

## syngenio in der Fachpresse

Medium: Javamagazin  
Thema: **Einfache Lösungen zur Implementierung von Single Sign-on (SSO)**  
Autor: Oliver Rummeyer und Jörg Düsterhaus  
Ausgabe: 10.2006

### SSO frei Haus

Sicher ist sicher. Deshalb hat inzwischen fast jede Webanwendung einen geschützten Bereich. Ob das für jede Anwendung im Einzelnen Sinn macht, sei einmal dahingestellt. Tatsache ist jedoch, dass dies zu einer wahren Flut von Zugangsdaten führt. Das bemerkt man meist erst, wenn man länger über die Benutzererkennung nachdenken muss als über das Passwort. Historisch gewachsene Webportale bestehen zudem nicht mehr nur aus einer einzigen Anwendung. Spätestens dann, wenn diese alle ihren eigenen Authentifizierungs-Mechanismus mitbringen, muss etwas getan werden, um den Benutzern das Leben zu erleichtern. Und das geht auch ohne teure Produkte und mit überschaubarem Aufwand.



### Kontakt & weitere Informationen:

syngenio AG  
Ivonne Machnacz  
Andreas-Hermes-Straße 3  
D-53175 Bonn  
Fon +49 (0)2 28-6 20 95-121  
Fax +49 (0)2 28-6 20 95-150  
ivonne.machnacz@syngenio.de  
www.syngenio.de

## Einfache Lösungen zur Implementierung von Single Sign-on (SSO)

# SSO frei Haus

■ VON OLIVER RUMMEYER UND JÖRG DÜSTERHAUS



Sicher ist sicher. Deshalb hat inzwischen fast jede Webanwendung einen geschützten Bereich. Ob das für jede Anwendung im Einzelnen Sinn macht, sei einmal dahingestellt. Tatsache ist jedoch, dass dies zu einer wahren Flut von Zugangsdaten führt. Das bemerkt man meist erst, wenn man länger über die Benutzererkennung nachdenken muss als über das Passwort. Historisch gewachsene Webportale bestehen zudem nicht mehr nur aus einer einzigen Anwendung. Spätestens dann, wenn diese alle ihren eigenen Authentifizierungs-Mechanismus mitbringen, muss etwas getan werden, um den Benutzern das Leben zu erleichtern. Und das geht auch ohne teure Produkte und mit überschaubarem Aufwand.

Um verschiedene Anwendungen in einem einheitlichen Bild erscheinen zu lassen, gilt es, Mehrfachanmeldung zu vermeiden. Aus Gründen der Benutzerfreundlichkeit wird hier also ein Single-Sign-on-(SSO-) Mechanismus angestrebt (also ein Mechanismus zur Einmalanmeldung), der es dem Benutzer erlaubt, sich mit einer einzigen Authentifizierung an einem gesamten Anwendungsverbund anzumelden. Der Benutzer muss sich somit nur noch eine Kennung und ein Passwort merken.

Das hierdurch entstehende Sicherheitsrisiko – einem Angreifer genügt das Ausspähen einer einzigen Benutzeridentität, um Zugriff auf den gesamten Verbund

zu bekommen – wird in den meisten Fällen in Kauf genommen, denn der Vorteil von SSO wiegt dies meist auf. Nach der Einmalanmeldung werden alle weiteren Aufrufe automatisch und zeitsparend mit den notwendigen Authentifizierungsdaten – den Credentials – versorgt.

## Vor- und Nachteile

Welche Vorteile bietet SSO denn nun und was sind die Nachteile? Zu den wichtigsten Vorteilen, sowohl für Endanwender als auch für Unternehmen, die eine SSO-basierte Infrastruktur einsetzen, zählen hierbei:

- einheitlicher Zugriff über einen einheitlichen Authentifizierungsmechanismus.
- Benutzer müssen sich nur genau eine Kennung und ein Passwort merken.

- Benutzer können ein komplexeres Passwort wählen, was die Sicherheit erhöht.
- im Idealfall nur eine manuelle Anmeldung pro Sitzung.
- Kostenreduktion durch bessere Wartbarkeit von Zugangs- und Benutzerdaten.

Obwohl die Vorteile in den meisten Fällen die Nachteile überwiegen, sollen diese nicht verschwiegen werden. Als besonders kritisch einzustufen sind folgende Aspekte:

- Angreifer können durch Ausspähen der Zugangsdaten auf alle Systeme zugreifen.
- Ein zentraler Anmeldevorgang kann Single Point of Failure sein, sowohl in Bezug auf Sicherheitsprobleme als auch hinsichtlich des Lastverhaltens.

