Application Security

From A Developer's Point of View

About Me

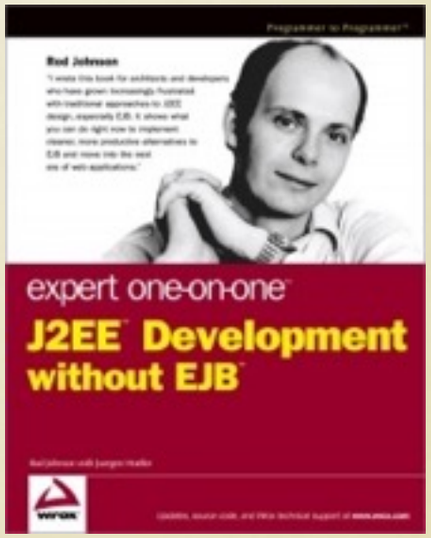
- Wolfgang Giersche
  Nuclear Physicist and Java Native
- 2006 - 2010 Lectures Enterprise Computing
- 9 years with zühlke empowering ideas
- currently running a competence unit of 50+ talented Java developers
- Still hacking…

About the Lecture

- Special Subjects from a developer's point of view
- Subjects: Security and Persistence
- With Spring, but could've been JEE, too

History of JEE

- 1999 J2EE 1.2: JSP, Servlet, EJB, RMI
  A specification implemented within application servers
- 2002 Rod Johnson introduces the Spring Framework
- Spring soon becomes the “better” alternative to J2EE
- 2006: JEE 5 on Java 1.5 wins back confidence
- 2014: JEE7 competes with Spring 4.1
- Servers: Oracle Glassfish and Red Hat Wildfly.



