Design Patterns in Blockchain-Based-ML-Deployment

Singleton Pattern

Spring's default bean scope is singleton, meaning there's only one instance of each bean in the app lication context:

Java

```
@Service

public class TokenProvider {

   // Only one instance is created by Spring container }
```

Java

```
@Bean

public PasswordEncoder passwordEncoder() {

   return new BCryptPasswordEncoder();

}
```

Implementation Details:

- Spring container ensures a single instance of each bean
- The @Service, @Component, @Repository, and @Controller annotations create singleton b eans
- Explicitly defined beans using @Bean are also singletons by default

Factory Pattern

Spring uses the factory pattern to create and configure beans:

Java

```
@Bean

public AuthenticationManager authenticationManager(AuthenticationConfiguration
authenticationConfiguration) throws Exception {
    return authenticationConfiguration.getAuthenticationManager();
}


@Bean
CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();
    // Configure the object
    // ...
    return source;
}
```

Implementation Details:

- @Bean methods act as factory methods
- They centralize the creation and configuration of complex objects
- Allow for different implementations based on configuration

Builder Pattern (via Lombok)

The Builder pattern is implemented using Lombok's @Builder annotation:

Java

```java
@Data
@EqualsAndHashCode(exclude = {"password"})
@ToString(exclude = {"password"})
@Document(collection = "users")
public class User {
    // Properties
}
```

Implementation Details:

- Lombok generates builder methods for model classes

- Makes object creation more readable and flexible
- Allows for constructing complex objects with optional parameters

}

Facade Pattern

Services act as facades, providing a simplified interface to a complex subsystem:

Java

```java
@Service
public class UserService {
   // Dependencies

   @Transactional
   public User updateProfile(UserPrincipal currentUser, ProfileUpdateRequest request) {
      // Implementation that coordinates multiple operations
   }

   // Other methods
}
```

Implementation Details:

- Provides a higher-level interface to a subsystem
- Hides implementation complexity from clients
- Coordinates multiple operations in a single transaction
- Centralizes business logic

Adapter Pattern

The CustomUserDetailsService adapts between Spring Security's model and the application's mod
el:

Java

```java
@Service
public class CustomUserDetailsService implements UserDetailsService {
  @Override
  @Transactional
  public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
    User user = userRepository.findByEmail(email)
        .orElseThrow(() -> new UsernameNotFoundException("User not found with email : " + email));

    List<GrantedAuthority> authorities = user.getRoles().stream()
        .map(role -> new SimpleGrantedAuthority(role.name()))
        .collect(Collectors.toList());

    return UserPrincipal.create(user, authorities);
  }
}
```

Implementation Details:

- Adapts domain User to Spring Security's UserDetails
- Converts domain roles to Spring Security's GrantedAuthority
- Allows Spring Security to work with the application's domain model

Command Pattern

The CommandLineRunner implementation in SecurityConfig acts as a command object:

Java

```
@Bean
CommandLineRunner initAdminUser(UserRepository userRepository, PasswordEncoder passwordEncoder) {
  return args -> {
    String adminEmail = "admin@example.com";
    String adminPassword = "secureAdminPassword123!";
    if (!userRepository.existsByEmail(adminEmail)) {
      User adminUser = new User();
      adminUser.setName("Administrator");
      adminUser.setEmail(adminEmail);
      adminUser.setPassword(passwordEncoder.encode(adminPassword));
      adminUser.setRoles(Set.of(Role.ROLE_ADMIN, Role.ROLE_USER));
      userRepository.save(adminUser);
      logger.info(">>> Created initial admin user: {}", adminEmail);
    } else {
      logger.info(">>> Admin user {} already exists.", adminEmail);
    }
  };
}
```

Implementation Details:

- Encapsulates a request as an object
- Lambda function acts as the command implementation
- Gets executed during application startup
- Allows parameterizing initialization logic

Strategy Pattern

The authentication mechanism in Spring Security implements the strategy pattern:

Java

```
@Bean
```

```
public AuthenticationManager authenticationManager(AuthenticationConfiguration
authenticationConfiguration) throws Exception {
   return authenticationConfiguration.getAuthenticationManager();
}
```

Implementation Details:

- AuthenticationManager defines a contract for authentication strategies
- Different authentication providers can be plugged in
- The application uses username/password authentication strategy
- The strategy can be changed without modifying client code

Proxy Pattern

The Spring Security filter acts as a proxy:

Java

```
@Override
protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
response,
                                FilterChain filterChain) throws ServletException, IOException {
   try {
      String jwt = getJwtFromRequest(request);


      if (StringUtils.hasText(jwt) && tokenProvider.validateToken(jwt)) {
         String userId = tokenProvider.getUserIdFromToken(jwt);
         UserDetails userDetails = customUserDetailsService.loadUserById(userId);
         UsernamePasswordAuthenticationToken authentication = new
UsernamePasswordAuthenticationToken(
               userDetails, null, userDetails.getAuthorities());
         authentication.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
         SecurityContextHolder.getContext().setAuthentication(authentication);
```

```
        }
    } catch (Exception ex) {
        logger.error("Could not set user authentication in security context", ex);
    }

    filterChain.doFilter(request, response);
}
```

Implementation Details:

- Intercepts requests before they reach the target controllers
- Performs authentication checks and establishes security context
- Delegates to the actual handler if security checks pass
- Controls access to protected resources