closed, when scope is finished. It should not be closed manually.

Hibernate Entity Life Cycle

- Hibernate entity can have one of four states.
- Transient
  - New Java object of entity class.
  - This object is not yet associated with hibernate.
- Persistent
  - Object in session cache.

    - For all objects created by hibernate or associated with hibernate.
    - State is tracked by hibernate and updated in database during commit.
    - Never garbage collected.
- Detached
    - Object removed from session cache.
- Removed
    - Object whose corresponding row is deleted from database.

Hibernate Caching

- Hibernate caches are used to speed up execution of the program by storing data (objects) in memory and hence save time to fetch it from database repeatedly.
- There are two caches
    - Session cache (L1 cache)
    - SessionFactory cache (L2 cache)

Hibernate Session Cache

- Collection (`Map<Serializable,Object>`: Key=Primary Key, Value=Entity Object + Flags) of entities per session – Persistent objects.
    - Flags = New or Deleted or Persistent or Modified (Dirty).
- Hibernate keep track of state of entity objects and update into database.
- Session cache cannot be disabled.
- If object is present in session cache, it is not searched into session factory cache or database.
- Use refresh() to re-select data from the database forcibly.

Hibernate SessionFactory Cache

- Collection of entities per session factory. Entities are stored in serialized form (not as java objects).
- By default disabled, but can be enabled and configured into hibernate.cfg.xml

```
<property name="hibernate.cache.region.factory_class">
    org.hibernate.cache.ehcache.EhCacheRegionFactory
```

```
</property>
<property name="hibernate.cache.use_second_level_cache"> true </property>
```

- Need to add respective additional second cache jars in project and optional cache config file (e.g. ehcache.xml).

```
<ehcache>
    <cache name="pkg.EntityClass" maxElementsInMemory="1000" />
</ehcache>
```

- Four types of second level caches: EHCache, Swarm Cache, OS Cache, Tree Cache
  - EHCache: It can cache in memory or on disk and clustered caching and it supports the optional Hibernate query result cache.
  - Swarm Cache: A cluster cache based on JGroups. It uses clustered invalidation, but doesn't support the Hibernate query cache.
  - OSCache (Open Symphony Cache): Supports caching to memory and disk in a single JVM with a rich set of expiration policies and query cache support.
  - JBoss Tree Cache: A fully transactional replicated clustered cache also based on the JGroups multicast library. It supports replication or invalidation, synchronous or asynchronous communication, and optimistic and pessimistic locking.
- Use @Cache on entity class to cache its objects.
- Decide cache policy for those object and specify into its usage attribute: READ_ONLY, READ_WRITE, NONSTRICT_READWRITE, TRANSACTIONAL Stores the objects into the map (with its id as key). So lookup by key is very fast. Note that not all cache support all strategies.