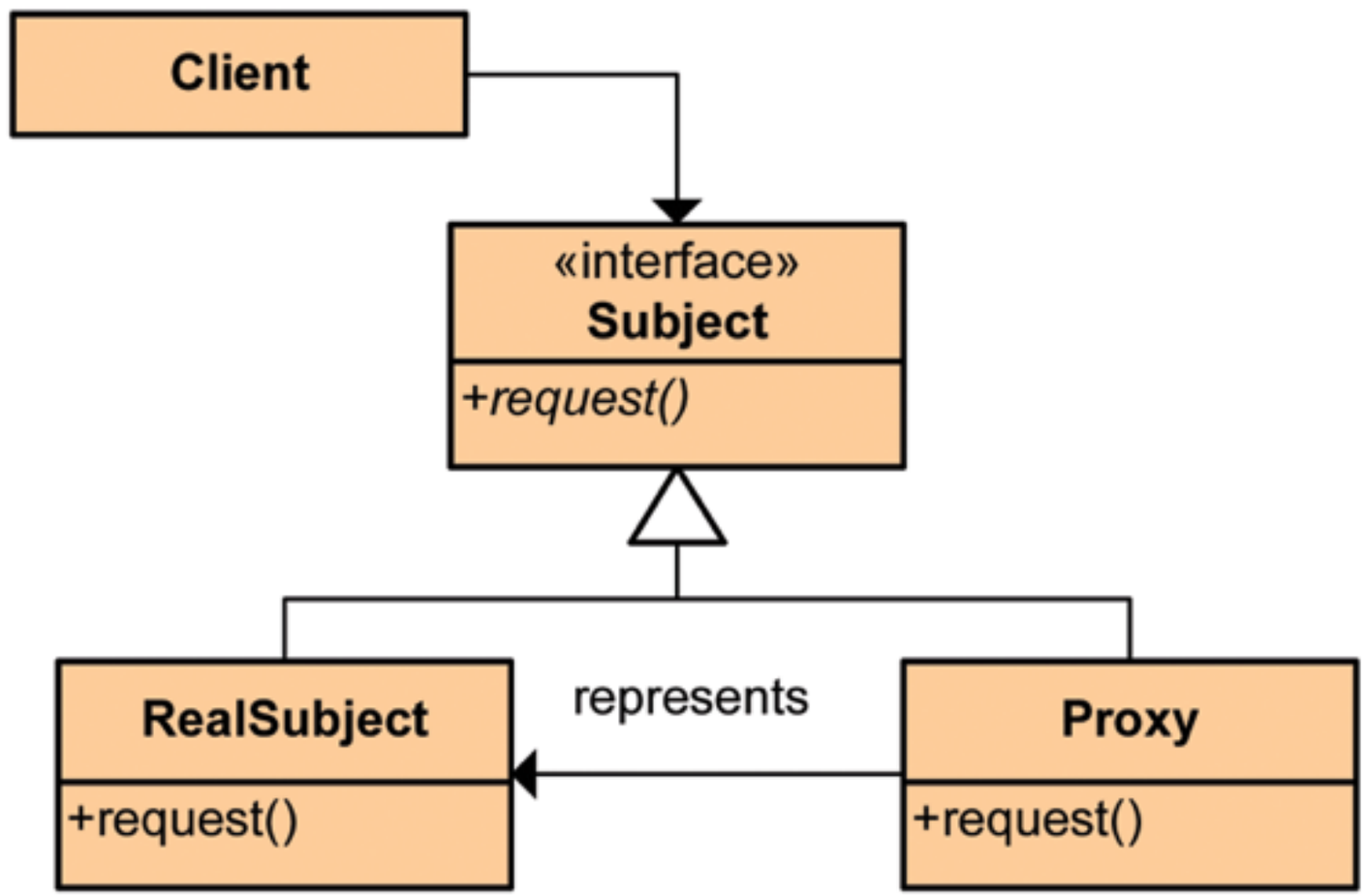
Springframework Very Short Introduction

- ApplicationContext is the central BeanFactory
- The Context contains the beans to be injected
- Annotations activate different aspects
- Transactions, Security Aspects and some more come with the framework
- We can add any new aspect easily

The Proxy Pattern

Proxy

Type: Structural



What it is:
Provide a surrogate or placeholder for another object to control access to it.

Essential Modules

- Spring-Context
- Spring-Security
- Spring-Data (we'll see JPA and Mongo)
- Spring-Batch
- Spring-WebMVC
- Spring Integration
- many, many more

Unit Tests with Spring

- @RunWith Annotation defines the Runner
- Spring Test Runner loads context
- …and injects dependencies
- then starts the @Test methods

The Test Class

```
3   import ...
21
22  /**
23   * This is a scenario test - not a unit test. It tests not only functions
24   * as implemented in methods but also the wiring of the context.
25   * This particular scenario test is actually used as a demo for the
26   * various aspects that come with implementing advanced security concepts
27   * with Spring Security.
28   */
29  @RunWith(SpringJUnit4ClassRunner.class)
30  @ContextConfiguration(classes = MultiTenantSecurityTestContext.class)
31  public class MultiTenantSecurityTest {
32
33      @Autowired
34      private ContractRepository contractRepository;
35
36      @Autowired
37      private UserFactory userFactory;
38
39      @Autowired
40      private AuthenticationManager authenticationManager;
41
```

Configuration Class

```
package org.smurve.hsr2014.security;

import ...


@Configuration
@EnableAspectJAutoProxy
@Import(value = {SpringJpaConfiguration.class, SpringSecurityContext.class})
@EnableJpaRepositories(basePackages = "org.smurve.hsr2014.repo")
@EnableGlobalMethodSecurity(prePostEnabled = true)
@ComponentScan(basePackages = {
        "org.smurve.hsr2014.security.restrictions",
        "org.smurve.hsr2014.repo"})
public class MultiTenantSecurityTestContext {

    @Bean
    public DatabaseConnector databaseConnector() { return new HSqlConnector(); }

    @Bean
    public HsqlDbHelper hsqlDbHelper() { return new HsqlDbHelper(); }

}
```

Spring-Data-JPA

- Interfaces simply extend JpaRepository
- Context scans the ClassPath for those interfaces
- Method names are “understood” by Spring
- Spring provides the implementation of the interface and injects it where required
- Massive ease for simple DB access functionality