**Quellcode auf CD**

Zudem kann es aufgrund von Datenschutzgesetzen ein Problem darstellen, wenn personenbezogene Daten zentral gespeichert werden. Ein über Unternehmensgrenzen hinweg arbeitendes Single Sign-on zu realisieren ist deshalb schwierig.

### Lösungsansätze

Es gibt verschiedene Methoden, einen SSO zu realisieren (mal ganz abgesehen von einer clientbasierten Lösung wie etwa einem Passwort-Manager). Im Rahmen dieses Artikels werden jedoch nur zwei der wichtigsten Lösungsansätze beschrieben: Zum einen die Verwendung einer zentralen Instanz, welche die Anmeldung vornimmt, zum anderen die Realisierung eines Circle of Trust, bei der die Anmeldung am Gesamtsystem durch Anmeldung an einem beliebigen System innerhalb des Verbunds erfolgt.

Eingrundsätzlicher Ansatz bei SSO, der vorab kurz erläutert werden muss, ist der Einsatz von Tickets. Diese Tickets werden bei einer erfolgreichen Anmeldung erzeugt und dienen ab diesem Zeitpunkt dazu, den Anwender zu authentifizieren. Das Ticket, das ein Benutzer erhält, wird idealerweise automatisch verifiziert (anstelle einer interaktiven Anmeldung durch Eingabe von Benutzerkennung und Passwort). Klar ist jedoch, dass ein solches Ticket – wird es denn von einem Angreifer abgefangen – ein potenzielles Risiko darstellt. (Dieses ist jedoch keinesfalls so schlimm wie ein erspähtes Passwort!) Tickets sollten deshalb mit einer Gültigkeit versehen werden. Falls ein Ticket entwendet und unrechtmäßig eingesetzt wird, ist es wahrscheinlich schon abgelaufen. Es wird dann wieder eine interaktive Anmeldung nötig, und der Angreifer hat seine Chance vertan. Um Veränderungen an einem Ticket zu verhindern (zum Beispiel um besagte Gültigkeit zu manipulieren), sollte dieses zudem möglichst verschlüsselt oder wenigstens signiert werden. Benutzerkennung und Passwort sind natürlich auf keinen Fall in ein solches Ticket zu speichern. Nein, und auch nicht verschlüsselt.

### Circle of Trust

Ein möglicher Lösungsansatz für SSO ist der so genannte Circle of Trust. Hierbei

wird ein Netz aus vertrauenswürdigen Anwendungen aufgebaut. Meldet sich ein Benutzer bei einer der Anwendungen an, so ist er danach für alle anderen Anwendungen gleichermaßen angemeldet. Die ideale Lösung sieht dabei so aus, dass der Benutzer bei der Anmeldung am ersten System ein Ticket bekommt, mit dem er sich bei allen anderen Anwendungen authentifizieren kann. Das Ticket muss hierzu bei jeder Anwendung aus dem Circle of Trust verifiziert werden können.

In der echten Welt (ich hätte doch die blaue Pille nehmen sollen) steht man jedoch ziemlich oft vor dem Problem, dass man alte Anwendungen hat, die man nur minimal verändern möchte oder kann. Oft muss auch mit kleinem Budget eine Lösung realisiert werden, die SSO umsetzt. Bei obigem Ansatz müsste man aber jede Anmeldeprozedur der Anwendungen durch einen neuen Mechanismus ersetzen, der dieses eine (allen Anwendungen gemeinsame) Ticket interpretieren kann.

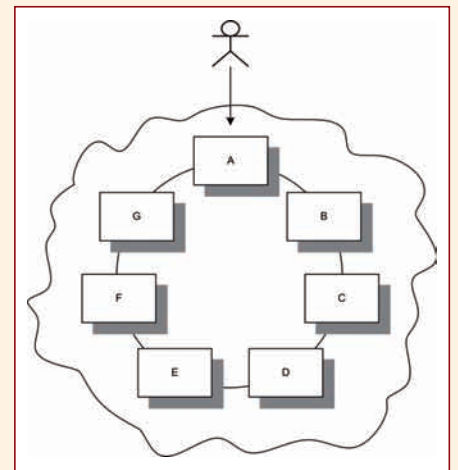


Abb. 1: Circle of Trust

Eine einfachere Möglichkeit, einen Circle of Trust für Java-basierte Webanwendungen zu realisieren, ist es, mit versteckter Anmeldung (Hidden Logon) zu arbeiten. Hierzu wird zwischen der Anmeldung und dem eigentlichen Anwendungsfall ein Schritt in den Ablauf integriert, welcher Anmeldungen an allen