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class EntityClass {
    // ...
}
```

Hibernate CRUD methods

- get() or find(): Find the database record by primary key and return it. If record is not found, returns null. find() is JPA compliant method.
- load(): Returns proxy for entity object (storing only primary key). When fields are accessed on proxy, SELECT query is fired on database and record data is fetched. If record not found, ObjectNotFoundException is thrown.
- save(): Assign primary key to the entity and execute INSERT statement to insert it into database. Return primary key of new record.
- persist(): Add entity object into hibernate session. Execute INSERT statement to insert it into database (for all insertable columns) while committing the transaction. persist() is JPA compliant.
- update(): Add entity object into hibernate session. Execute UPDATE statement to update it into database while committing the transaction. All (updateable) fields are updated into database (for primary key of entity).
- saveOrUpdate() or merge(): Execute SELECT query to check if record is present in database. If found, execute UPDATE query to update record; otherwise execute INSERT query to insert new record. merge() is JPA compliant.
- delete() or remove(): Delete entity from database (for primary key of entity) while committing the transaction. remove() is JPA compliant.
- evict() or detach(): Removes entity from hibernate session. Any changes done into the entity after this, will not be automatically updated into the database. detach() is JPA compliant.
- clear(): Remove all entity objects from hibernate session.
- refresh(): Execute SELECT query to re-fetch latest record data from the database.

Hibernate: get() vs load()

- get(): Find the database record by primary key and return it. If record is not found, returns null. find() is JPA compliant method.
- load(): Returns proxy for entity object (storing only primary key). When fields are accessed on proxy, SELECT query is executed on database and record data is fetched. If record not found, ObjectNotFoundException is thrown.

Hibernate: persist() vs save()

- persist(): Add entity object into hibernate session. Return type is void. Execute INSERT statement to insert it into database (for all insertable columns) while committing the transaction. persist() is JPA compliant.
- save(): Assign primary key to the entity and execute INSERT statement to insert it into database. Return primary key of new record.

Hibernate: saveOrUpdate() vs merge()

- Execute SELECT query to check if record is present in database. If found, execute UPDATE query to update record; otherwise execute INSERT query to insert new record. merge() is JPA compliant.

JPA

- JPA is specification for ORM.
- JPA specifications are given in form of interfaces and annotations -- javax.persistence package.
    - Annotations: @Entity, @Table, @Column, @Id, @Transient, @Temporal, @OneToMany, @ManyToOne, ...
    - Interfaces:
        - EntityManagerFactory -- create the entity manager -- Hibernate SessionFactory is an impl of EntityManagerFactory.
        - EntityManager -- encapsulate JDBC connection -- Hibernate Session is an impl of EntityManager.
            - find(), persist(), merge(), remove(), detach(), ...
        - Transaction -- tx management.
        - Query -- represent jpql queries.
    - JPQL query language.
- All ORM implementations follow JPA specification e.g. Hibernate, Torque, iBatis, EclipseLink, ...
- Hibernate implements JPA specs.
    - SessionFactory extends EntityManagerFactory
    - Session extends EntityManager
        - find(), persist(), merge(), refresh(), remove(), detach(), ...
    - HQL is similar to JPQL.
- Traditionally JPA is configured with persistence.xml (similar to hibernate.cfg.xml).
- JPA versions
    - Java Persistence API: 1.0, 1.1, 2.0, 2.1, 2.2
    - Jakarta Persistence API: 2.2, 3.0, 3.1

JPA Entity life cycle

- JPA Entity life cycle is similar to Hibernate entity life cycle.
    - New (Transient)
    - Managed (Persistent)
    - Detached (Detached)
    - Removed (Removed)
- JPA also enables implementing callback methods to handle life cycle events.
    - Before persist is called for a new entity – @PrePersist

- After persist is called for a new entity – @PostPersist
  - Auto-generated id will be available here.
- Before an entity is removed – @PreRemove
- After an entity has been deleted – @PostRemove
- Before the update operation – @PreUpdate
  - Called only if object is modified.
- After an entity is updated – @PostUpdate
- After an entity has been loaded – @PostLoad

* Callback methods are implemented in the entity class.

```
@Entity
class Customer {
    // ...
    @PostPersist
    public void postPersist() {
        System.out.println("postPersist() : " + this);
    }
}
```

Spring Hibernate Integration

* Create Maven Project.
* Add Spring-ORM and Hibernate dependencies in pom.xml.

```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.6.15.Final</version>
    </dependency>
    <dependency>
```

```
            <groupId>com.mysql</groupId>
            <artifactId>mysql-connector-j</artifactId>
            <version>8.1.0</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.3.30</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-orm</artifactId>
            <version>5.3.30</version>
        </dependency>
    </dependencies>
```

- Create entity classes with ORM annotations.
- Create resources/hibernate.cfg.xml to provide dialect and entity classes.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQL8Dialect</property>
        <property name="hibernate.show_sql">true</property>

        <mapping class="com.sunbeam.Category"/>
    </session-factory>
</hibernate-configuration>
```

