СУ “Св. Климент Охридски”

Факултет по математика и информатика

Катедра “Информационни технологии”

Д И П Л О М Н А Р А Б О Т А

на тема

Среда за автентикация с използване на JAAS спецификацията при сървъри за приложения

Дипломант: Светлана Димова Станчева, фак. № М21352

Специалност / специализация: информатика / РСМТ

Научен ръководител: доц. д-р Боян Бончев

София

Февруари 2007

Съдържание

СЪДЪРЖАНИЕ 2

1. УВОД 5

1.1. ВЪВЕДЕНИЕ 5

1.2. ЦЕЛ И ЗАДАЧИ НА ДИПЛОМНАТА РАБОТА 6

1.3. СТРУКТУРА НА ДИПЛОМНАТА РАБОТА 6

2. JAAS СПЕЦИФИКАЦИЯ 8

3. МЕТОДИ ЗА АВТЕНТИКАЦИЯ ПРИ НЯКОИ СЪРВЪРИ ЗА ПРИЛОЖЕНИЯ 12

3.1. JBOSS СЪРВЪР 12

3.1.1. Конфигурация 13

3.1.2. Автентикация на отдалечен клиент 15

3.1.3. Логин модули 16

3.1.4. Същности и документи за самоличност 17

3.1.5. Обработчик на съобщения 17

3.2. BEA WEBLOGIC СЪРВЪР 17

3.2.1. Конфигурация 19

3.2.2. Набор от класове за автентикация 19

3.2.3. Логин модули 20

3.2.4. Същности и документи за самоличност 20

3.2.5. Обработчик на съобщения 22

3.3. СРАВНЕНИЕ МЕЖДУ МЕХАНИЗМИТЕ ЗА АВТЕНТИКАЦИЯ НА ДВАТА СЪРВЪРА 23

4. РЕАЛИЗАЦИЯ НА СРЕДАТА ЗА АВТЕНТИКАЦИЯ 25

4.1. АНАЛИЗ И ДИЗАЙН 25

4.1.1. Програмна автентикация с JAAS 25

4.1.1.1. Проблеми 25

4.1.1.2. Решение 25

4.1.2. Осъществяване на функционалност, която е обща за всички логин модули 26

4.1.2.1. Проблеми 26

4.1.2.2. Решение 27

4.1.3. Избиране на същност, идентифицираща потребителя 29

4.1.3.1. Проблеми 29

4.1.3.2. Решение 29

4.1.4. Осигуряване на гъвкавост на механизмите за автентикация при

стандартни J2EE приложения. *30*
4.1.4.1. Проблеми 30
4.1.4.2. Решение 30
4.1.5. Изискване на име и парола от потребителя в зависимост от типа автентикация. *30*
4.1.5.1. Проблем 30
4.1.5.2. Решение 31
4.2. Използвани технически средства 31
4.3. Реализация 32
4.3.1. Компоненти *32*
4.3.2. Конфигурация и логин контекст *33*
4.3.3. Същности *35*
4.3.4. Логин филтър *36*
4.3.5. Съобщения и обработчик на съобщения *37*
4.3.5.1. Поведение на обработчика на съобщения при FORM логин 40
4.3.5.2. Поведение на обработчика на съобщения при BASIC логин 42
4.3.6. Логин модули *43*
4.3.7. Конфигуриране на стекове от логин модули *45*
4.3.7.1. Приложение 45
4.4. Случаи на употреба 47

5. ТЕСТВАНЕ 51

5.1. Инсталиране и конфигуриране 51
5.2. Тестови приложения 52
5.2.1. Модули с декларативна автентикация *52*
5.2.2. Модули с програмна автентикация *53*
5.3. Тестови сценарии 54

6. ЗАКЛЮЧЕНИЕ И НАСОКИ ЗА БЪДЕЩО РАЗВИТИЕ НА РАЗРАБОТКАТА 55

7. ИЗПОЛЗВАНА ЛИТЕРАТУРА 57

8. РЕЧНИК НА ТЕРМИНИТЕ 59

9. ПРИЛОЖЕНИЕ С ФРАГМЕНТИ ОТ КОДА 61

9.1. Клас AppConfigurationEntryExtension 61
9.2. Изброен списък ControlFlag 61
9.3. Клас LoginModuleConfiguration, метод AppConfigurationEntry[] getAppConfigurationEntry(String context) 62
9.4. Клас ForwardLoginModule 62

9.4.1. Метод initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options) *62*
9.4.2. Метод boolean login() *63*
9.5. Клас CustomLoginContext, метод Principal getPrincipal() 63
9.6. Клас AuthStackProcessAction, метод Object run() 64
9.7. Клас LoginFilter 67
9.7.1. Метод init(FilterConfig config) *67*
9.7.2. Метод doFilter(ServletRequest request, ServletResponse response, FilterChain chain) *67*
9.8. Изброен списък AuthenticationStatusCallback 68
9.9. Клас UserIdPrincipal 68
9.10. Клас CreateSSOTokenLoginModule, метод boolean commit() 69

1. Увод

1.1. Въведение

Големите сървъри за приложения работят едновременно с много и различни видове приложения, които са реализирани от различни доставчици. При някои от тези приложения може да имат специфични изисквания към механизмите за автентикация. При други, механизмите за автентикация, които следва да се използват зависят от системата и средата, в която те ще се интегрират и от това за какви цели ще се използва тя.

При липса на стандарт, всяко от тези приложения би трябвало да се грижи само да реализира тази своя потребност и това би ставало по специфичен за него начин. Освен това, различната реализация и различният начин на конфигуриране на многото приложения, вървящи на даден сървър, би затруднило значително тяхното конфигуриране от администратора на сървъра. Много от приложенията може да са пригодени да използват само фиксиран метод на автентикация, което би затруднило интеграцията между тях (например конфигурирането на Single Sign-On).

Поради тази причина, сървърите за приложения се нуждаят да има независимост на механизмите за автентикация от реализацията на приложенията. Използването на стандарт за гъвкава автентикация, ще им даде възможност да обединят начина, по който се конфигурират приложенията и ще улесни внедряването и конфигурирането на различен набор от механизми за автентикация в зависимост от конкретните условия и потребители. Освен това, използването на стандарт за автентикация, улеснява в значителна степен и доставчиците на приложения, които няма нужда да се грижат да измислят и реализират тази функционалност сами.

"Java Authentication and Authorization Service" (JAAS) представлява Java версия на стандартната среда за автентикация "Pluggable Authentication Module" (PAM). Тази среда предоставя функционалност за автентикация и упълномощаване, която да се използва от различни приложения. Различните видове механизми за автентикация са капсулирани в логин модули. Всяко приложение може да използва един или комбинация от няколко логин модула. Чрез JAAS, сървърите могат да поддържат едновременно различни видове механизми за автентикация, които лесно могат да се конфигурират за различните приложения, без да е необходима промяна в реализацията им.

Достъпът от клиент към сървър за приложения (application server) може да се осъществява посредством различни протоколи - HTTP, SOAP, RMI, JRFC. За всеки от тези случаи, сървърът трябва да идентифицира потребителя, който стои зад съответната заявка. JAAS интерфейсите предоставят възможност идентификацията да се осъществява по унифициран начин, независимо от протокола и вида на приложението, към което е потребителската заявка.

1.2. Цел и задачи на дипломната работа

Цел: *Да се създаде среда за автентикация, базираща се на JAAS, предназначена да бъде използвана от сървъри за приложения*.

За да се постигне поставената цел, трябва да се изпълнят следните задачи:

Зад. 1: Да се проучат JAAS интерфейсите и възможностите, които те предлагат.

Зад. 2: Да се установи по какъв начин и в каква степен известни сървъри за приложения използват JAAS.

Зад. 3: Да се създаде собствена среда за автентикация на уеб приложения, която демонстрира възможностите на JAAS.

Зад. 4: Да се създадат тестови приложения, чрез които да се провери правилното функциониране на средата.

Изисквания: Средата за автентикация трябва да предоставя възможност за приложенията да правят автентикация по стандартизиран начин. Освен това, трябва да има възможност за взаимозаменяемост на механизмите за автентикация и Single Sign-On. Тази среда трябва да се интегрира със функционалността на уеб контейнера и да е съобразена с изискванията на Servlet спецификацията [7].

Ограничения: За да се реализира модул, предоставящ услуги, покриващи горе изброените изисквания и нужди, той трябва да използва функционалност, специфична за съответния сървър, за да интегрира тази услуга с контейнерите.

1.3. Структура на дипломната работа

Дипломната работа е организирана в девет глави като първите шест от тях

представляват основната част, а последните три са приложения:

- **Основна част**

  Първа глава: Увод. Тази глава ни въвежда в темата и проблемите, които се разглеждат в дипломната работа. Тя ни дава и информация за целите, задачите и структурата на дипломната работа.

  Втора глава: JAAS спецификация. В тази глава се разглежда набора от функционалности, които JAAS спецификацията предлага.

  Трета глава: Методи за автентикация при някои сървъри за приложения. Разглеждат се решенията при някои от сървърите за приложения и се прави сравнение между тях.

  Четвърта глава: Реализация на среда за автентикация. Описва се собствената разработка на среда за автентикация за уеб приложения (web applications), базираща се на JAAS.

  Пета глава: Тестване. Тук се описва по какъв начин се инсталира и се конфигурира приложението. Описват се тестовите приложения и по какъв начин се извършва тестването с тях.

  Шеста глава: Заключение. В тази глава се прави равносметка на свършената работа и се разглеждат насоките за бъдещо развитие на продукта.

- **Приложения**

  Седма глава: Използвана литература. Тук се съдържа списъкът с използвана литература.

  Осма глава: Речник на термините. В тази глава са изброени всички термини, които са били използвани в дипломната работа както и техните оригинални названия на английски език.