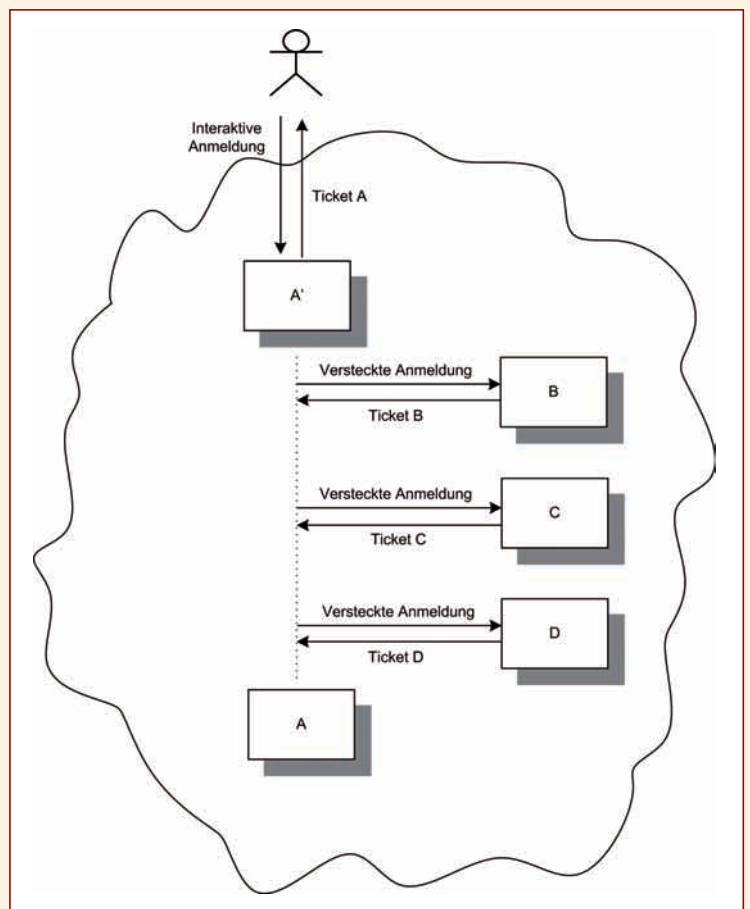


Abb. 2: Versteckte Anmeldung

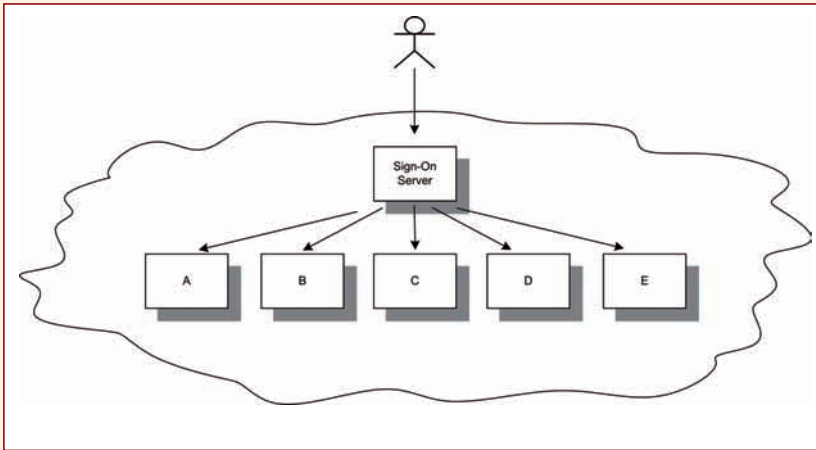


Abb. 3: Sign-on-Server

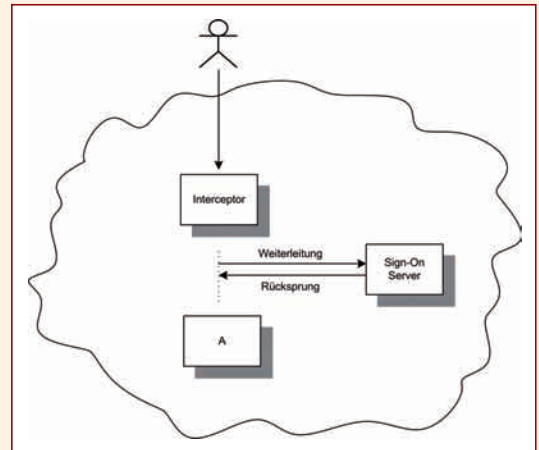


Abb. 4: Interceptor

beteiligten Anwendungen des Circle of Trust durchführt (mittels eines HTTP-Aufrufs) – quasi ein Login anstelle des Benutzers. Dieser neue Schritt bringt den Zustand der Sitzungen aller Anwendungen auf „angemeldet“.