- Configure data source properties and hibernate related spring beans in resources/beans.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
        http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context-
4.3.xsd
        http://www.springframework.org/schema/tx https://www.springframework.org/schema/tx/spring-tx-4.3.xsd">
    <context:component-scan base-package="com.sunbeam"/>

    <bean id="dataSrc" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/test"/>
        <property name="username" value="nilesh"/>
        <property name="password" value="nilesh"/>
    </bean>

    <bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSrc"/>
        <property name="configLocation" value="classpath:hibernate.cfg.xml"/>
    </bean>

    <bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <tx:annotation-driven transaction-manager="transactionManager"/>
</beans>
```

- Create DAO class and autowire SessionFactory in it.

```
@Repository
public class CategoryDao {
    @Autowired
    private SessionFactory sessionFactory;

    @Transactional
    public List<Category> findAll() {
        String hql = "FROM Category c";
        Session session = sessionFactory.getCurrentSession();
        Query<Category> q = session.createQuery(hql);
        return q.getResultList();
    }
}
```

- Create application context in main(), get DAO bean and invoke its methods.

```
public class Main {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        ctx.registerShutdownHook();

        CategoryDao dao = ctx.getBean(CategoryDao.class);
        List<Category> list = dao.findAll();
        list.forEach(System.out::println);
    }
}
```

Spring JPA Integration

- Create Maven Project.
- Add Spring-ORM and Hibernate dependencies in pom.xml.

```xml
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>5.6.15.Final</version>
    </dependency>
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <version>8.1.0</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.3.30</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>5.3.30</version>
    </dependency>
</dependencies>
```

- Create entity classes with ORM annotations.
- Create resources/persistence.xml to provide jpa provider and entity classes.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="blogs" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <class>com.sunbeam.Category</class>
    </persistence-unit>
</persistence>
```

- Configure data source properties and JPA related spring beans in resources/beans.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
https://www.springframework.org/schema/beans/spring-beans-4.3.xsd
        http://www.springframework.org/schema/context https://www.springframework.org/schema/context/spring-context-
4.3.xsd
        http://www.springframework.org/schema/tx https://www.springframework.org/schema/tx/spring-tx-4.3.xsd">
    <context:component-scan base-package="com.sunbeam"/>

    <bean id="dataSrc" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.cj.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost:3306/test"/>
        <property name="username" value="nilesh"/>
        <property name="password" value="nilesh"/>
    </bean>

    <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSrc"/>
        <property name="persistenceXmlLocation" value="classpath:persistence.xml"/>
    </bean>

    <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory"/>
    </bean>
    <tx:annotation-driven transaction-manager="transactionManager"/>

    <bean id="persistenceExceptionTranslationPostProcessor"
class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />
</beans>
```

- Create DAO class and autowire EntityManager in it.

```
@Repository
public class CategoryDao {
    //@PersistenceUnit(name="blogs")
    //private EntityManagerFactory emf;
        // to get EntityManager from factory:  emf.createEntityManager();

    @PersistenceContext
    private EntityManager em;

    @Transactional
    public List<Category> findAll() {
        String hql = "FROM Category c";
        Query q = em.createQuery(hql);
        return q.getResultList();
    }
}
```

- Create application context in main(), get DAO bean and invoke its methods.

```
public class Main {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        ctx.registerShutdownHook();
        CategoryDao dao = ctx.getBean(CategoryDao.class);
        List<Category> list = dao.findAll();
        list.forEach(System.out::println);
    }
}
```

Testing

Unit Testing

- Testing each unit individually.
- JUnit is Java's framework for Unit testing.
- Example: Java -- JUnit.