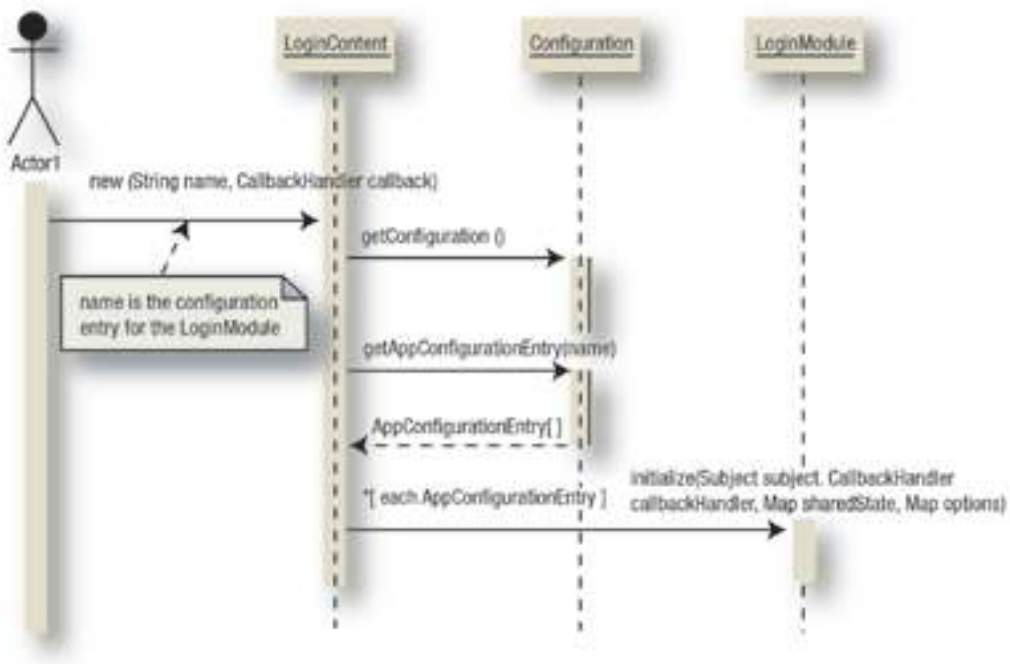
  Девета глава: Приложение с фрагменти от кода. Тук са поместени по-интересните части от кода.

2. JAAS спецификация

Автентикацията и упълномощаването са едни от най-основните механизми за осигуряване на сигурността [12]. Автентикацията представлява идентифициране и доказване на самоличността на потребителя. Упълномощаването представлява решение дали на потребителя да се даде право да достъпи определени данни или ресурси. JAAS предоставя стандартни услуги за извършване на тези две операции. Тези услуги стават част от стандартното издание на JDK (Java Development Kit) за версия 1.4.

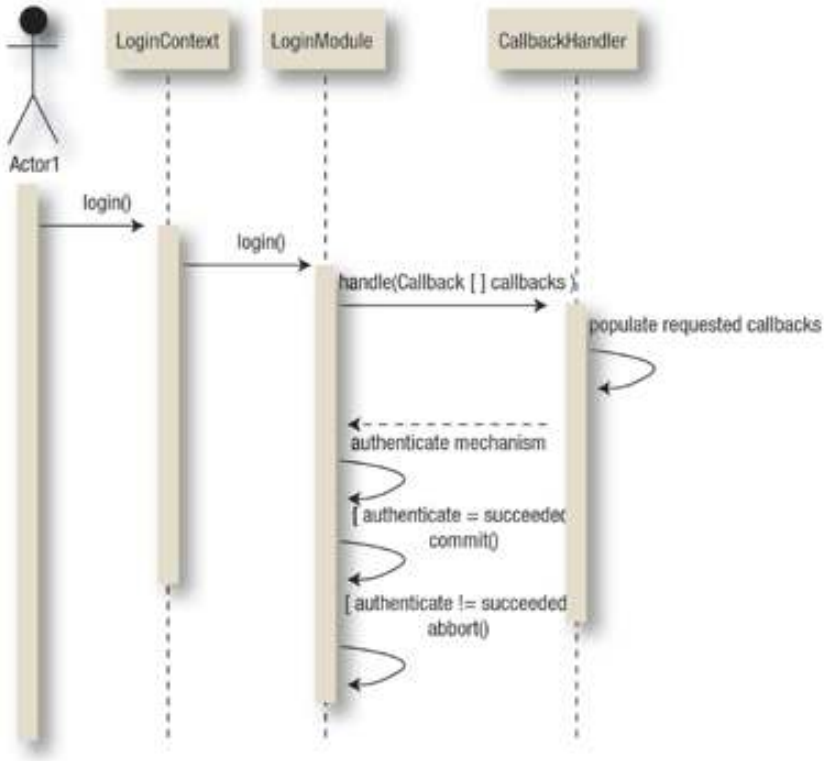
При JAAS стандарта, механизмите за автентикация се описват в конфигурационен файл. Автентицирания потребител се представя чрез обект от тип Subject (субект). Субектът може да съдържа набор от същности (principals) и документи за самоличност (credentials). Даден потребител може да притежава няколко същности – например човек с име Иван Иванов, гражданин на страната си с ЕГН 1234567890, студент в университет с факултетен номер У12345. Също така, потребителят може да притежава множество документи, доказващи самоличността му – лична карта, студентска книжка, шофьорска книжка. При успешна автентикация данните, използвани за установяване на самоличността на потребителя, се записват в субекта. Този субект се използва по-късно за упълномощаване на потребителя за извършването на различни действия. Потребителските данни, които ще се използват в процеса на автентикация зависят от конфигурираните механизми в конфигурационния файл.

Автентикацията започва със създаване на екземпляр на логин контекст (LoginContext). Подава му се като първи аргумент името на стека от логин модули, който трябва да се вземе от конфигурационния файл. Освен това, логин контекстът получава и обработчик на съобщения (CallbackHandler), който той предава на логин модулите (login modules). Логин модулите са тези, които осъществяват различните механизми за автентикация (например: име и парола, клиентски сертификат, Kerberos). Чрез обработчика на съобщения те могат да се обърнат обратно към приложението и да изискат допълнителни данни за потребителя. Логин контекстът също подава и субекта на логин модулите, така че те да могат да му добавят там съответните данни.



Фиг. 1 Инициализиране на логин контекст със стек от логин модули зададени в конфигурацията [12].

След това, на така инициализирания логин контекст му се извиква метода login(). Логин контекстът, от своя страна, извиква последователно метода login() на всеки от конфигурираните логин модули. Всеки логин модул изисква и обработва определен вид потребителски данни и документи в зависимост от това какъв механизъм на автентикация реализира. Изискването на потребителските данни се извършва посредством съобщения (callbacks), които се подават на обработчика на съобщения (callback handler). Това представлява първата фаза на автентикацията. Тя завършва успешно, ако всички извикани логин модули с флагове REQUIRED и REQUISITE завършат успешно. Ако няма логин модули с такива флагове, то тогава поне един от логин модулите трябва да е завършил успешно. Ако първата фаза завърши успешно, то по време на втората фаза се извиква последователно метода commit() на всички логин модули. Чрез този метод се финализира логин процеса като логин модулите, които са завършили успешно през първата фаза, добавят съответните същности и документи за самоличност в субекта. Ако първата фаза завърши неуспешно, то тогава логин контекстът извиква последователно метода abort() за всеки от логин модулите. Чрез този метод логин модулите изчистват всякакви междинни състояния и данни, запазени по време на изпълнение на първата фаза.



Фиг. 2 Логин процес [12].

Всеки от логин модулите може да се конфигурира с един от следните флагове, които представляват константи от тип AppConfigurationEntry.LoginModuleControlFlag:

- Required – Логин модулът трябва задължително да успее. Независимо дали успее или не, логин контекстът продължава с изпълнението на следващите логин модули.
- Requisite - Логин модулът трябва задължително да успее. Ако успее, логин контекстът продължава с изпълнението на следващия логин модул. Ако не успее, логин контекстът прекратява изпълнението на стека от логин модули.
- Sufficient – Не е задължително този логин модул да успее. Ако успее, останалите логин модули не се изпълняват. Ако не успее, се продължава с изпълнението на следващия логин модул.
- Optional - Не е задължително този логин модул да успее. Независимо дали успее

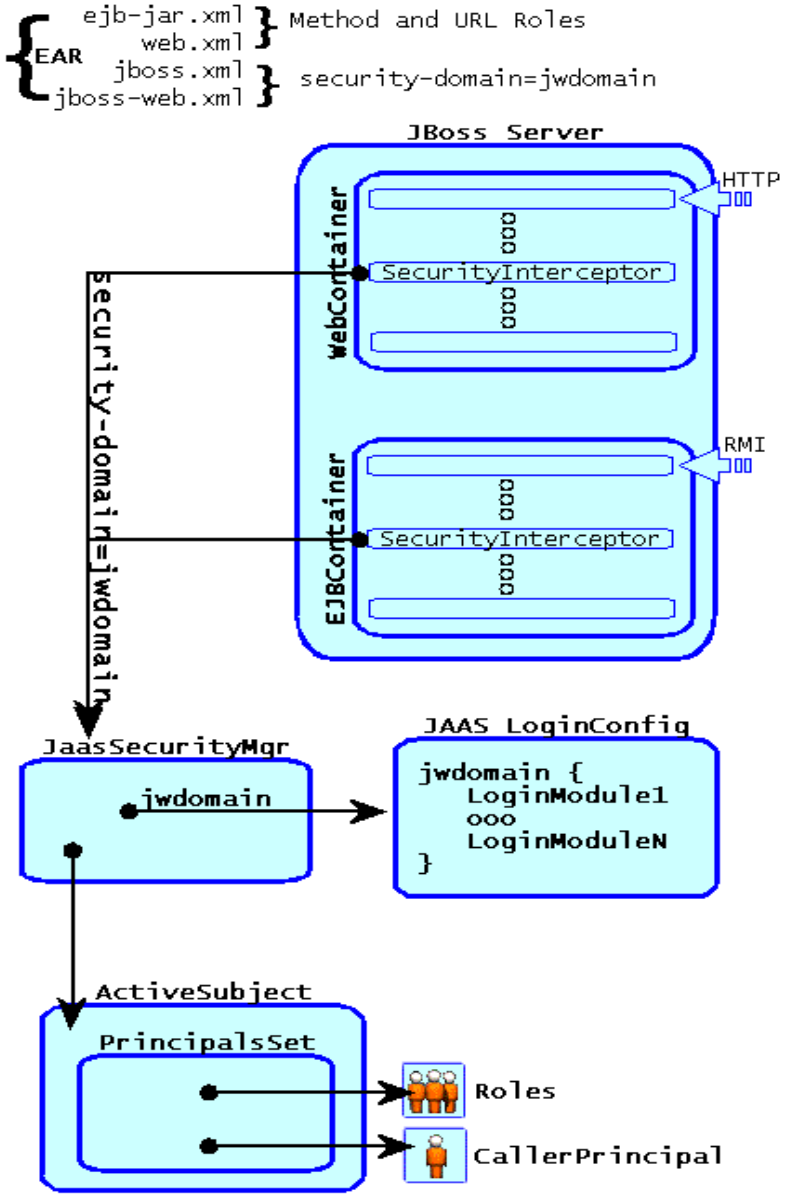
или не, логин контекстът продължава с изпълнението на следващите логин модули.