Spring-Data-JPA

```
@Configuration
@EnableAspectJAutoProxy
@Import(SpringJpaConfiguration.class)
@EnableJpaRepositories(basePackages = "org.smurve.hsr2014.repo")
@ComponentScan(basePackages = {
  "org.smurve.hsr2014.security.restrictions",
  "org.smurve.hsr2014.repo"})
public class SpringSecurityContext {
```

```
package org.smurve.hsr2014.repo;

import ...

public interface TenantRepository extends JpaRepository<Tenant, String> {

    public Tenant findByTenantId(String tenantId);
}
```

Conceptual Background

- Quality attributes
- Authentication basics
- Access control semantics
- Different access control models
- Resources to be secured
- Plead for AOP

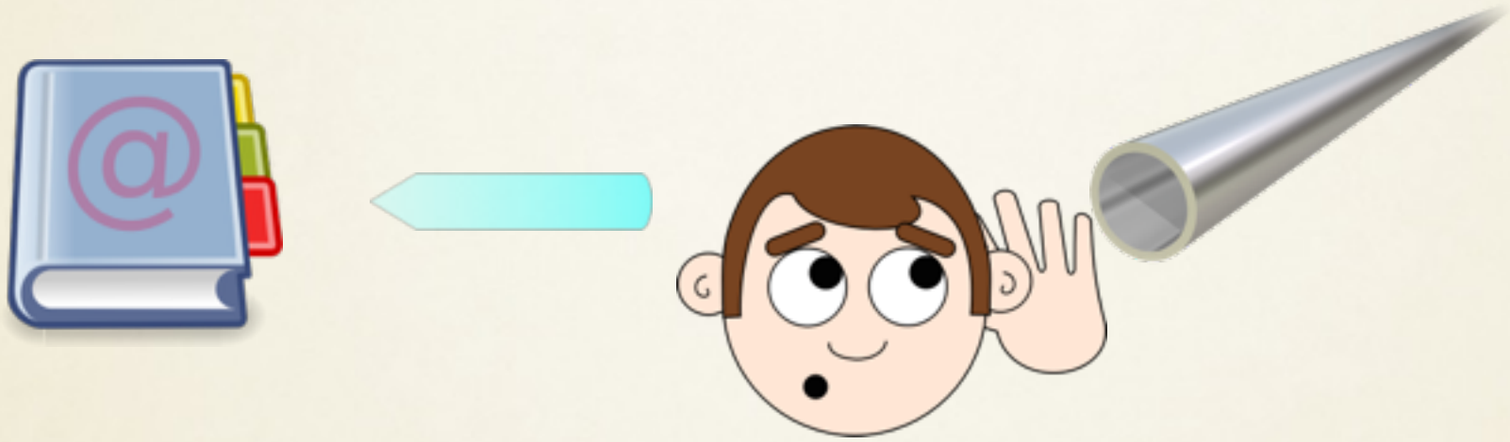
Security-Related Quality Attributes

- Confidentiality
- Integrity
- Availability
- Authenticity
- Non-repudiation
- Controlled Access
- Auditability

Authentication from a developers view

- What exactly is authenticated?
- The request?
- The session?
- The current Thread of Execution?
- The Message (if asynchronous)?
- There's always some kind of trust involved

Authentication Semantics



- The channel (once encrypted) is associated with knowledge (enc key, password etc)
- The knowledge is associated with some known entity called "User"
- The session and/or the thread of execution for anything from that channel is then associated with that "User" entity.

Authentication in a Container

- App Server and Web Server provide Authentication SPIs (Service Provider Interface)
- The SPI is connected to DB, LDAP, or other
- In production, authentication typically happens at the perimeter, where the SSL tunnel terminates.

Authorization Semantics

- Subject
  Who does it: Typically the authenticated thread in synchronous scenarios
- Resource
  What is there to be secured
- Action
  What happens to the resource:
  read, create, change, remove, execute

Common Access Control Models

- RBAC (Role-based Access Control): At least one of the required roles must be held by the accessor (wide-spread use in applications everywhere).
- Multi-Tenant: Resource and accessor must share the same tenant association
- Ownership: accessor can only access “her” resources.
- Clearance: Only resources particularly cleared for access are visible. Typically hierarchical.