```
// class to be tested
public class MyClass {
    public int add(int a, int b) {
        return a + b;
    }
    // ...
}
```

- JUnit Test class

```
public class MyClassTests {
    @Test
    public void testAdd() {
        MyClass obj = new MyClass();
        int result = obj.add(12, 5);
        assertEquals(17, result); // arg1: expected, arg2: actual
        }
    }
```

- Serveral assert methods are given in org.junit.jupiter.api.Assertions class to check the test cases.

Spring -- Unit Testing

- Spring auto-creates required application context and initialize (auto-wire) all required beans.
- Spring Boot -- @SpringBootTest -- auto-configure all test frameworks e.g. JUnit, JMockito, Local Web Server, ...
- @SpringBootTest -- auto-configure w.r.t. project dependencies and application.properties.

Spring -- Unit Testing with Mocking

- To test an dependent object individually by mocking dependency object.
- JMockito library is used to mock the Java objects.

```
// in test class mock the object
@MockBean
private MovieDao movieDao;
```

```
// in @Test method, mention results expected from mocked objects for given input
Movie m = new Movie(100, "Inception", Date.valueOf("2010-04-08"));
when(movieDao.findById(100)).thenReturn(m);
// then implement the test case
```

- MockMvc object is used to mock the Spring MVC. This mocks HTTP requests to the controllers without running the web server.

```
mockMvc
    .perform(get("/movies/100"))
```

```
        .andDo(print())
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.status", "success").exists())
        .andExpect(jsonPath("$.data", content().json(mJson)).exists());
```

Spring -- Integration Testing

- To test dependent object with dependency object functionality.
- Spring REST controller testing is done using TestRestTemplate.

```java
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
class MovieRestControllerIntegrationTests {
    @LocalServerPort
    private int port;
    @Autowired
    private TestRestTemplate restTemplate;
    @Autowired
    private ObjectMapper objectMapper;