3. Методи за автентикация при някои сървъри за приложения

В тази глава са разгледани механизмите за автентикация на два от по-известните сървъри за автентикация – JBoss и WebLogic.

3.1. JBoss сървър

Сървърът JBoss [9] използва собствена реализация на java.lang.SecurityManager - JaasSecurityManager, в която реализира своята функционалност свързана с автентикация и проверка на правата на потребителите. Тази разработка се базира на JAAS интерфейсите. Това означава, че JaasSecurityManager използва JAAS интерфейси, за да реализира услугите, които предлага. Поради тази причина, наборът от механизми за автентикация и проверка на правата могат лесно да се подменят с други. Освен това, тази разработка на SecurityManager може да се замени с друга, която осъществява автентикацията и проверката на правата по друг начин, който не е базиран на JAAS. За да се направи това, обаче, е необходимо да се използват специфични интерфейси, зададени от JBoss.



Фиг. 3 Реализация на автентикация използваща JAAS при JBOSS [9].

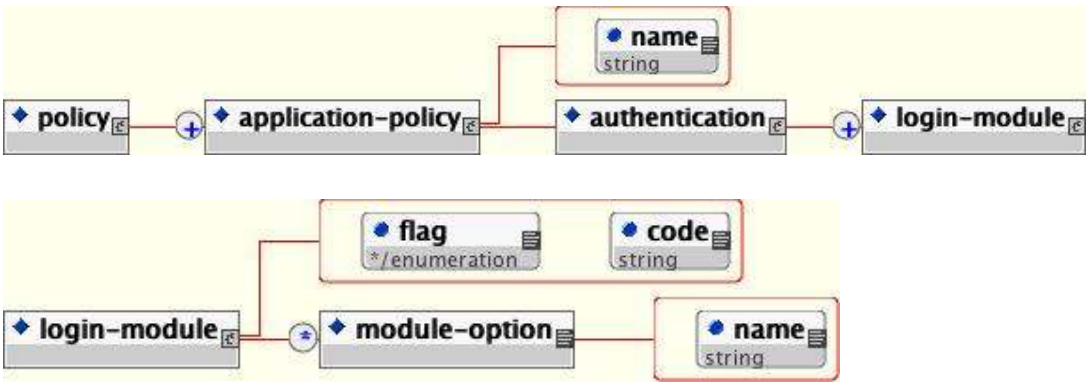
3.1.1. Конфигурация

Както знаем, от Servlet спецификацията [7], уеб модулите съдържат web.xml, в който се декларират различни характеристики на приложението, чрез които се конфигурира поведението на уеб контейнера (web container) при достъпването на ресурси от това приложение. В JBoss има и допълнителен xml файл – jboss-web.xml,

който съдържа конфигурации, специфични за JBoss. В този файл е добавен специален етикет (tag) “security-domain”, който може да съдържа произволен низ от символи. Този низ, се използва като идентификатор на стека от логин модули. Уеб контейнерът инициализира мениджъра по сигурността (security manager) като му подава стойността, която е записана в етикета “security-domain”. Тази стойност се използва от мениджъра по сигурността при инициализирането на логин контекст (login context), необходим за извършване на автентикацията.

Подобно е и поведението на EJB контейнера.

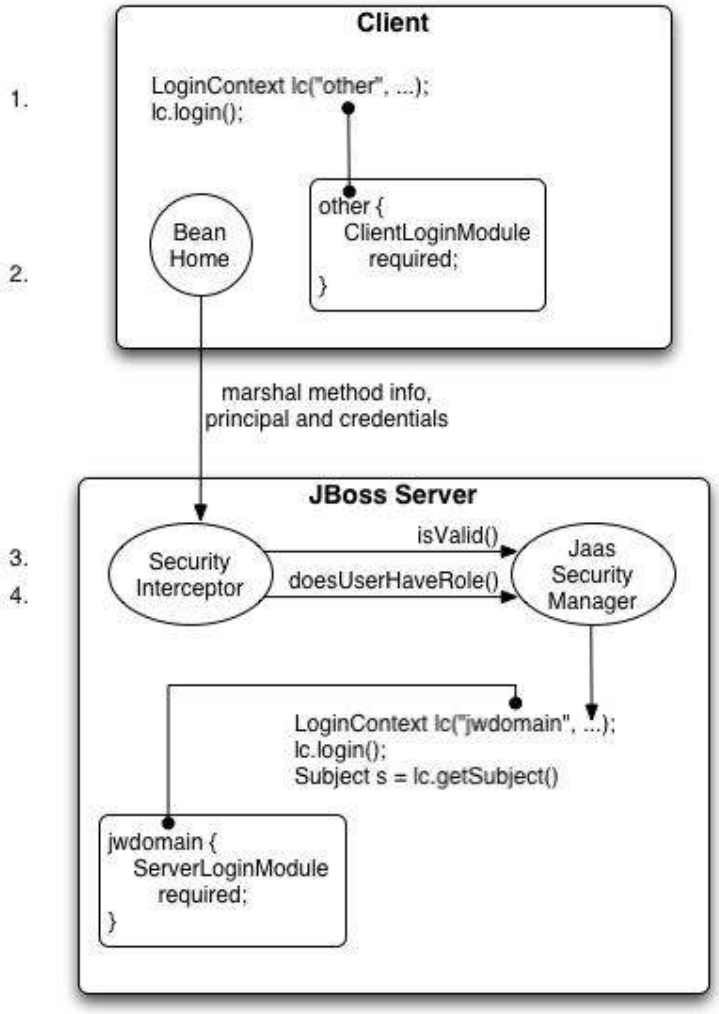
Конфигурацията от логин модули се намира във файла security_config.xml. Неговият синтаксис е показан на следната фигура:



Фиг. 4 Синтаксис на конфигурационния файл security_config.xml [9].

Стойността на атрибута “name” на етикета “application-policy” е името на стека от логин модули, който ще се изпълни при логин в уеб модул (web module), притежаващ етикет “security-domain”, чиято стойност съвпада с това име.

3.1.2. Автентикация на отдалечен клиент



Фиг. 5 Автентикация на отдалечен клиент към JBoss сървър [9].

Клиентът първо трябва да направи логин, използвайки JAAS интерфейси. Този логин се изпълнява на клиентската страна. Това не води до осъществяване на реална автентикация. По този начин клиентското приложение само взима необходимите същности и документи за самоличност от потребителя. Тези данни се пазят при клиента и се асоциират с всяко следващо отдалечено извикване (remote invocation) на метод от сървъра.

Клиентското приложение осъществява тази операция като създава екземпляр на LoginContext със име на стека от логин модули “other” и му извиква метода login().

Когато клиентът се опита да осъществи отдалечен достъп до сървъра, тогава асоциираните същности и документи за самоличност се пренасят до сървъра. Сървърът прави автентикация, преди да се изпълни съответната функционалност, инициирана от клиента. Това се осъществява чрез мениджъра по сигурността. Той извиква логин модулите конфигурирани за съответната област на сигурност (security domain). При успешна автентикация се създава субект, с който се асоциират същностите и документите за самоличност на потребителя.

3.1.3. Логин модули

JBoss предлага известен набор от логин модули, които покриват нуждите за автентикация за по-често срещаните сценарии. Освен това, се предлага и възможност за внедряване на външни логин модули. Съществена особеност при логин модулите предлагани от JBoss е, че те осъществяват механизми не само за автентикация, но и за проверка на правата. Друга особеност на предлаганите логин модули е, че те са предназначени не само за различни логин механизми (име и парола, клиентски сертификат), но и за различни места, от където се четат потребителските регистрации – база от данни, LDAP (Light Directory Access Protocol) сървър.

При писане на логин модул за JBoss сървър, освен правилата споменати в JAAS спецификацията, трябва да се съблюдават и някои изисквания, които са специфични за JBoss. Предоставя се абстрактен клас org.jboss.security.auth.spi.AbstractLoginModule, който осъществява специфичните операции. Всеки логин модул трябва да наследи този абстрактен клас като пренапише следните методи:

- void initialize(Subject, CallbackHandler, Map, Map) ако логин модулът използва специфични опции.
- boolean login() за да осъществи автентикацията. Този метод трябва да инициализира променливата loginOk съответно със стойност “истина” или “лъжа”, в зависимост от това дали логинът е успял или не.
- Principal getIdentity(), който трябва да връща самоличността на потребителя, автентициран от метода логин.
- Group[] getRoleSets(), който трябва да връща поне една група, съдържаща ролите на потребителя и една група, съдържаща самоличността на потребителя, който е направил заявката към сървъра.

