12.–15.09.2010
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Alles Nichts, oder?!

JPA Security in Theorie und Praxis

Stefan Hildebrandt

consutling.hildebrandt.tk

Problemstellung

- Natürliche Berechtigungen
  - Nutzer dürfen eigene Daten ändern
- Individuelle Berechtigungen
  - Benutzergruppe XY darf Privatkunden bearbeiten, Z Geschäftskunden, Benutzer Admin beides
- Mandantenfähigkeit
  - Das System soll für unterschiedliche Mandanten getrennte Datenbestände verwalten
- Customer Selfservice
  - Für Firmenkunden in hierarischer Form
- ACLs
  - Veränderbare Definition von Rechten an einzelnen Objekten

Lösung 1: Handarbeit Code

- Ausfiltern der „falschen“ Daten im Code
  - Pro
    - Ohne weitere Maßnahmen für jeden Entwickler möglich
    - Keine weiteren Bibliotheken
  - Contra
    - Sehr fehleranfällig
    - Sehr hoher Pflegeaufwand
    - Es werden alle Daten in den Speicher geladen
    - Ineffizient

Lösung 2: Handarbeit DB

- Anpassung aller Queries per Hand
  - select u from User u
  - Regel: <Object>.clientId=:clientId
  - select u from User u where u.clientId=:clientId
  - Select a from Address a join User u on u.id=a.userId where u.id=:userId AND a.streetName=:streetName AND a.streetNumber=:steetNumber AND a.zip=:zip AND a.city=:city AND u.clientId=:clientId AND p.clientId=:clientId

Lösung 2: Handarbeit DB

- Anpassung aller Queries per Hand
  - Regel: <Object>.aclId=acl.id AND acl.groupId=group.id AND group.id=userGroup.groupId AND userGroup.userId=:userId
  - Select a from Address a join User u on u.id=a.userId join Acl acl1 ON a.aclId=acl1.id JOIN group g1 ON acl1.groupId=g1.id JOIN usergroup ug1 ON g1.id=ug1.groupId [...]where u.id=:userId And a.streetName=:streetName AND a.streetNumber=:steetNumber AND a.zip=:zip AND a.city=:city AND ug1.userId=:userId AND ug2.userId=:userId

Lösung 2: Handarbeit DB

- Pro
  - Keine weiteren Voraussetzungen
  - Performant
- Contra
  - Fehleranfällig
  - Vermischung fachlicher und technischer Aspekte
  - Hoher Pflegeaufwand
  - Nur Prüfung auf Read-Berechtigung
  - Navigation über Relationen hinweg
    - Eventuell weitere Prüfung notwendig

Lösung 3: Hibernate Filter

- Anpassung aller Queries mittels hinterlegten Regeln

```
<filter-def name="clientFilter">
    <filter-param name="clientId" type="int"/>
</filter-def>

<class name="User" ...>
    ...
    <filter name="clientFilter" condition=":clientId = clientId"/>
</class>

session.enableFilter("clientFilter").setParameter("clientId", 25);
```

Lösung 3: Hibernate Filter

- Pro
  - Bei Verwendung von Hibernate einfach anwendbar
  - Automatische Anpassung aller Queries
  - Performant
- Contra
  - Kann nur mit Hibernate verwendet werden
  - Aktivierung des Filters muss eingebaut werden
  - Filterung nur direkt auf Spalten der aktuellen Tabelle
    - Erweiterung nur über subselect oder stored procedures
  - Ist in SQL formuliert
  - Nur Prüfung auf Read-Berechtigung
  - Speichern von Relation mit ausgefilterten Objekten kann zu inkonsistenten Zuständen führen

Lösung 4: Spring Domain Object Security

- Teil von Spring Security
- Prüfservice zur Überprüfung von ACLs an Entitäten
- Mitlieferung von Manipulationsservices
  - Definition von Rechten an Entitäten
  - ACL und Benutzerverwaltung
- Konfiguration über Spring
- Prüfungen müssen an geeigneten Stellen plaziert werden
- Links
  - http://static.springsource.org/spring-security/site/docs/3.0.x/reference/springsecurity-single.html#domain-acls
  - http://www.denksoft.com/wordpress/?page_id=20