    @SuppressWarnings("unchecked")
    @Test
    void testGetMovie() throws Exception {
        String url = "http://localhost:"+port+"/movies/"+1;
        Map<String,Object> result = restTemplate.getForObject(url, Map.class);
        assertThat(result).containsEntry("status", "success");
        Movie movie = objectMapper.convertValue(result.get("data"), Movie.class);
        assertEquals(1, movie.getId());
        System.out.println("Movie Received: " + movie);
    }
}
```

Spring Security

- Not in DMC Syllabus
- Security is essential part of any application.
  - Web MVC application
  - REST services
  - Micro-services
- Spring security enables application level security.
  - Username/password authentication
  - Single-Sign-on SSO (Facebook, Google, Okta, etc).
  - Role based security (in application)
  - OAuth - Application authorization with another application
  - Micro-services security (JWT)
  - Method level security
- Spring security is separate spring project.
  - Not part of Spring core framework.
  - Not part of Spring boot.
  - Spring security has separate release cycle.
  - Spring Boot provides auto-configuration for Spring security.

Spring security terminologies

- Authentication
  - Who are you?
  - Identity proof
    - Knowledge based authentication
      - Username and password
      - Secret question
      - Combination with other details
      - Single-Sign-on
    - Possession based authentication
      - Email/SMS code
      - QR-Code authentication using Mobile

        - Swipe cards/RSA tokens
        - Bio-metric
      - Multifactor authentication
        - Combination of multiple options
- Authorization
  - What you are allowed to do?
  - Role based
  - Different users in application have different access - Banking System
    - Cashier
    - Branch Manager
    - Loan officer
- Principal
  - User of system identified by process of authentication i.e. Currently logged in user/account
  - Principal is stored by spring application to track user.
- Granted Authority
  - Well-defined actions for which user is authorized/allowed to.
  - Banking users are authorized for differnent actions.
    - Cashier
      - view customer balance
      - deposit/withdraw from customer account
      - manage/tally daily cash
    - Branch Manager
      - approve resources for branch
      - assign/monitor responsibilities
      - review branch business
  - Authorities are fine-grained
- Roles
  - Group of authorities (multiple authorities)
  - Coarse-grained authorities/permissions
  - Helps assigning set of authorities to users on similar role/position.
    - ROLE_CASHIER

    - ROLE_BRANCH_MANAGER
    - ROLE_LOAN_OFFICER

Spring Security in web MVC

- Spring security is based on "set" of Java EE filters called springSecurityFilterChain.
- Filters in this chain intercept each request and validate access to resource against Principal/Granted authority.
- These filters also perform default actions (login, logout, store principal, invalidate session/cookie) as appropriate.
- In Spring boot application spring security is auto-configured when added on classpath.
- Default behaviour of Spring security
  - Add authentication to URLs (except /error)
  - Add default login form
  - Create default user (name: user) and password (on console).
  - Handles login error
  - Store principal in HttpSession upon success
  - Logout on /logout URL
- Spring security is very flexible with little configuration options.

Username/Password Authentication

- AuthenticationManager:
  - Responsible for user authentication -- authenticate() method -- returns Authentication object.
  - Internally manages/uses AuthenticationProvider.
  - Built using AuthenticationManagerBuilder.
- Authentication:
  - Represents the token for an authentication request or for an authenticated principal once the request has been processed by the AuthenticationManager.
  - It holds credentials before login and principal object corresponding to successfully logged in user.
  - Typically principal object is simply UserDetails object - holding user information and authorities.
  - Authentication will be typically stored in a thread-local SecurityContext managed by the SecurityContextHolder by the authentication mechanism.
  - Methods:
    - getCredentials()

        - getPrincipal()
        - getAuthorities()
        - isAuthenticated()
        - setAuthenticated()
    - Available Authentication implementations:
        - UsernamePasswordAuthenticationToken
        - JwtAuthenticationToken
        - OAuth2AuthenticationToken
        - ...
- AuthenticationManagerBuilder:
    - Follow builder design pattern to build AuthenticationManager.
        - inMemoryAuthentication()
        - jdbcAuthentication()
        - ldapAuthentication()
- AuthenticationProvider:
    - AuthenticationManager delegates the fetching of persistent user information to one or more AuthenticationProviders.
    - Available AuthenticationProvider implementations:
        - DaoAuthenticationProvider
        - OpenIDAuthenticationProvider
        - LdapAuthenticationProvider
        - ...
    - If supports() given Authentication object type, then authenticate() it.
    - Responsible for user authentication -- authenticate() method -- returns Authentication object.
        - Input: Authentication object (hold credentials)
        - Output: Authentication object (hold principal)
        - Typically clears credentials after authentication.
- UserDetailsService:
    - AuthenticationProvider use UserDetailsService service to load the user details.
        - UserDetails loadUserByUsername(java.lang.String username);
    - Available implementations are:
        - InMemoryUserDetailsManager

    - JdbcDaoImpl
    - LdapUserDetailsManager
    - ...
- UserDetails:
  - UserDetailsService returns UserDetails for given username.