3.1.4. Същности и документи за самоличност

В JBoss могат да се използват различни реализации на java.security.Principal в зависимост от данните, които се пазят там. Използва се също и java.security.acl.Group, който наследява java.security.Principal, за групиране на данни за потребителя. Дефинирани са 2 групи от данни съответно за съхраняване на потребителските роли и за съхраняване на потребителската самоличност.

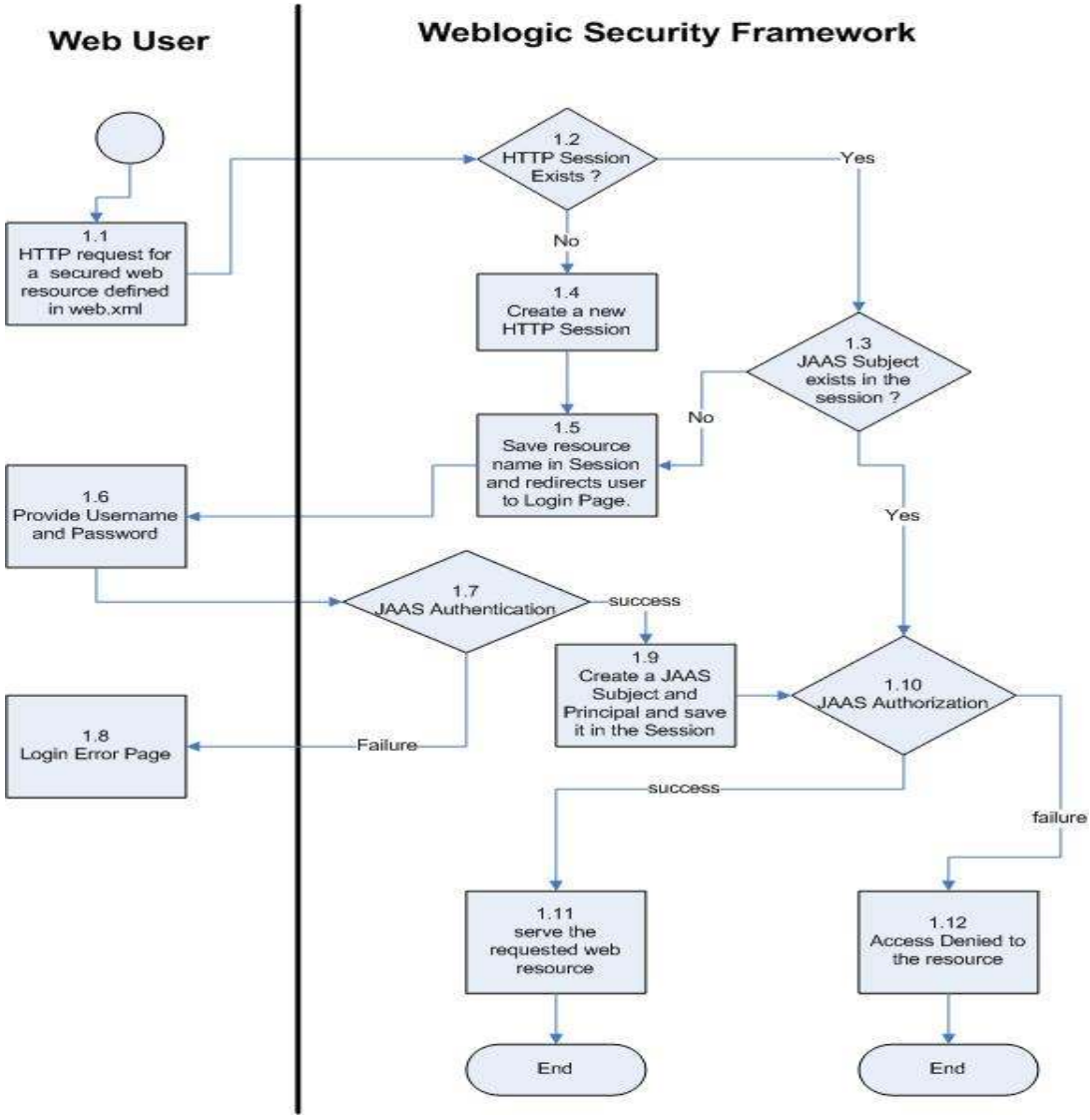
3.1.5. Обработчик на съобщения

JBoss има собствена реализация на javax.security.auth.callback.CallbackHandler. Тя се използва от JaasSecurityManager за предоставяне на данните, поискани от логин модулите. Има възможност мениджърът по сигурността да се конфигурира да използва и друг обработчик на съобщения, предоставен от клиента.

3.2. BEA WebLogic сървър

Механизмът за автентикация при WebLogic [10] се базира на JAAS. Той е реализиран чрез доставчик на услуга за автентикация (authentication provider). Един такъв доставчик може да има само един логин модул. Конфигурирането на стек от повече от един механизма за автентикация се осъществява чрез конфигурирането на няколко доставчика на тази услуга. За всеки от тези доставчици може да се конфигурира контролен флаг.

Освен това, сървърът поддържа и всички изисквания за декларативна защита на Уеб ресурси. Сървърът поддържа три различни вида J2EE (Java 2 Enterprise Edition) механизми за автентикация – BASIC, FORM, CLIENT-CERT. Долната графика показва по какъв начин се извършва автентикацията при FORM механизъм:



Фиг. 6 Логин във WebLogic използвайки FORM тип автентикация [11].

Когато до сървъра достигне заявка за достъп до даден ресурс, той проверява дали има HTTP сесия за този клиент. Ако няма HTTP сесия или тя не съдържа субект, това означава, че клиентът не е автентициран. При това положение, сървърът създава HTTP сесия и запазва в нея информация за това какъв ресурс клиентът е искал да достъпи. След това сървърът препраща клиента към логин страницата, като по този начин го подканя да въведе име и парола. След като потребителят изпрати формуляра за логин, сървърът извършва автентикация по JAAS. При успешна автентикация се създава субект, който се пази в HTTP сесията. Този субект се използва по нататък при проверките за упълномощаване на потребителя. След това, сървърът прочита от

сесията кой е бил първоначално заявения ресурс от потребителя и го препраща към него. При неуспешна автентикация, потребителя се препраща към страница със съобщение за грешка.

3.2.1. Конфигурация

WebLogic използва стандартния конфигурационен клас предоставен от Sun, който работи с конфигурационен файл.

3.2.2. Набор от класове за автентикация

WebLogic използва weblogic.servlet.security.ServletAuthentication при декларативна автентикация на уеб приложения [15]. Този клас би могъл да се използва и директно от приложенията за извършване на програмна автентикация. Това става като се извика някой от методите на този клас. Класът ServletAuthentication притежава много и разнообразни методи за логин. Някои от тях използват фиксиран механизъм за автентикация – например за име и парола и за клиентски сертификат. Други методи позволяват използването на произволен набор от логин модули, конфигурирани за съответното приложение, както и подаването на собствен обработчик на съобщения. Тук има и много помощни методи които биха могли да се използват при програмна автентикация и при реализиране на собствени логин модули и собствен обработчик на съобщения. Такъв метод е например getTargetURLForFormAuthentication, който връща оригиналното URL запазено в сесията преди препращането към логин страницата. Други методи пък позволяват взимане, регенериране и унищожаване на бисквитката представляваща идентификатор на сесията (session id cookie).

Класът weblogic.security.services.Authentication също може да се използва за програмна автентикация. Това може да стане като се използва този клас в комбинация със ServletAuthentication по следния начин [14]:

```
CallbackHandler handler = new URLCallbackHandler(username, password);
Subject mySubject = weblogic.security.services.Authentication.login(handler);
weblogic.servlet.security.ServletAuthentication.runAs(mySubject, request);
```

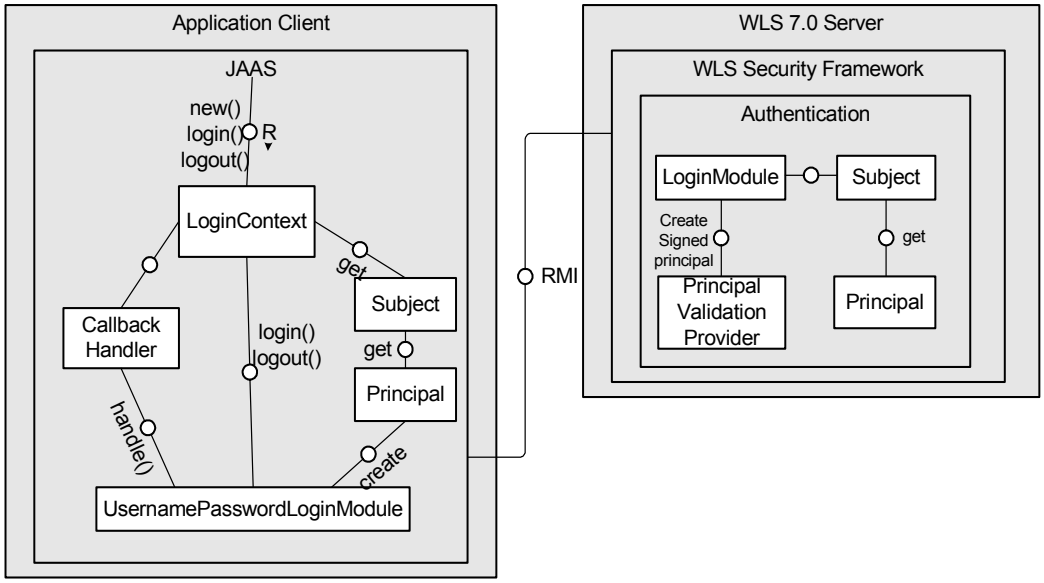
Класът Authentication се използва предимно при писане на собствени логин модули. Всеки логин модул, чрез който искаме да се автентицираме на сървъра, трябва да извиква метода Authentication.login(CallbackHandler).