Falls die beteiligten Anwendungen alle mit Cookies arbeiten, ist man danach im Idealfall schon fertig. Jedes Cookie im Browser identifiziert den Benutzer gegenüber der Anwendung, die er „ansurft“, und der Benutzer erhält dadurch Zugriff. Er muss sich nicht interaktiv anmelden, denn er wurde ja versteckt automatisch angemeldet, und die Anwendung erkennt den Benutzer durch das Cookie, das er mit sich bringt.

Hinlänglich bekannt sein sollten allerdings die Risiken beim Einsatz von Cookies. Falls sich mehrere Benutzer einen Rechner teilen, muss sichergestellt werden, dass bei einem Benutzerwechsel keine gültigen Cookies mehr auf dem Rechner vorhanden sind. Zudem existiert das Problem von möglichen Angriffen durch Cross-Site-Scripting-Attacks.

Die Umsetzung dieses einfachen Circle of Trust wird schwieriger, wenn die Identifikation der Anwendungssitzungen mittels SessionId-Parameter in dem URL realisiert ist. Dann muss jeder Link auf eine der Anwendungen zusätzlich den richtigen Wert für den URL-Parameter gesetzt bekommen. Man kann dies beispielsweise über ein JSP Custom Tag realisieren, welches – abhängig von dem Anwendungs-URL – den richtigen Parameter anhängt. Allerdings muss trotzdem noch das Problem gelöst werden, wo man

sich die ganzen Werte für die SessionId-Parameter merkt. Zudem muss man noch jeden Link auf allen JSP-Seiten dahingehend anpassen, dass das neue JSP Custom Tag verwendet wird. Das klingt nach viel Arbeit und Chaos. Je nach Beschaffenheit der Anwendungen, ist es das auch.

Es gibt auch noch eine Reihe weiterer möglicher Probleme bei dieser Lösung: Die beteiligten Anwendungen können beispielsweise in Time-outs laufen. Das kann zwar meist durch Erhöhen des Session-Time-outs verhindert werden, ist dies jedoch im Einzelfall nicht möglich, muss ein „Keep alive“-Mechanismus realisiert werden. Des Weiteren muss eine Abmeldung vom System natürlich für alle beteiligten Anwendungen durchgeführt werden und Sonderfälle, wie eine erzwungene Änderung des Passworts bei der ersten Anmeldung, müssen ebenso berücksichtigt werden.

Eine naive Umsetzung der versteckten Anmeldung impliziert zudem, dass alle Anwendungen die gleichen Zugangsdaten verwenden. Möchte man nicht für alle Anwendungen eine Angleichung der Benutzerdaten (gleiche Benutzerkennung und gleiches Passwort) durchführen, muss man sich an dieser Stelle auch noch etwas überlegen, quasi eine Zuordnung zwischen den neuen, „einigen“ Benutzerdaten und den pro Alt-Anwendung existierenden Benutzerdaten.

Je mehr man über diese vermeintlich einfache Lösung nachdenkt, desto mehr wünscht man sich eine sauberere Vorgehensweise, bei der die Anmeldeprozeduren und die Sitzungsverwaltung

entsprechend angepasst sind. Bei einer Cookie-basierten Sitzungsverwaltung kommt man mit der Methode der versteckten Anmeldung jedoch oft relativ schnell zum Ziel. Allerdings nur, falls die angedeuteten Probleme vernachlässigt werden können.

### Zentraler Sign-on-Server

Ein anderer Lösungsansatz ist, für die Anmeldeprozedur eine zentrale Instanz zu etablieren. Bei dieser Lösung – mit einem dedizierten Sign-on- bzw. Ticket-Server – wird eine eigene Anwendung implementiert, die zu Beginn jeder Sitzung angesprochen werden muss. Der Client erhält über diesen Server das Ticket, mit dem er sich gegenüber allen Anwendungen im Verbund authentifizieren kann.

Das Ticket muss vom Client bei jeder Anfrage an eine Anwendung des Verbunds mitgegeben werden. Jede einzelne Anwendung enthält dann eine spezielle Logik, die das Ticket verifiziert bzw. beim zentralen Ticket-Server verifizieren lässt. Falls das Ticket erfolgreich verifiziert werden konnte, erhält der Benutzer Zugriff auf die Anwendung.