Lösung 4: Spring Security

- Pro
  - Ausdruckstarkes ACL-Modell
    - Baumstrukturen möglich
    - Bis zu 32 unterschiedliche Berechtigungen
  - Datenbank-Layer zur Verwaltung von ACLs
- Contra
  - Nur Prüfung, Filterung muss an anderer Stelle erfolgen
  - Prüfung muss an geeigneter Stelle eingebaut werden
  - Ressourcenbedarf
    - Daten werden parallel aus der Datenbank geladen
    - Alle Daten werden in den Speicher geladen, danach aussortiert
  - Bei OR-Mapper: Lazy-Loading ist kaum zu Prüfen

Lösung 5: JpaSecurity

- Wird mit dem JPA-Provider verbunden
- Änderung der Anfragen vor Ausführung
- Deklarativer Ansatz
- Definitionsmöglichkeit von Regeln für jede Entität
- Regeln in JPQL naher Sprache verfasst
- Mehrere Regeln werden mit „oder“ verknüpft
- Prüfung analog zum Principal (User, Roles)
  - Mittels Provider kann dieser aus vielen Quellen bezogen werden
  - Die Felder lassen sich auch anders belegen

JpaSecurity: Arbeitsweise

- TODO: bild

JpaSecurity: Arbeitsweise

- Proxy um EntityManager, …
- Rückgabe von Kopien der Entitäten
- Proxies um Collections in Entitäts-Kopien
  - Filterung beim Zugriff auf Basis der Regeln
  - In-Memory Überprüfung
- Lese-Berechtigung durch Erweiterung der Query
- Update- und Delete-Berechtigungen vor commit
  - In-Memory-Überprüfung
- Ausführung der Lifecycle-Methoden auf den Kopien
- Abweichungen bei persist
  - Rückgabe eines Proxies laut JPA nicht möglich
    - Nur Pre-Commit-Prüfungen

JpaSecurity: Einbindung

- persistence.xml
  - Austausch des Persistence-Providers
  - Angabe folgender Parameter:
    - Originaler PersistenceProvider
    - AuthentictionProvider
    - AccessRulesProvider
- Bereitstellung der Zugriffsregeln

JpaSecurity: Einbindung

```
<persistence-unit name="contacts" transaction-type="Ressource_LOCAL">

  <provider>net.sf.jpasecurity.persistence.SecurePersistenceProvider</provider>

  <class>net.sf.jpasecurity.contacts.model.User</class>
  <class>net.sf.jpasecurity.contacts.model.Contact</class>

  <properties>
    <property name="net.sf.jpasecurity.persistence.provider"
                  value="org.hibernate.ejb.HibernatePersistence"/>
    <property name="net.sf.jpasecurity.security.authentication.provider"
                  value="net.sf.jpasecurity.security.authentication.StaticAuthenticationProvider"/>
      ...
  </properties>

</persistence-unit>
```

JpaSecurity: Beispiel

- Query
  - select u from User u
- Security.xml

```
<persistence-unit name="jpasecuritySample">
   <access-rule>
     GRANT ACCESS TO User u WHERE u.id = CURRENT_PRINCIPAL
   </access-rule>
   <access-rule>
     GRANT ACCESS TO User u WHERE u.creator.id = CURRENT_PRINCIPAL
   </access-rule>
   <access-rule>
     GRANT ACCESS TO User u WHERE 'admin' IN(CURRENT_ROLES)
   </access-rule>
 </persistence-unit>
```

JpaSecurity: Beispiel

AuthenticationProvider.authenticate(-1L, "admin");
→ SELECT u FROM User u

AuthenticationProvider.authenticate(creatorId);
→ SELECT u FROM User u WHERE ((u.id = :user) OR (u.creator.id = :user))

AuthenticationProvider.authenticate(creatorId+1);
→ SELECT u FROM User u WHERE ((u.id = :user) OR (u.creator.id = :user))