За автентикация на отдалечени клиенти може да се използва следната

функционалност:

weblogic.security.auth.Authenticate.authenticate(Environment, Subject);

Като първи параметър се подава JNDI (Java Naming Directory Infrastructure) среда (environment), от която се взимат личните данни на потребителя. След успешна автентикация, личните данни се добавят към субекта (subject) и се връщат на приложението.



Фиг. 7 Автентикация на отдалечен клиент към WebLogic сървър [12].

Ако искаме да напишем логин модул, който да се изпълнява на клиента, трябва да направим така, че той да извиква този метод, за да се направи автентикация на сървъра.

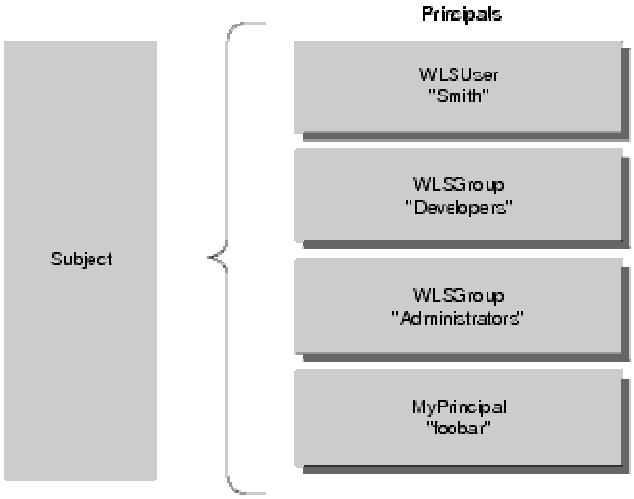
3.2.3. Логин модули

Както вече споменахме по-горе, логин модулите, осъществяващи автентикация към WebLogic сървъра, трябва задължително да правят обръщения към weblogic.security.auth.Authentication.authenticate().

3.2.4. Същности и документи за самоличност

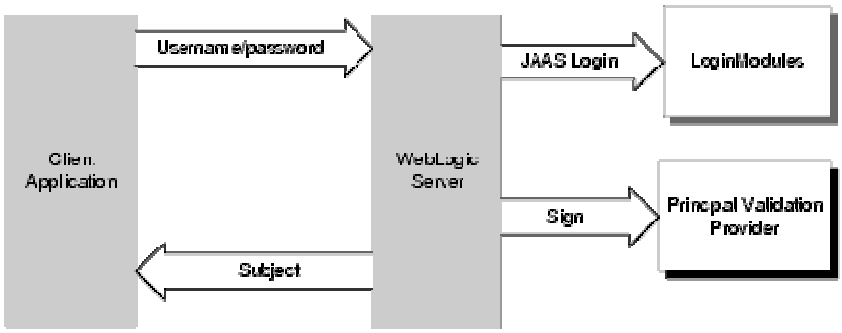
Във WebLogic, има два вида реализации на java.security.Principal – WLSUser и WLSGroup. Те трябва да се използват за същности, представляващи съответно

потребители и потребителски групи.



Фиг. 7 Видове същности, които се използват във WebLogic сървър [10].

Личните данни се съхраняват в субекта подписани и при всеки опит за четене, се проверят за достоверност. Това се извършва от доставчик на услуга за проверка на същности (Principal Validation Provider). Наличният доставчик на такива услуги във WebLogic може да се замени с друг, дефиниран от клиента.



Фиг. 8 WebLogic подписва същностите, преди да ги добави към субекта [13].

Когато за едно приложение са конфигурирани повече от един логин модула, те трябва да направят проверка за съществуващ вече идентификатор на потребител в субекта и в такъв случай да не добавят нов.

Когато е необходимо да се определи текущия потребител на приложението, тогава трябва да се избере една от многото същности в субекта. В такъв случай се избира същността от тип WLSUser. Ако в субекта има повече от една същности от този тип, то тогава се избира първата такава същност върната от метода Subject.getPrincipals().iterator(). Ако няма нито един обект от тип WLSUser, тогава се избира първата същност, която не представлява потребителска група. Ако няма такава, се връща стойност null.

Горепосоченият алгоритъм за определяне на текущия потребител се използва от методите HttpServletRequest.getUserPrincipal(), HttpServletRequest.getRemoteUser() и EJBContext.getCallerPrincipal().

WebLogic предоставя и допълнителен метод за извличане на текущия потребител, който може да се използва от всеки тип приложение:

weblogic.security.SubjectUtils.getUsername(weblogic.security.Security.getCurrentSubject());

3.2.5. Обработчик на съобщения

WebLogic използва стандартните съобщения: NameCallback, PasswordCallback и TextInputCallback. NameCallback и PasswordCallback се използват стандартно съответно за изискване на потребителско име и парола. TextInputCallback се използва за други допълнителни данни, които биха могли да се изискат от потребителя.

WebLogic дефинира и собствени съобщения – URLCallback и UserInfoCallback за извличане на информация за текущото URL, както и за получаване на екземпляр за класа UserInfo.

Всяко приложение може да използва собствен обработчик на съобщения, който да изисква съответните данни от потребителя по специфичен начин. WebLogic предоставя вграден обработчик на съобщения, който може да се използва от приложенията. Това е класът weblogic.security.URLCallbackHandler. Той може да се използва за извличане на данни, съдържащи се в потребителското име, парола, URL (Uniform Resource Locator) и контейнер контекста (реализиран чрез класа ContextHandler).

3.3. Сравнение между механизмите за автентикация на двата сървъра

Характерно и за двата сървъра е, че използват специфични типове същности, подходящи за техните собствени цели. И при двата сървъра имаме специфичен тип същност, който представлява идентификатор на потребителя. Разликата е в това, че при JBoss имаме същности представляващи потребителски роли, а във WebLogic имаме същности представляващи потребителски групи. Необходимостта при JBoss от същности, представляващи потребителски групи се дължи на факта, че те използват JAAS логин модули също и за упълномощаване. При WebLogic, за разлика от WebLogic, същностите съхранявани в субекта са подписани, което затруднява фалшифицирането им.

При автентикация от отдалечен клиент, WebLogic има изискване логин модулите, конфигурирани при клиента да извършат обръщение към специфичен метод, който осъществява автентикацията на сървъра. При JBoss при този сценарий имаме два стека от логин модули – един, който се изпълнява при клиента и служи само за събиране на потребителските данни и втори, който се изпълнява на сървъра, като използва данните, които е събрал предишният стек. По този начин, конфигурирането на стековете от логин модули се затруднява, тъй като двата стека – при клиента и на сървъра – трябва да се поддържат да са синхронизирани. Потребителските данни и документи за самоличност, които се изискат от клиентските логин модули, трябва да отговарят на потребителските данни и документи за самоличност, които са в състояние да обработят логин модулите, конфигурирани на сървъра. Затова, модела на WebLogic изглежда по-надежден, въпреки че е по-труден за реализация от страна на сървъра.

При JBoss имаме изискване потребителските логин модули да наследяват специфичен клас като реализират определен набор от методи. При WebLogic отново решението е по-опростено и няма такова изискване.

Следната таблица систематизира някои от основните характеристики на механизмите за автентикация при двата сървъра:

Характеристика	JBoss	WebLogic
Поддържани типове	BASIC, FORM и CLIENT-CERT	BASIC, FORM и CLIENT-CERT

Характеристика	JBoss	WebLogic
автентикация според Servlet спецификацията		
Поддръжка на JAAS	Чрез мениджър по сигурността	Чрез доставчик на услуга за автентикация
Същности	Използва се java.security.acl.Group с имена “Roles” и “CallerPrincipal”.	Използват се WLSUser и WLSGroup.
Обработчик на съобщения	Работи се със SecurityAssociationHandler. Може да се замени с друг чрез атрибута “CallbackHandlerClassName” на мениджъра по сигурността.	Могат да се използват URLCallbackHandler или SimpleCallbackHandler или потребителски дефиниран обработчик на съобщения.
Механизми за автентикация	- Име и парола - X.509 сертификати - Логин с фиксирана самоличност - Логин модул осигуряващ изпълнението на останалите логин модули със специална роля - Логин модул проверяващ правата на потребителя - Клиентски логин модул за събиране на потребителските данни и асоцииране им с потребителската заявка.	- Име и парола - X.509 сертификати - SAML (Security Assertion Markup Language) - SPNEGO (Simple and Protected Negotiate) протокол
Поддръжка на потребителски логин модули	Потребителските логин модули трябва да наследяват AbstractLoginModule.	Потребителските логин модули трябва да извикват Authentication.authenticate().

Таблица 1: Основни характеристики на механизмите за автентикация при JBoss и WebLogic

4. Реализация на средата за автентикация

4.1. Анализ и дизайн

4.1.1. Програмна автентикация с JAAS

4.1.1.1. Проблеми

Сървърите в много случаи се нуждаят да осъществят специфична функционалност по време на автентикация – например създаване и проследяване на потребителска сесия, одит информация. Затова те са принудени да изискват от разработчиците на приложения и логин модули да използват, наред със стандартните интерфейси, също и специфични за сървъра функционалности. Както видяхме вече, при JBoss имаме AbstractLoginModule, който трябва да се наследява от всеки логин модул и да реализира неговите методи. При WebLogic логин модулът се задължава да извика специфичен метод. Освен това програмната автентикация се извършва чрез специфични класове и методи.

4.1.1.2. Решение

Този проблем би могъл да се реши, ако се намери начин при изпълнението на логин контекста и логин модулите да се извика автоматично функционалност от сървъра, така че той да може да изпълни необходимите за него действия. JAAS спецификацията предвижда възможности за създаване и внедряване на собствени логин модули, същности, документи за самоличност, обработчик на въпроси и конфигурация [1], [3]. Тя обаче не предоставя непосредствена възможност за създаване и използване на собствен логин контекст.