JEE Security

- RBAC with hard-coded roles
- @RolesAllowed({“R1”, “R2”})
- @PermitAll
- @DenyAll
- @RunAs(“ADMIN”)

What Resources Are There to be Secured?

- domain objects
- methods
- URLs
- Files
- UI Parts

Security is an Aspect

- Security code should be almost “invisible” for “regular” developers.
- Semantics should be runtime-configurable.
- Apply security aspects during method invocation
- Collect access context free from semantics
- Evaluate context at a central point. Choose semantics there based on configuration

Security with Spring

- AOP via Proxy or bytecode manipulation
- Based on Annotations
- activated by @EnableGlobalMethodSecurity
- Method call context evaluated centrally

Security Tests

```
@Test
public void test_access_if_contract_has_adequate_tenant() {...}

@Test
public void test_access_denied_when_wrong_tenant() {...}

@Test
public void test_access_denied_when_not_authenticated() {...}

@Test
public void test_access_granted_when_same_tenant() {...}

@Test
public void test_post_filter_on_tenant() {...}

@Test
public void test_audit_records() {...}

@Test
public void test_audit_records_with_exceptions() {...}

//@Test
public void test_ownership_restrictions() {...}
```

A UNIT TEST

```
@Test
public void test_access_if_contract_has_adequate_tenant() {

  given_Tenants_Schmitz_and_Schulz();

  given_Users_Wolfie_Schmitz_And_Harry_Schulz();

  given_authenticated("wolfie", "wolfie");

  when_saving_a_contract_for(schmitz, ContractType.CONSULTING);

  contract_should_end_up_in_db();

  the_database_should_show(1, ContractType.CONSULTING);
}
```

Authorising Method Entry

```
@Override
@Audit(classifier = "create")
@PreAuthorize("hasPermission(#newContract, 'createOrUpdate')")
@Transactional
public void save(Contract newContract) {

    entityManager.persist(newContract);
}
```

Filtering Results

```
@PostFilter("hasPermission(filterObject, 'read')")
public List<Contract> findByContractType(ContractType contractType) {
    TypedQuery<Contract> query = entityManager.createQuery(
            "SELECT c FROM Contract c WHERE c.contractType = :contractType", Contract.class);
    query.setParameter("contractType", contractType);
    return query.getResultList();
}
```