- Optimierung der Query anhand der Paramter

JpaSecurity: Speicherbedarf

- ca. 50%-100% Aufschlag im Bereich der Entitäten
  - Original
  - Kopie JpaSecurity
  - Kopie PersistenceProvider
    - Teilweise unvollständig
- Anteil des Speicherbedarfs für Entitäten je nach Anwendung sehr unterschiedlich

JpaSecurity: Rechenzeit

- Anpassung der Query
  - Bei komplexen Statements schneller (30ms) als Hibernate (300ms)
    - Statements ohne Security einfacher
- Erstellung der Proxies und kopieren der Werte
  - Queries mit vielen Ergebnissen
  - Für geringe Anzahl Objekte kaum messbar
  - Bei sehr vielen Entitäten (>100.000) bis zu 200% Aufschlag
- Lifecycle-Prüfungen und In-Memory-Prüfungen
  - Massendatenimporte mit sehr vielen Entitäten (>10.000) bis zu 1000% Aufschlag

JpaSecurity: Entwicklungsstand

- Produktionsreife
  - Version 0.3.0 wird released (TODO)
  - Wird in zwei größeren Projekten eingesetzt
  - Läuft in mir bekannten Projekt mittlerweile unauffällig
  - Erweiterungen konnten kurzfristig umgesetzt werden
  - Offene Probleme:
    - Verdecken von Fehlern in der Query durch NPEs im Parser
    - Ressourcenbedarf in einigen Anwendungfällen enorm
      - Ursache für die vielen Kopien liegt meist wo anders
      - Maßnahmen zur Senkung des Aufwands sind geplant

Lösung 5: JpaSecurity

- Pro
  - Einbindung über JPA an sehr zentraler Stelle
  - Einfache Regeldefinition
  - Ganzheitliche Behandlung des Data-Access-Layers
  - Definition getrennter Read /Delete-/Update-Regeln

Lösung 5: JpaSecurity

- Contra
  - Erhöhter Ressourcenbedarf bei sehr vielen Entitäten
  - Nur JPA 1.0
    - NativeSQL Queries werden nicht gesichert
    - Weiterentwicklung für JPA 2.0 hat erst begonnen
  - Join on
    - Regeln müssten bei outer joins in den Join generiert werden
      - Wird von JPA nicht unterstützt
      - Daher auch nicht von JpaSecurity
  - Fehlende Serialisierbarkeit von Entitäten
    - Der Interceptor an den Entitäten ist derzeit nicht serialisierbar

Statement-Komplexität

- Statements mit vielen Regeln oder Parametern werden sehr komplex
  - Ausführungszeiten steigen teils drastisch
  - Parsen der Query durch JPA-Provider und die Datenbank
    - Hibernate teilweise mehr als 2 Sekunden
    - Oracle
  - Grenzen in der Datenbank
    - Ausführungszeit
      - 64k Statements in Oracle
      - Beschränkung der Elemente in einer IN(...)-Liste

Regel-Komplexität

- Komplizierte Regeln mit mehreren Joins
- Beispiel
- Extreme Erhöhung des Aufwandes zur Prüfung
  → „Compilieren“ von fachlichen Regeln

Achtung ACLs

- Explizite Angabe von Berechtigungen
  - Keine Prüfung anhand natürlicher Regeln
- Erstellung von ACLs anhand von Regeln
  - Auf Basis natürlicher Regeln
  - Auf Grund von fachlichen Zusammenhängen
  - Ableitung aus Geschäftsprozessen

  → Neue Komplexität

  → Einsatz einer Regelengine
- ACLs an Daten ziehen ACLs beim Starten von Aktionen nach sich.
  - z.B. als Kombination von Rechten der zu verändernden Daten

Vielen Dank!

Stefan Hildebrandt

consulting.hildebrandt.tk

consulting.hildebrandt.tk

- Entwicklung und Entwicklungsunterstützung im Bereich JEE-Anwendungen
- Coaching (im Projekt) und Schulung
  - Frameworks und Tools
  - Pairprogramming
  - Test-Driven-Development
- Softwarearchitektur und Projektsetup
  - Buildprozess, CI
  - Frameworkauswahl und -integration
  - Automatisiertes Deployment
  - Betriebssetup Jboss, Tomcat, Jetty
- Leichtgewichtige Entwicklungsprozesse