От наличните средства, които JAAS спецификацията ни предоставя, бихме могли да използваме конфигурацията, като средство за внедряване на собствена функционалност. Тя изглежда най-подходяща за тази цел, защото процесът на автентикация започва от нея. Чрез нея се определя какви и колко логин механизми ще бъдат изпълнени. Затова, конфигурацията би могла да се реализира по такъв начин, че за всеки стек от логин модули тя да връща фиксиран масив от логин модули, състоящ се от един служебен логин модул. Този служебен логин модул може да служи като посредник, който само предава информацията за това какъв стек от логин модули трябва да се изпълни на специално реализиран за целта логин контекст. Този логин контекст би могъл да намери съответния стек от логин модули и да направи автентикация за него.

По този начин, ще може да се изработи приложение, което да позволява да се осъществява автентикация (програмна и декларативна) без използването на специфична функционалност от страна на логин модулите и приложенията.

4.1.2. Осъществяване на функционалност, която е обща за всички логин модули

4.1.2.1. Проблеми

Съществуват редица случи, когато при успешна автентикация се налага да се направят някои заключителни действия. Когато в стека от логин модули имаме само един логин модул, тогава можем лесно да конфигурираме втори логин модул, който да изпълнява заключителните действия. Следният пример демонстрира как може да стане това:

```
login-module-stack-name {
  NameAndPasswordLoginModule REQUISITE;
  FinalizingLoginModule SUFFICIENT;
};
```

Когато обаче в стека имаме повече от един логин модули с флаг “SUFFICIENT”, тогава това се оказва доста трудна, дори невъзможна задача. Един доста популярен пример за няколко алтернативни логин механизми в един стек е следният:

```
login-module-stack-name {
  ClientCertificateLoginModule SUFFICIENT;
  NameAndPasswordLoginModule SUFFICIENT;
};
```

Чрез този стек от логин модули даваме право на потребителя да се автентицира с един от двата механизма. Ако потребителят притежава клиентски сертификат, той може да избере по-сигурния и по-силния механизъм за автентикация със сертификат. Ако, обаче, потребителят не притежава сертификат или достъпва приложението чрез HTTP клиент, който не поддържа SSL (Secure Socket Layer), той пак би трябвало да може да достъпи ресурса, като подаде име и парола.

Ако в този стек искаме да конфигурираме логин модул, който да извърши някакви заключителни действия, независимо кой от двата механизма за автентикация е бил използван, се оказва, че голямото разнообразие от флагове за логин модулите не е достатъчно. Единственият начин да се постигне приблизително желания резултат е

демонстриран в долния пример:

```
login-module-stack-name {
  ClientCertificateLoginModule OPTIONAL;
  FinalizingLoginModule SUFFICIENT;
  NameAndPasswordLoginModule REQUISITE;
  FinalizingLoginModule SUFFICIENT;
};
```

При тази конфигурация има два основни проблема:

- Единият логин модул се повтаря два пъти, което утежнява стека и го прави по-неразбираем.
- FinalizingLoginModule ще се изпълни дори ако ClientCertificateLoginModule не успее, което не съвпада с нашето желание. За да укажем, че искаме FinalizingLoginModule да се изпълни само ако ClientCertificateLoginModule е успял, то трябва да конфигурираме ClientCertificateLoginModule с флаг “REQUISITE”. Ако направим това, обаче, тогава никога няма да стигнем до изпълнението на NameAndPasswordLoginModule.

4.1.2.2. Решение

За решаването на този проблем, предлагам разширяване на стандарта JAAS и добавянето на нов флаг “CLOSING” към вече съществуващите SUFFICIENT, OPTIONAL, REQUIRED и REQUISITE. Неговата семантика е следната:

- За методите login() и commit() важат следните правила:
  - Логин модул с флаг CLOSING се изпълнява само и винаги, когато стекът, образуван от предхождащите го логин модули, е завършил успешно.
  - Логин модул с флаг CLOSING е последният логин модул, който се изпълнява от стека. Ако след него има други логин модули с флагове различни от CLOSING, те не се изпълняват.
  - Ако в стека има повече от един логин модул с флаг CLOSING, то те се изпълняват последователно в реда, в който са декларирани.
  - Цялостната автентикация успява, ако всички логин модули с флаг CLOSING успеят.

- Както знаем от JAAS спецификацията, при извикването на методите abort() и logout() не се взимат предвид флаговете на съответните логин модули. Изпълняват се последователно съответните методи на всички логин модули от стека, за да сме сигурни, че цялата информация относно автентикацията на потребителя ще се изтрие успешно.

Ако използваме флаг CLOSING, стекът от логин модули ще изглежда по следния начин:

```
login-module-stack-name {
  ClientCertificateLoginModule SUFFICIENT;
  NameAndPasswordLoginModule SUFFICIENT;
  FinalizingLoginModule CLOSING;
};
```

На таблица 2 се вижда какъв ще е резултатът от първата фаза на автентикацията в зависимост от изпълнението на отделните логин модули. Символът "*" означава, че методът login() на съответния логин модул няма да се извика от логин контекста.

Login Module Name	**Flag**	**Login Status**				
ClientCertificateLoginModule	sufficient	pass	pass	fail	fail	fail
NameAndPasswordLoginModule	sufficient	*	*	pass	pass	fail
FinalizingLoginModule	closing	pass	fail	pass	fail	*
Overall Authentication		pass	fail	pass	fail	fail

Таблица 2: Резултат от изпълнение на първа фаза от автентикацията.

При успешно преминаване на първата фаза на автентикацията се извиква последователно метода commit() за логин модулите конфигурирани в стека. От горната таблица се вижда, че за горепосочения стек имаме два случая, когато първата фаза завършва успешно. Възможните резултати при извършването на втората фаза от автентикацията са изброени в следващите две таблици. Символът "*" означава, че методът commit() на съответния логин модул няма да бъде извикан от логин контекста.

Login Module Name	**Flag**	**Login Status**	**Commit Status**		
ClientCertificateLoginModule	sufficient	pass	pass	pass	fail

NameAndPasswordLoginModule	sufficient	*	*	*	fail
FinalizingLoginModule	closing	pass	pass	fail	*
Overall Authentication		pass	pass	fail	fail

Таблица 3: Резултат от изпълнението на втората фаза от автентикацията при успешен първи логин модул от първата фаза.

Login Module Name	Flag	Login Status	Commit Status		
ClientCertificateLoginModule	sufficient	fail	fail	fail	fail
NameAndPasswordLoginModule	sufficient	pass	pass	fail	pass
FinalizingLoginModule	closing	pass	pass	*	fail
Overall Authentication		pass	pass	fail	fail

Таблица 4: Резултат от изпълнение на втора фаза от автентикацията при успешен втори логин модул от първата фаза.

4.1.3. Избиране на същност, идентифицираща потребителя

4.1.3.1. Проблеми

След успешна автентикация приложението би трябвало да може да определи кой е потребителя, който използва приложението. Това би трябвало да може да стане като се използват стандартните методи от Servlet спецификацията HttpServletRequest.getRemoteUser() и HttpServletRequest.getUserPrincipal(). Това, обаче, се оказва не особено тривиална задача. Както знаем, логин модулите са тези, които идентифицират потребителя, който иска да влезе в приложението. Освен това в един стек можем да имаме повече от един логин модула и всеки един от тях да идентифицира потребителя чрез различна същност или набор от същности.

4.1.3.2. Решение

За да решим горния проблем, се налага да измислим правило, по което да се определя една от същностите като основна същност, идентифицираща потребителя. Това може да се осъществи, като изработим същност, която представлява клас от специален тип и да фиксираме, че тази същност идентифицира потребителя. Когато се налага да определим коя от многото същности ще се върне като резултат от извикването на метода getUserPrincipal(), можем да вземем първата същност от

субекта, която е от този специален тип.

4.1.4. Осигуряване на гъвкавост на механизмите за автентикация при стандартни J2EE приложения.

4.1.4.1. Проблеми

Servlet спецификацията предвижда четири основни типа автентикация, които да могат да се указват чрез файла web.xml за отделните уеб модули. Това са BASIC, FORM, CLIENT-CERT и DIGEST. Наред с това, обаче, искаме тези механизми да могат да се заменят със други, които осигуряват същата функционалност, но адаптирана за нуждите на конкретната система. Например, при автентикация със сертификат, стандартно сървърът може да идентифицира потребителя, като взима потребителското му име от някой от атрибутите на сертификата. При някои системи, обаче, може да има изискване потребителското име да се взима от друг атрибут на сертификата. В други случаи пък, може да има изискване наред със задължителната функционалност, осъществяваща съответния механизъм за автентикация, да се добави и допълнителен модул, който да извършва допълнителни проверки и действия.

4.1.4.2. Решение

За да бъдат изпълнени горепосочените изисквания, трябва съответните механизми за автентикация (BASIC, FORM, CLIENT-CERT и DIGEST) да бъдат реализирани под формата на логин модули. Стандартната конфигурация на средата за автентикация трябва да съдържа стекове от логин модули, чиито имена да съответстват на съответните типове автентикация. Стековете от своя страна, трябва да съдържат логин модули, осъществяващи съответните механизми. При опит за навлизане в защитена област от приложение, то средата за автентикация трябва да инициализира логин контекст с име, съответстващо на типа автентикация, декларирана в съответния Уеб модул.

4.1.5. Изискване на име и парола от потребителя в зависимост от типа автентикация.

4.1.5.1. Проблем

При наличие на възможност за гъвкаво подменяне на механизмите за автентикация, сървърът не може предварително да знае какви точно данни ще са необходими за автентикация. Това е известно само на логин модулите. Затова, изискването на съответните потребителски данни от клиента може да стане само след

тяхното поискване от страна на логин модулите. При това начинът, по който се изискват потребителското име и парола, трябва да е съобразен с типа автентикация (BASIC или FORM) деклариран за съответния Уеб модул.