Permission Evaluator

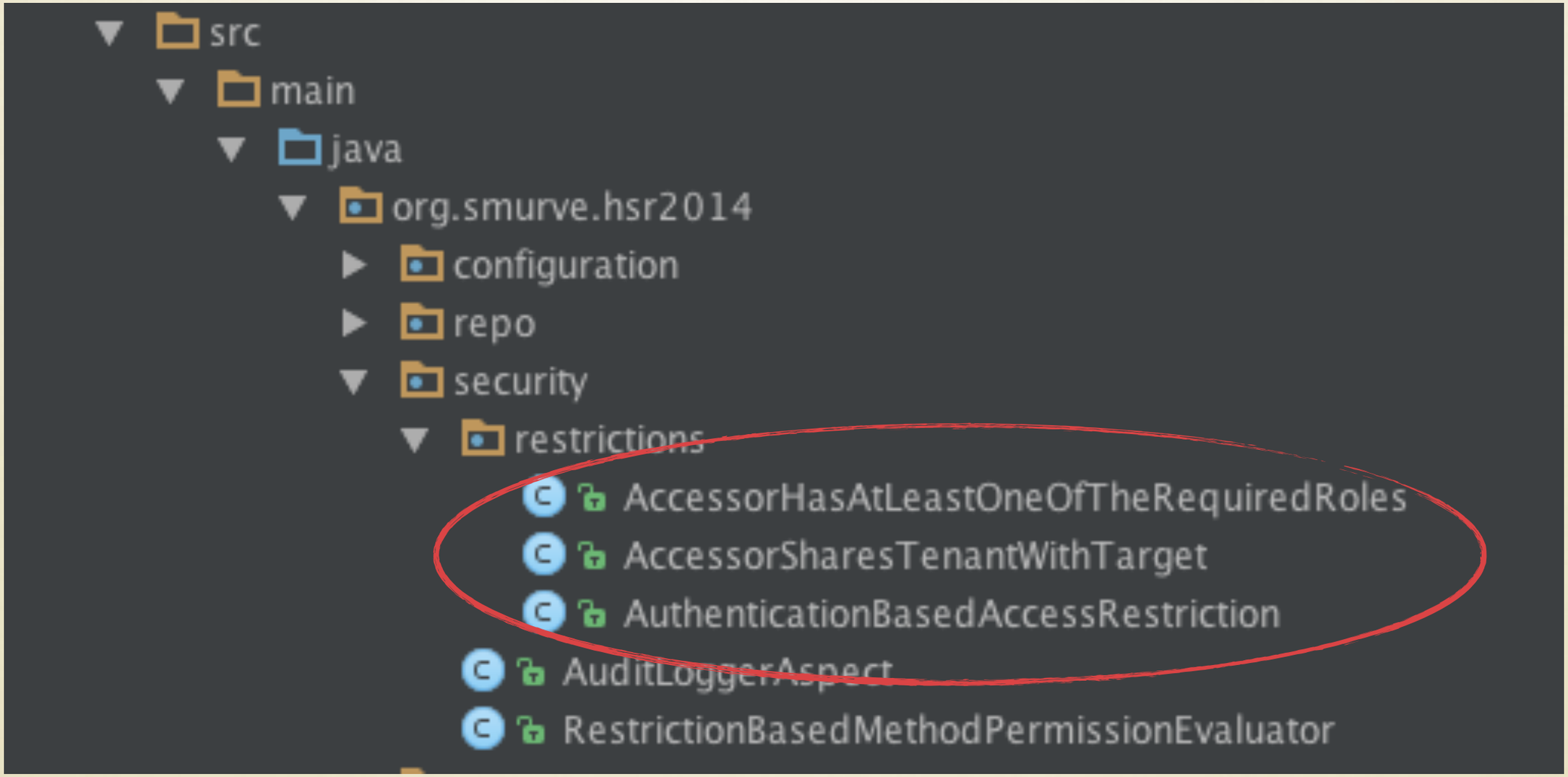
```
/**
 * Evaluates permission for the current Authentication object in the securityContext.
 * <p>
 * Our service methods are annotated with @PreAuthorize and @PostFilter.
 * <p>
 * This PermissionEvaluator delegates to particular restrictions and grants access if
 * no restriction applies
 */
@Service
public class RestrictionBasedMethodPermissionEvaluator implements PermissionEvaluator {
    private static final Logger LOGGER = LoggerFactory.getLogger(RestrictionBasedMethodPe

    private final AccessControlService accessControlService;

    @Autowired
    private List<AccessRestriction> accessCriteria;

    @Autowired
    public RestrictionBasedMethodPermissionEvaluator(AccessControlService acs) {
        accessControlService = acs;
        this.accessCriteria = accessCriteria;
    }
```

Loosely Coupled Restrictions



Enhancement

- Ownership and tenant should be implied from authentication
- Setting tenant and owner may be forgotten or implemented wrong.
- An Aspect could manage setting implied security attributes

Enhancement Aspects

```
/* intercept save method and enrich the entity, if it is a SecuredObject
   Set tenant and owner from looking at the authentication info of the current Thread
 */
@Around("execution(* org.smurve.hsr2014.repo.*.save(..))")
public Object setTenantAndOwner(final ProceedingJoinPoint joinPoint) throws Throwable {

    Object[] args = joinPoint.getArgs();

    if ( args.length != 1 )  return joinPoint.proceed();
    Object object = args [0];
    if ( !(object instanceof SecureResource )) return joinPoint.proceed();

    SecureResource resource = (SecureResource) object;
    String userName = findAuthenticatedUser();

    // nothing to enhance here.
    if ( userName == null ) return joinPoint.proceed();

    User user = repo.findByUsername(userName);

    resource.setOwner(userName);
    resource.setTenant(user.getTenant());
```