Alte Anwendungen müssen entsprechend dieser Logik angepasst werden. Die alte Anmeldeprozedur muss durch einen neuen Ablauf ersetzt werden. Anstatt über die ursprüngliche Anmeldeseite muss über den Sign-on-Server eingestiegen werden, welcher dann an die Anwendung weiterleitet (oder umgekehrt). Zusammengefasst enthält das Vorgehen für eine erfolgreiche Anmeldung am Anwendungsverbund folgende Schritte:

- Jeder Aufruf eines Benutzer-Clients wird zuerst an den Sign-on-Server gerichtet (oder dahin umgelenkt).
- Nach der interaktiven Anmeldung erhält der Client vom Sign-on-Server ein persönliches Ticket.
- Die Anfrage des Clients wird vom Sign-on-Server an die eigentliche Anwendung weitergeleitet (mittels einer mitgegebenen Rücksprunginformation z.B. in Form eines Rücksprungs-URL).
- Der Client überträgt das Ticket an die Anwendung.
- Die Anwendung verifiziert das Ticket mithilfe eines Aufrufs an den zentralen Sign-on-Server.
- Bei gültigem Ticket wird dem Benutzer Zugriff auf die eigentliche Anwendung gewährt.

Dieses Vorgehen hat den Vorteil, dass die Client-Anwendung niemals das Passwort des Benutzers sehen muss. Es wird lediglich an den zentralen Sign-on-Server übermittelt, der daraufhin ein Ticket generiert und dieses an die Client-Anwendung zurückgibt.

Ein interessanter Aspekt im Ablauf des oben beschriebenen Anmeldevorgehens ist die Weiterleitung vom Sign-on-Server zur Anwendung oder – als Alternative – von der Anwendung zum Sign-on-Server. Man hat einerseits die Möglichkeit zu vereinbaren, dass der Einstieg in alle Anwendungen des Verbunds immer über den zentralen Server erfolgen muss. Dieser muss dann entscheiden, auf welche Zielanwendung der Aufruf nach der erfolgreichen Anmeldung weitergeleitet werden soll. Dies kann beispielsweise anhand eines Parameters erfolgen, welcher in einem URL für die Weiterleitung umgesetzt wird.

Die andere Möglichkeit ist die, direkt den URL der Ziel-Anwendung aufzurufen. Dieser überprüft, ob der Benutzer bereits angemeldet ist (das ist er, falls er ein gültiges Ticket mitbringt) und leitet den Aufrufer an die eigentliche Anwendung weiter. Ist der Benutzer noch nicht angemeldet, so wird auf den Sign-on-Server (und nur dann!) umgeleitet, wo eine interaktive Anmeldung gemacht werden muss. Damit der Sign-on-Server weiß, wohin die Reise nach der Anmeldung wei-

tergeht, muss die Applikation auch hier einen URL zu Weiterleitung mitgeben. Dies liegt jedoch in der Verantwortung der einzelnen Anwendungen. Der Vorteil ist, dass man die Einstiege in die Einzelanwendungen nicht anpassen muss. Falls man einen Interceptor-Mechanismus verwendet (beispielsweise mittels AOP oder einfach in Form eines Servlet-Filters), ist der Aufruf der Anmeldung zudem weitgehend transparent.

Sieht man von dem etwas höheren Implementierungsaufwand ab, hat man hier jedoch eine saubere Lösung aus einem Guss. Weitere Anwendungen lassen sich bei dieser Lösung transparent in den Anwendungsverbund eingliedern. Der Aufwand ist ja eigentlich auch gar nicht so groß, denn die meisten Anfragen werden einfach an den zentralen Server delegiert. Man muss jedoch die Sitzungsverwaltung komplett neu implementieren und die Benutzerdaten auch irgendwo speichern. Aber muss man deswegen jetzt das Rad neu erfinden? Stattdessen kann man auch eine bereits existierende Implementierung dieser Art verwenden, wie beispielsweise den frei verfügbaren Central Authentication Service (CAS).

### Central Authentication Service

Ursprünglich von der Yale University entwickelt, ist Central Authentication Service (CAS) heute ein Open-Source-Projekt der Java Architectures Special Interest Group (JA-SIG). Es ist die Implementierung einer zentralen Instanz zur Behandlung von Authentifizierungen.

CAS ist als eigenständige Webapplikation entworfen und als Java Servlet implementiert. Der Zugriff erfolgt im Wesentlichen über zwei URLs: den Login-URL und den Validierungs-URL. Die Architektur entspricht prinzipiell der oben beschriebenen Vorgehensweise für einen zentralen Sign-on-Server bzw. Ticket-Server. CAS basiert dabei auf diversen

Open-Source-Technologien, unter anderem auch auf dem Spring Framework.

### Der Server

Der CAS-Server lässt sich im Auslieferungszustand einfach in den Tomcat-Server deployen, indem die Datei *target/cas.war* in das Verzeichnis *\$CATALINA\_HOME/webapps* kopiert wird. Er ist

**CAS ist die Implementierung einer zentralen Instanz zur Behandlung von Authentifizierungen.**

dann zwar sofort lauffähig und kann getestet werden, in den meisten Fällen wird man jedoch Anpassungen der Konfiguration vornehmen wollen, um beispielsweise seinen eigenen Authentifizierungsmechanismus zu realisieren.