4.1.5.2. Решение

Според JAAS спецификацията данните за потребителя се изискват от логин модулите чрез изпращане на съобщения към обработчика на съобщения. Обработчикът на съобщения, от своя страна, трябва да поиска от клиента съответната информация. Тъй като разработваната среда за автентикация има за цел да поддържа само уеб приложения, то ни е нужен само един обработчик на съобщения, който да работи с HTTP заявки и отговори. Този обработчик на съобщения би трябвало да може да изиска име и парола от клиента по двата начина, описани в Servlet спецификацията – BASIC и FORM. Информацията за типа автентикация на съответното приложение трябва да му се подава в конструктора по време на инициализация.

4.2. Използвани технически средства

Средата за автентикация е реализирана под формата на уеб приложение, за да може лесно да се внедри в сървъра.

При проектиране на продукта, като и за онагледяване на начина, по който функционира, са използвани UML диаграми, които са разработени върху Microsoft Visio.

Средата за автентикация е осъществена с използването на език за програмиране Java, версия 5.0 [4], [5], [6], която е най-новата финализирана версия до момента.

Реализирането на приложението е направено с помощта на средата за програмиране Eclipse 3.2 [16].

Тя е разработена върху SAP NetWeaver Application Server 7.10 [17]. Спряла съм се на този сървър, защото той е първият сървър, след този на Sun, който е сертифициран за поддръжка на Java 5. Освен това, това е сървърът, с който съм работила най-продължително време и познавам най-добре.

SAP NetWeaver Application Server сам по себе си също поддържа JAAS. Предлаганата разработка представлява алтернативна среда за автентикация, базирана върху JAAS.

4.3. Реализация

Тази разработка представлява среда за автентикация на уеб приложения. След внедряването и конфигурирането на продукта, той може да се използва за автентикация на уеб приложения. Средата може да се използва както за програмна автентикация, с изричното създаване и викане на методите на стандартния JAAS логин контекст, така и за декларативна автентикация, с обявяване на защитена област (protected area) и тип на автентикацията (authentication type). Тази разработка може да се използва като модел за реализиране на автентикация с JAAS от сървърите за приложения.

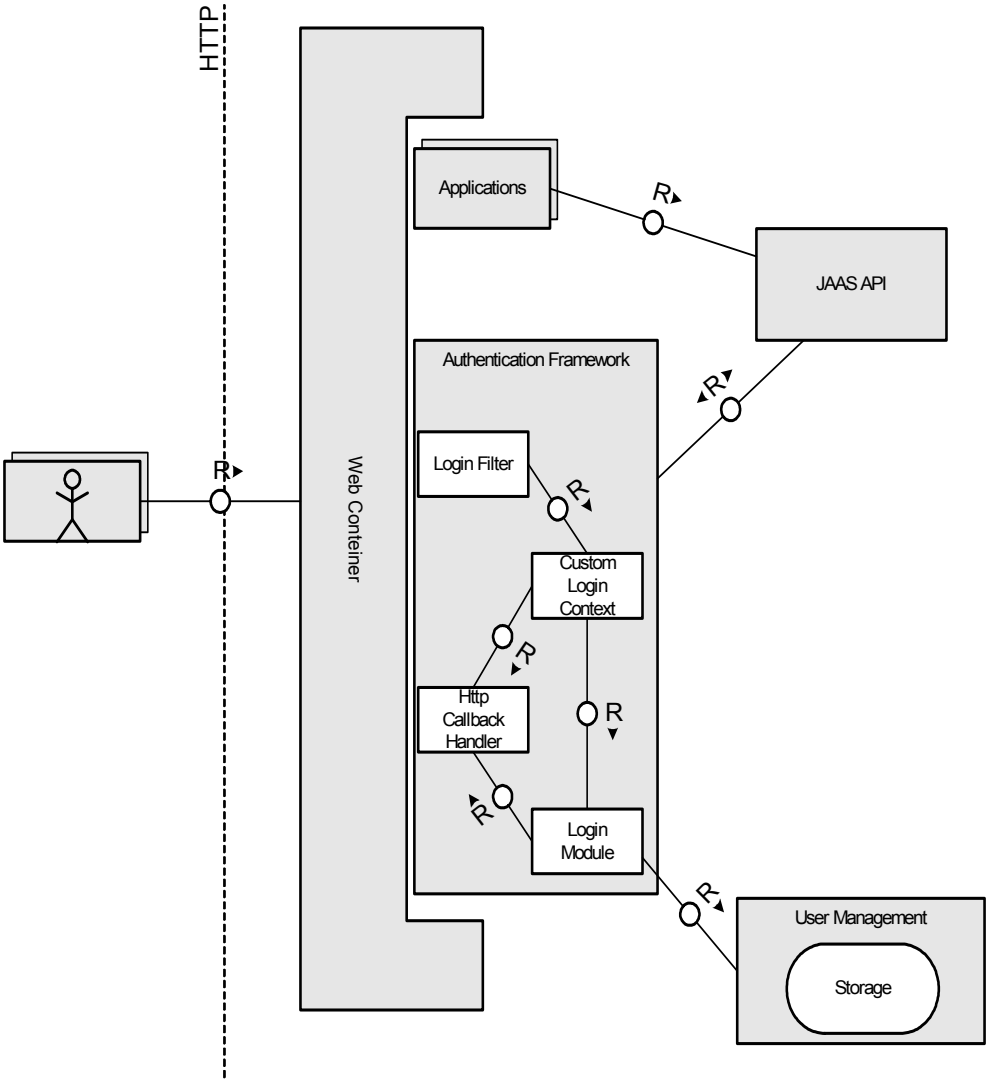
4.3.1. Компоненти

Средата за автентикация е реализирана чрез уеб приложение AuthenticationFramework.ear, състоящо се от един уеб модул login.war. Уеб модулът съдържа следните класове, групирани в няколко пакета:

- Логин контекст и помощни класове – пакет authentication.context: CustomLoginContext , AuthStackProcessAction, LoginFilter, ForwardLoginModule
- Логин модули – пакет authentication.module: NameAndPasswordLoginModule, ClientCertificateLoginModule, EmailLoginModule, CreateSSOTokenLoginModule, EvaluateSSOTokenLoginModule
- Съобщения и обработчик на съобщения – пакет authentication.callback: HttpCallbackHandler, EmailCallback, CertificateCallback, SSOTokenGetterCallback, SSOTokenSetterCallback, AuthenticationStatusCallback
- Конфигурация – пакет authentication.configuration: LoginModuleConfiguration, AppConfigurationEntryExtension, ControlFlag
- Същности – пакет authentication.principal: UserIdPrincipal, NameAndPasswordPrincipal, EmailPrincipal, CertificatePrincipal, SSOTokenPrincipal
- Документи за самоличност – пакет authentication.credential: PasswordCredential, CertificateCredential, SSOTokenCredential

Освен това, уеб модулът притежава и файл index.jsp, който съдържа връзки към тестовите приложения, чрез които може да се извърши тестването на средата.

Средата за автентикация използва и конфигурационен файл LoginModuleConfiguration.txt, който трябва да се постави в работната директория на сървъра.



Фиг. 9 Взаимодействия между основните компоненти на собствената среда за автентикация.

4.3.2. Конфигурация и логин контекст

В основата на реализацията на средата за автентикация стои класът LoginModuleConfiguration (приложение 9.3), който наследява javax.security.auth.login.Configuration. Той не връща действителната конфигурация от

логин модули, а връща винаги масив, съдържащ един логин модул - ForwardLoginModule. Името на автентициращия стек се подава като опция с име “login.modules.stack.name” на логин модула.

По време на инициализацията си, ForwardLoginModule (приложение 9.3) създава обект от тип CustomLoginContext. Като име на логин контекста се подава стойността на опцията “login.modules.stack.name”. Също така параметрите subject и callbackHandler, подадени на ForwardLoginModule от стандартния JAAS логин контекст се прехвърлят като параметри на създадения CustomLoginContext.

По този начин, дадено приложение може да направи програмна автентикация, като създаде екземпляр на стандартния логин контекст с параметри, описани в JAAS спецификацията – име на конфигурация, субект, обработчик на съобщения. Тези параметри ще се подадат на нашия CustomLoginContext, който ще инициализира логин модулите, специфицирани в конфигурацията със съответното име. Когато приложението извика метода login() на стандартния логин контекст, то ForwardLoginModule, от своя страна, ще предаде управлението на CustomLoginContext като извика неговия метод login(). CustomLoginContext пък ще извика методите login(), а след това и методите commit() или abort(), последователно за всеки от инициализираните логин модули. По аналогичен начин се осъществява и излизането (logout) от приложението.

Предлаганата реализация на логин контекст взима логин модулите, които са конфигурирани за съответния стек, чието име е подадено като първи параметър на конструктора на CustomLoginContext. Стековете от логин модули се конфигурират във файл с име LoginModuleConfiguration.txt. Той се намира в работната директория на сървъра - SAP\<SID>\JC00\j2ee\cluster\server0. Нашият логин контекст прочита конфигурацията от там и си инициализира масив от логин модули, съответстващи на тази конфигурация. CustomLoginContext последователно зарежда и създава обекти за всеки един от логин модулите. След това, извиква метода login() на всеки логин модул от стека, като се съобразява с флаговете им и със върнатия резултат. В зависимост от резултата от изпълнението на логин фазата, той извиква commit() или abort() последователно на всички логин модули. В случай, че commit фазата не успее, също се извиква abort() на всички логин модули. Автентикацията успява и CustomLoginContext връща резултат истина, ако фазите login и commit завършат успешно.