Activating an Aspect

```
public class SpringSecurityContext {

    @Bean
    public AuditLoggerAspect auditLoggerAspect() {
        return new AuditLoggerAspect();
    }

    @Bean
    public SecurityEnhancementAspect securityEnhancementAspect() {
        return new SecurityEnhancementAspect();
    }
```

Caveat 1: The Knowledge Problem

- Example: activateCustomerWithID(String id)
- Need to access the customer to find out whether access is allowed

Caveat 2: The ORM Problem

- ORM persists entire entity graphs
- To fully prevent illegal manipulation, the entire graph must be traversed and searched for ownership or tenant information.
- Suggestion: Disallow cascading updates

Architectural Implications

- Method invocations should be secured where meaningful and possible (service layer)
- Problem: “Need access to know whether you need access” must be addressed
- Method signatures must be designed to allow for method security.

Audit logs

- (see code in the exercises)
- Based on Annotations
- @Around(“execution(@…Audit * org.smurve.hsr2014.repo.*.*(..))")
- @Audit(classifier = "create")

Implementing Aspects

```
@Aspect
@Order(AspectOrder.OUTERMOST)
public class AuditLoggerAspect {
    private static final Logger LOGGER = LoggerFactory.getLogger(AuditLoggerAspect.class);

    @Autowired
    private AuditRecordRepository repo;

    // intercept any method in the repo package that has the Audit Aspect
    @Around("execution(@org.smurve.hsr2014.security.Audit * org.smurve.hsr2014.repo.*.*(..))")
    public Object createAuditLog(final ProceedingJoinPoint joinpoint) throws Throwable {

        Object[] args = joinpoint.getArgs();
```

Summary

- Use aspect-oriented programming for security implementation
- deny access or filter results
- Semantic-free context collection
- Room for arbitrary new restrictions
- new restrictions will be found automatically
- Not necessary to use Springframework for this

Alternative Approaches

- JDBC Layer Access Control
- SQL Enhancement (commercially av'le in CH)
- DB Security where possible

Wiederholungsfragen

- Welche Zugriffskontrollmodelle kennen Sie?
- In welcher Form sollte Sicherheit implementiert werden?
- Was sind die wichtigsten Qualitätsattribute im Zusammenhang mit Security?
- Welches sind “schützenswerte” Objekte?
- Wo genau (im Code) ist Zugriffskontrolle am sinnvollsten?

Exercise

- Implement owner-based restriction
- Establish security on service layer
- Write an Aspect that prevents a SecuredResource from being returned to the calling method.
- Make the resp. test methods succeed:
  - MultiTenantSecurityTest
  - ContractServiceSecurityTests

Working with Aspects

```
@Override
@Transactional
public void save(Contract newContract) {

    entityManager.persist(newContract);
}
```

```
"main"@1 in group "main": RUNNING
save():28, UnsecuredJpaContractRepository (org.smurve.hsr2014.repo)
invoke0():-1, NativeMethodAccessorImpl (sun.reflect)
invoke():57, NativeMethodAccessorImpl (sun.reflect)
invoke():43, DelegatingMethodAccessorImpl (sun.reflect)
invoke():606, Method (java.lang.reflect)
invokeJoinpointUsingReflection():317, AopUtils (org.springframework.aop.support)
invokeJoinpoint():190, ReflectiveMethodInvocation (org.springframework.aop.framework)
proceed():157, ReflectiveMethodInvocation (org.springframework.aop.framework)
proceedWithInvocation():98, TransactionInterceptor$1 (org.springframework.transaction.interceptor)
invokeWithinTransaction():262, TransactionAspectSupport (org.springframework.transaction.interceptor)
invoke():95, TransactionInterceptor (org.springframework.transaction.interceptor)
proceed():179, ReflectiveMethodInvocation (org.springframework.aop.framework)
proceed():85, MethodInvocationProceedingJoinPoint (org.springframework.aop.aspectj)
setTenantAndOwner():60, SecurityEnhancementAspect (org.smurve.hsr2014.security)
invoke0():-1, NativeMethodAccessorImpl (sun.reflect)
```

Special Exercise

- If a contract had a contractValue, only users with role “SalesRep” should see the value, but everyone can see the contract.