CAS bietet eine flexible Plug-in-Architektur für so genannte AuthenticationHandler. In dem hier vorgestellten Beispiel wurde eine entsprechende Klasse geschrieben, welche das Interface AuthenticationHandler implementiert. Diese Klasse prüft die Eingabedaten gegen eine statische HashMap (Listing 1).

Die beiden implementierten Methoden werden von CAS nach dem Hollywood-Prinzip aufgerufen. Zuerst wird die

## Anzeige

*supports*-Methode aufgerufen. Hier wird zunächst überprüft, ob der angesprochene Handler in der Lage ist, die übergebenen Authentifizierungsdaten (Credentials) formal korrekt zu behandeln. Der *HashMapAuthHandler* ist für die Behandlung von Benutzerkennungs- und Passwortdaten zuständig, also für Eingabedaten des Typs *UsernamePasswordCredentials*. Diese Klasse ist Bestandteil von CAS und repräsentiert einen Container für Anmeldeinformationen, die aus dem Benutzernamen und dem zugehörigen Passwort bestehen. Die *supports*-Methode liefert deshalb „true“ zurück.

Die Methode *authenticate* enthält die eigentliche Logik zur Verifizierung der übergebenen Credentials. An diesem Punkt beginnt die eigentliche Implementierung der Authentifizierungslogik. Man könnte hier entsprechende Anfragen an eine Datenbank richten oder sich beliebig andere Vorgehensweisen überlegen, wie

beispielsweise eine LDAP-Anbindung. CAS ist an diesem Punkt einfach mit Technologien wie Java Authentication and Authorization (JAAS) kombinierbar, mit der man auch eine Kerberos-Integration realisieren kann.

Die AuthenticationHandler müssen in der Datei *WEB\_INF/deployerConfigContext.xml* eingetragen und konfiguriert werden. Sie werden bei einem Authentifizierungsvorgang der Reihe nach auf die übergebenen Credentials ausgeführt:

```
<property name="authenticationHandlers">
<list>
...
<bean class="com.syngenio.cas.authentication.handlers.
support.HashMapAuthHandler"/>
</list>
</property>
```

Sind mehrere AuthenticationHandler konfiguriert, so durchläuft der CAS-Server diese nacheinander. Sobald eine *supports*-Methode erfolgreich durchlaufen

wurde (also „true“ zurückliefert), wird dieser Handler zur Authentifizierung herangezogen. Falls auch die *authenticate*-Methode erfolgreich ist, ist der Benutzer angemeldet.

## Client-Integration

CAS bietet Bibliotheken für die schnelle Implementierung eines Java-basierten Clients. Dieser wird als Servlet-Applikation realisiert. Die Bibliotheken enthalten Java-Klassen, welche den clientseitigen Teil des CAS-Protokolls implementieren – also Klassen für die Überprüfung von Tickets, Servlet-Basisklassen sowie Klassen für die Umleitung von HTTP-Aufrufen zum CAS-Server (Filter). Die *jar*-Datei mit den benötigten Klassen muss nach *\$CATALINA\_HOME/common/lib* kopiert werden. Dadurch können alle Webanwendungen auf die CAS-Client-Klassen zugreifen.

Damit die Anwendung SSO-fähig wird, muss sie bei jeder Anfrage (*HttpRequest*) die Gültigkeit des übergebenen Tickets beim CAS-Server prüfen lassen. Die Aufgabe, diese Anfrage an den CAS-Server zu richten, übernimmt der Servlet-Filter mit dem Namen *CASFilter* aus der CAS-Client-Bibliothek. Diese Klasse spielt eine zentrale Rolle für den Ablauf der Anmeldung. Der CAS-Filter wird in

### Listing 1

```
public class HashMapAuthHandler implements
AuthenticationHandler {
private static final HashMap<String, String>
DUMMY_STORE = new HashMap();

static {
// Map laden, damit wir etwas haben, gegen das wir
// Eingaben prüfen können: Form <Benutzer, Passwort>
DUMMY_STORE.put("Harry", "hirsch");
DUMMY_STORE.put("foo", "bar");
DUMMY_STORE.put("Oli", "Rummeyer");
DUMMY_STORE.put("Joe", "dust");
}

public boolean authenticate(Credentials credentials)
throws AuthenticationException {
UsernamePasswordCredentials upCredentials =
(UsernamePasswordCredentials) credentials;

String user = upCredentials.getUsername();
String pw = upCredentials.getPassword();

if (!DUMMY_STORE.containsKey(user))
return false;

String compareMe=DUMMY_STORE.get(user);
return compareMe.equals(pw);
}

public boolean supports(Credentials credentials) {
return credentials instanceof
UsernamePasswordCredentials;
}
}
```