  - It hold details like username, password, authorities, enabled, etc.
  - Available implementations are:
    - User
    - LdapUserDetailsImpl
    - ...

Authorization

- HttpSecurity:
  - Configures security using builder design pattern.
  - Allow URL for given role/user.
    - Using antMatchers.
  - Configure 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 are currently authenticated.
    - https://portswigger.net/web-security/csrf
    - https://portswigger.net/web-security/csrf/tokens
    - https://spring.io/blog/2013/08/21/spring-security-3-2-0-rc1-highlights-csrf-protection/
  - Configure CORS (Cross-Site Resource Sharing).
    - Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources on a web page to be requested from another domain outside the domain from which the first resource was served.
    - https://portswigger.net/web-security/cors

JWT (JSON Web Token)

- JWT is a open standard (RFC 7519) to transfer data among two parties securely.
- Spring Web MVC applications store Authentication token into HttpSession, so that authorization info can be used over sub-sequent requests.
- Spring REST services (typically micro-services) are stateless and do not use state management mechanisms like session/cookie.

- JWT is used to maintain client state on client side itself like cookies (but signed -- cannot be tampered).
- JWT is signed (encrypted) web token that cannot be tampered by the user (until it has secret).
- JWT has three parts
  - Header: Encryption algorithm & token type (JWT)
  - Payload: JSON data to be stored.
  - Signature: JWT token secret
- JWT tokens can be inspected at jwt.io.
- JWT tokens can be generated/interpreted using specialized libraries.
  - Apache JJWT.

Authorization using JWT

- JWT is designed for authorization after authentication process is completed keeping desired details into JWT payload.
- Typical process:
  - Client sends credentials to the server.
  - Server authenticate the user, put client identity into JWT payload and send to client.
  - Client store JWT information into local storage or cookie or some other mechanism.
  - While accessing a protected resource, client sends JWT to the server (typically in header).
  - Server get JWT, validates it and retrieve the payload information.
  - Based on client identity, server allows/denies access to the resource.
- Few considerations
  - JWT (payload) can be easily decrypted/viewed. So never store sensitive client information in it.
  - JWT can be stolen i.e. another client may access protected resources using JWT tokens of some other client.
  - JWT cannot be disabled (unlike session), but can be expired.

JWT Authentication/Authorization

- Step 1: Create Spring Boot Starter application with Web, MySQL, Spring Data JPA, Security starter, JWT Library.
- Step 2: Create UserOpsController and AdminOpsController -- @RestController.
  - /user/welcome mapping returns "User " + username
  - /admin/welcome mapping returns "Admin " + username

- Step 3: Create users table in MySQL with columns id, email, password, mobile, enabled, and role. Insert a few records there for testing. Use https://bcrypt-generator.com/ for encrypting password.
- Step 4: In application.properties, do necessary JPA/DB config.
- Step 5: Create AppUser entity class.
- Step 6: Create AppUserDao interface with findByEmail() method.
- Step 7: Implement UserDetailsService in a user-defined class (AppUserService). loadUserByUsername() using JPA repository. Convert return value into spring's User class and return.
- Step 8: Implement JwtUtil class for JWT common operations createToken() and validateToken().
- Step 9: Create @RestController AuthController to authenticate user with POST request (having email & passwd in body) and generate JWT token if user is valid. Use AuthenticationManager (@Autowired) to authenticate user.
- Step 10: Implement SecurityConfig @Configuration class.
  - Add @EnableWebSecurity on the class -- optional.
  - Provide a BCryptPasswordEncoder bean.
  - Create beans AuthenticationManager and SecurityFilterChain as given.
- Step 11: In SecurityConfig, create AuthenticationManager bean using AuthenticationManagerBuilder. Attach userDetailsService().
  - This step can be done using AuthenticationConfiguration.
- Step 12: In SecurityConfig, create SecurityFilterChain bean.
  - Allow / and /signin for all users.
  - Allow /user/welcome only for ROLE_USER.
  - Allow /admin/welcome only for ROLE_ADMIN.
- Step 13: Add Swagger dependency in pom.xml and given SwaggerConfig class in config. and Test /signin endpoint with swagger/postman. Json body include email and password.
  - Ensure that generated response contains valid payload (using jwt.io).
- step 14: Test /user/welcome or /admin/welcome endpoint by adding JWT token into Authorization header with Bearer prefix. It will not work.
- Step 15: Create JwtFilter inherited from OncePerRequestFilter. Mark @Component for DI. In its doFilterInternal() method.
  - Check if appropriate "Authorization" header is available (with prefix Bearer).
  - Get the token from the header value (by stripping Bearer prefix).
  - Validate the token. If wrong, it will throw exception.
  - If valid JWT token, create and attach an Authentication token to current SecurityContext (if no Authentication already set).
  - Execute the next filter in the chain.
- Step 16. Add Authorization settings in SecurityConfig's SecurityFilterChain bean creation.

  - Ensure that there is no HttpSession created, by setting SessionCreationStrategy as STATELESS.
  - Add jwtFilter before UsernamePasswordAuthenticationFilter in spring security filter chain.
- Step 17: Test application with user and password of different roles.
  - First make request to /signin to get JWT token.
  - Use JWT token in Authorization header of request to /user/welcome and /admin/welcome.