### Listing 2

```
<filter>
<filter-name>CAS Filter</filter-name>
<filter-class>edu.yale.its.tp.cas.client.filter.
CASFilter</filter-class>

<init-param>
<param-name>edu.yale.its.tp.cas.client.filter.
loginUrl</param-name>
<param-value>https://localhost:8443/cas/login
</param-value>
</init-param>
<init-param>
<param-name>edu.yale.its.tp.cas.client.filter.
validateUrl</param-name>
<param-value>https://localhost:8443/cas/
proxyValidate</param-value>
</init-param>
<init-param>
<param-name>edu.yale.its.tp.cas.client.filter.
serverName</param-name>
<param-value>localhost:8080</param-value>
</init-param>
</filter>

<!-- Filter für alle URL's aktivieren -->
<filter-mapping>
<filter-name>CAS Filter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

### Listing 3

```
protected void doGet(HttpServletRequest req,
HttpServletRequest res) throws ServletException,
IOException {
...
// der cas-filter erzeugt eine session, explizit keine
// neue erstellen
HttpSession s = req.getSession(false);

// name des angemeldeten benutzers auslesen
String authenticatedUser =
(String) s.getAttribute("edu.yale.its.tp.cas.client.
filter.user");

// die http-antwort zusammenbauen
...
out.println("<p>Hello "+authenticatedUser+",
you have successfully\
signed on using the Central Authentication Service
</p>");
...
}
```

der *web.xml*-Datei der Client-Anwendung konfiguriert (Listing 2).

Der Filter kann über die oben aufgeführten Parameter in der *web.xml* konfiguriert werden bzw. benötigt diese, um korrekt zu funktionieren. Der Parameter *validateUrl* verweist auf die Adresse, unter der der CAS-Service für die Überprüfung der Tickets installiert ist. Sollte die Überprüfung des Tickets negativ ausfallen bzw. ist der Benutzer nicht angemeldet, leitet der Filter die Anfrage auf die unter *loginUrl* konfigurierte Adresse um. Fällt die Überprüfung des Tickets positiv aus, so wird sofort auf den unter *serverName* angegebenen Server weitergeleitet und die ursprüngliche Anwendung aufgerufen. Alternativ kann statt *serverName* auch *serviceUrl* verwendet werden, wenn der Benutzer nach einem Login stets auf denselben URL weitergeleitet werden soll (ohne Berücksichtigung des ursprünglichen Anwendungspaths). Dies kann nützlich sein, um Quereinstiege zu verhindern und dafür zu sorgen, dass die

Anwendung über einen zentralen Einstieg wieder korrekt initialisiert wird.

Die Konfiguration lässt eine Frage offen: Was passiert, wenn eine Anwendung für mehrere virtuelle Hostnamen (z.B. *www.foo.com* und *www.bar.com*) funktionieren soll? Es kann bei obigem Filter nur ein Hostname angegeben werden. Es sei an dieser Stelle auf das Projekt Extended CAS Client verwiesen, welches einen erweiterten CAS-Filter enthält. Dieser Filter ist in der Lage, genau dieses Problem zu beheben und ermöglicht es somit, mehrere Hostnamen zu konfigurieren. Der passende Ziel-Server wird dabei zur Laufzeit anhand des *HOST\_NAME*-Request-Parameters ermittelt.

### Hallo, CAS User!

Der Benutzer soll in der Beispielanwendung mit seinem Namen begrüßt werden, falls er sich erfolgreich anmelden konnte. Hierzu wurde ein Servlet entwickelt, das diese Aufgabe übernimmt (Listing 3).

Der CAS-Filter legt im *SessionContext* ein Attribut mit dem Namen *edu.yale.its.tp.cas.client.filter.user* an, das den Namen des angemeldeten Benutzers enthält. Der Benutzername wird im obigen Beispiel ausgelesen und als HTML-formatierter Text ausgegeben. Alternativ zu diesem Vorgehen existiert in den CAS-Client-Bibliotheken noch eine umfangreiche Tag Library, welche für die JSP-Entwicklung genutzt werden kann.

Alternativ zu der beschriebenen Vorgehensweise mit dem CAS-Filter können für die JSP-Entwicklung auch entsprechende Tags genutzt werden. Diese werden mit dem CAS Client ausgeliefert und funktionieren analog zum CAS-Filter.

Die Anwendung muss nun lediglich in den Tomcat-Server deployt werden. Danach kann der URL *http://localhost:8080/cas/hellouser/helloCasUser* in einem Browser-Fenster eingegeben werden. Daraufhin erscheint statt der angeforderten Adresse die Login-Seite des CAS-Servers. Die Anmeldung kann beispielsweise mit der Net-ID *foo* und dem Passwort *bar* erfolgen. Der *AuthenticationHandler* prüft die Anmeldeinformationen und leitet die Anfrage zum eigentlichen Ziel weiter – der Benutzer wird mit seinem Namen begrüßt.

Der CAS-Filter lässt sich für beliebige andere Webapplikationen (z.B. auch für Tomcats *HelloWorld*-Anwendung) konfigurieren. Wird danach eine solche Webapplikation per URL angesprochen, funktioniert dies nur mittels Anmeldung beim CAS-Server oder falls bereits ein gültiges Ticket vorhanden ist.

### Fazit

Mit CAS bekommt man eine ausgereifte Software frei zur Verfügung gestellt. Es lassen sich damit die üblichen Anforderungen eines zentralen Single-Sign-on-Service abdecken.

CAS bietet eine flexibel erweiterbare Architektur und zudem umfangreiche Integrationsmöglichkeiten in heterogenen Umgebungen. Für nahezu jede gängige Programmiersprache existieren Client-Bibliotheken (z.B. C#.NET oder PHP). Auch bestehende Portale können einfach an CAS angebunden werden (Unterstützung von JSR 168: Portlet Specification) und sogar eine Integration mit Acegi (Security System for Spring) wird unterstützt. Beim Einsatz von CAS ist man also auch für die Zukunft gut gewappnet.

### Sicherheit durch HTTPS

Der CAS-Server benötigt für die Authentifizierung aus Sicherheitsgründen SSL, damit die Kommunikation vollständig über HTTPS ablaufen kann. Die Anmeldeinformationen werden damit nicht mehr unverschlüsselt über das Netzwerk übertragen, was das Risiko potenzieller Angriffe drastisch reduziert.

Für das vorgestellte Beispiel muss daher im Server eine SSL-Unterstützung konfiguriert werden. Damit die CAS-Clients den Server via HTTPS kontaktieren können, müssen diese ebenfalls SSL-fähig gemacht werden. Die Java Secure Socket Extension (JSEE) ist seit JDK 1.4 bereits Bestandteil von Java, sodass kein zusätzlicher Installationsaufwand entsteht. Zur Verifikation des Server-Zertifikats durch den Client muss dieses jedoch in den lokalen Keystore für vertrauenswürdige Zertifikate aufgenommen werden. Für das Beispiel wurde der Einfachheit halber ein „self-signed Certificate“ erstellt. Dieses muss in den Keystore der Java-Laufzeitumgebung importiert werden, der sich im Pfad *\$JAVA\_HOME/jre/lib/security* befindet.

Da das Thema Zertifikate und deren Verwaltung auch für erfahrene Java-Entwickler nicht immer einfach zu handhaben sind, gibt es hierzu auch detaillierte Beschreibungen auf der CAS-Webseite oder in der Tomcat-Dokumentation.

### Benötigte Ressourcen

Um den Central Authentication Service zu betreiben und die Beispielanwendung damit auszuführen, werden folgende Ressourcen benötigt:

- Java SE 5.0: [java.sun.com/j2se/1.5.0/](http://java.sun.com/j2se/1.5.0/)
- Tomcat 5.5.17: [apache.speedbone.de/tomcat/tomcat-5/](http://apache.speedbone.de/tomcat/tomcat-5/)
- CAS Server 3.0.4: [www.ja-sig.org/downloads/cas/](http://www.ja-sig.org/downloads/cas/)
- CAS Client 2.0.11: [www.ja-sig.org/downloads/cas-clients/](http://www.ja-sig.org/downloads/cas-clients/)



Oliver Rummeyer und Jörg Düsterhaus sind als Berater bei der syngenio AG tätig. Sie beschäftigen sich mit komplexen Java EE-basierten Anwendungen in heterogenen IT-Landschaften von Banken und Finanzdienstleistern.

### Links & Literatur

- [1] [de.wikipedia.org/wiki/Single\\_Sign\\_On](http://de.wikipedia.org/wiki/Single_Sign_On)
- [2] [www.ja-sig.org/products/cas](http://www.ja-sig.org/products/cas)
- [3] [www-128.ibm.com/developerworks/web/library/wa-singlesign](http://www-128.ibm.com/developerworks/web/library/wa-singlesign)
- [4] [tomcat.apache.org/tomcat-5.5-doc/ssl-howto.html](http://tomcat.apache.org/tomcat-5.5-doc/ssl-howto.html)
- [5] [www.discursive.com/projects/cas-extend](http://www.discursive.com/projects/cas-extend)