Операциите за login, commit, abort и logout се изпълняват в привилегировано действие (privileged action). Това се прави по аналогия на реализацията на стандартния

JAAS логин контекст. Причината за това е, че на логин модулите може да са им необходими специални права за извършване на някои операции като например актуализация на субекта. Тези права би трябвало да се конфигурират за съответните логин модули, както и за всички компоненти, които правят програмна автентикация. Тъй като това би затруднило ненужно конфигурирането на тези права, то логин контекстът извиква логин модулите чрез привилегировано действие. По този начин, съответните методи за проверка на правата, отчитат само правата, които са дадени на съответния логин модул и на логин контекста.

В тази разработка е реализиран клас AuthStackProcessAction (приложение 9.6), който наследява java.security.PrivilegedExceptionAction. Той обхожда логин модулите, като изпълнява съответната операция (login, commit, abort или logout) върху тях. Решението за това дали да продължи със следващия логин модул се основава на контролния флаг и върнатия резултат на текущия логин модул.

След изпълняването на стека от логин модули, CustomLoginContext изпраща съобщение (callback) на обработчика на съобщения (callback handler), съдържащ статуса на автентикацията – дали е успешна или неуспешна. Тази информация се използва по-късно от обработчика на съобщения, за да генерира подходящ отговор (response) към клиента.

При успешна автентикация се асоциира субекта (заедно с всички негови същности (principals) и лични документи (credentials) добавени от логин модулите) със сесията на приложението. Тази стъпка се извършва чрез специфична за SAP Web Application Sever функционалност. Така се уведомява уеб контейнера за извършената автентикация. По този начин, текущата заявка, както и всяка следваща заявка към същото приложение и от същия клиент, се извършват от името на вече автентицирания потребител. Резултатът може да се види като се извикат методите HttpServletRequest.getRemoteUser() и HttpServletRequest.getUserPrincipal() от съответното приложение.

4.3.3. Същности

При горната стъпка, освен субекта като цяло, за HTTP сесията се асоциира и същност, която идентифицира потребителя. Тъй като в субекта може да има повече от една същности, то логин контекстът избира една от тях по следния алгоритъм (приложение 9.5):

- Избира се първата същност, която се върне от итератора на множеството от

същности, съдържащи се в субекта, която е от тип UserIdPrincipal.

- Ако няма такава същност, тогава се избира първата от множеството от същности, която е от произволен тип.

Този алгоритъм е много подобен на алгоритъма, който използва BEA WebLogic, с тази разлика, че там може да има и същности, които идентифицират потребителска група и затова същността, която се избира, трябва да не представлява група.

Средата за автентикация предоставя няколко вида същности, всеки от които съответства на определен тип автентикация: NameAndPasswordPrincipal, EmailPrincipal, CertificatePrincipal, SSOTokenPrincipal. Те всички наследяват UserIdPrincipal (приложение 9.9) и съдържат името на идентифицирания потребител. По този начин едновременно постигаме две цели:

- Независимо какъв механизъм за автентикация използваме, добавяме в субекта същност, която идентифицира потребителя и може да се използва от контейнерите за проверки на правата.

- Приложението може да разбере с какъв механизъм е направена автентикацията по типа на същността, намираща се в субекта и да предприеме съответните действия. Например, определена функционалност може да е достъпна от потребителя само ако се е влязъл в системата с подаване на клиентски сертификат.

4.3.4. Логин филтър

Декларативната автентикация се осъществява от класа LoginFilter (приложение 9.7). Той представлява реализация на javax.servlet.Filter [6], [8]. За да се интегрира с уеб контейнера, трябва да се конфигурира в global-web.xml под името AuthenticationFilter:

```
<filter>
  <filter-name>AuthenticationFilter</filter-name>
  <filter-class>LoginFilter</filter-class>
</filter>
```

Тази конфигурация, разбира се, е специфична за САП. Тя ни осигурява, че класът LoginFilter ще бъде извикан от уеб контейнера в случаите, когато трябва да се направи автентикация според Servlet спецификацията [7]. Тези случаи се свеждат до два:

- Когато потребителят иска да достъпи ресурс, който се намира в защитена област.
- Когато потребителската заявка е към URL, завършващо на “j_security_check”.

Основното, което прави класът LoginFilter, е да извърши автентикация чрез CustomLoginContext. Името на стека, което се подава на логин контекста, се определя от типа автентикация, зададен във файла web.xml. Филтърът взима тази конфигурация чрез подадения като параметър екземпляр на FilterConfig при инициализацията му. Според Servlet спецификацията [7], сървърите трябва да поддържат задължително BASIC, FORM и CLIENT-CERT (или CLIENT_CERT) типове автентикация и по желание – DIGEST. Стандартно след инсталирането и конфигурирането на средата за автентикация, в конфигурационния файл “LoginModuleConfiguration.txt” се съдържат няколко конфигурации на стекове от логин модули:

```
basic {
  NameAndPasswordLoginModule SUFFICIENT;
};
form {
  NameAndPasswordLoginModule SUFFICIENT;
};
client-cert {
  ClentCertificateLoginModule SUFFICIENT;
};
other {
  NameAndPasswordLoginModule SUFFICIENT;
};
```

По този начин, автентикацията на приложението ще се изпълни съгласно Servlet спецификацията [7]. Ако, например, приложението е с basic тип автентикация, ще се изпълни NameAndPasswordLoginModule, който прави логин с потребителско име и парола. Разбира се, администраторът на системата има възможност да замени така конфигурираните логин модули с други механизми за автентикация или да добави нови към стека. По този начин, уеб контейнерът все още поддържа изискванията на Servlet спецификацията [7], като едновременно с това е гъвкав за добавяне на нови механизми и технологии за автентикация.

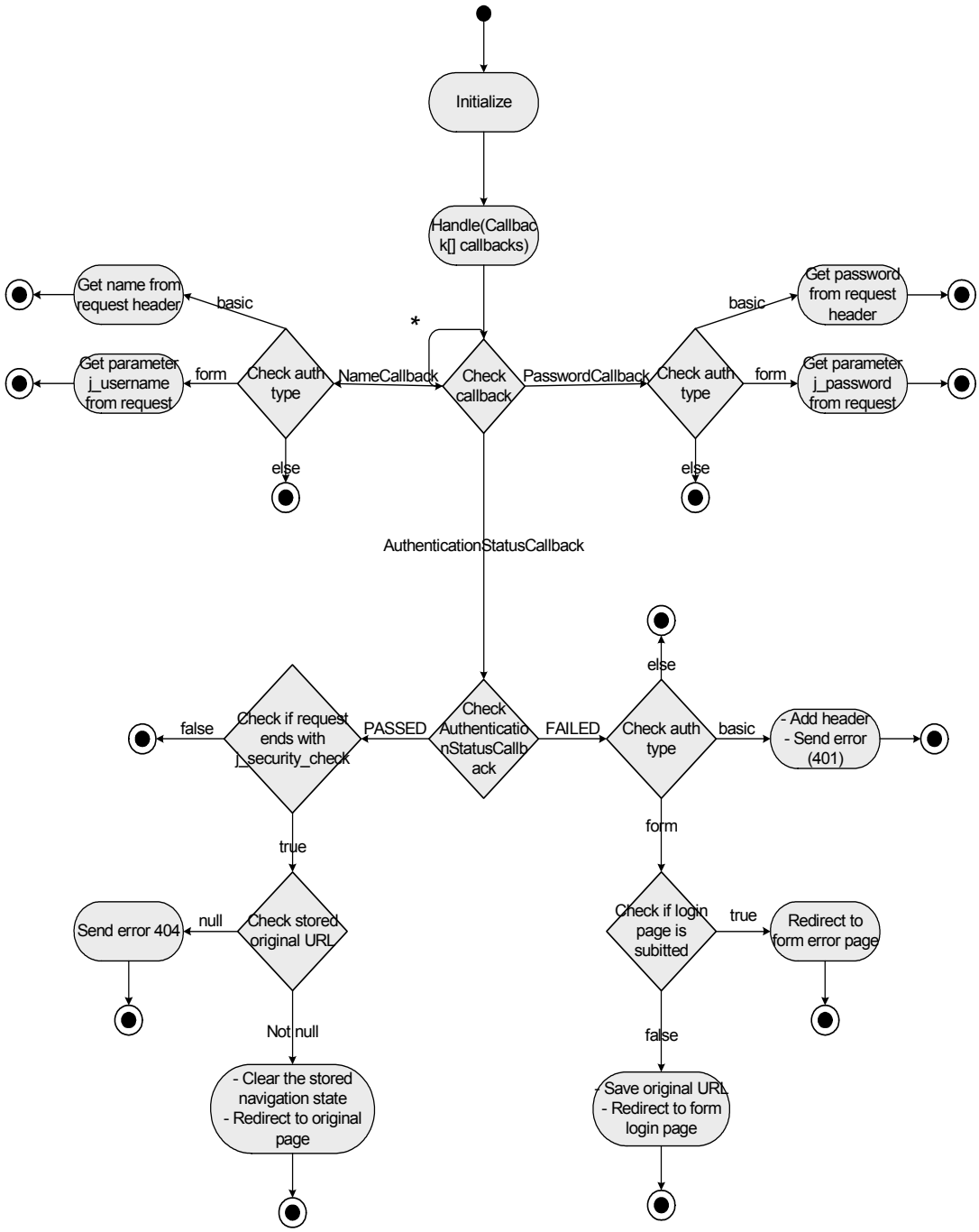
4.3.5. Съобщения и обработчик на съобщения

За да извърши автентикация, логин филтърът създава екземпляр на

HttpCallbackHandler, който подава като параметър на логин контекста. Обработчикът на съобщения се инициализира с HttpServletRequest, HttpServletResponse и WebApplicationConfig. Тези параметри са му нужни, за да може да обработва съобщенията, получени от логин модулите и да им предоставя необходимите данни.

HttpCallbackHandler може да обработва следните видове съобщения: стандартните NameCallback и PasswordCallback и специфични за тази среда за автентикация – CertificateCallback, EmailCallback, SSOTokenGetterCallback, SSOTokenSetterCallback, AuthenticationStatusCallback. NameCallback и PasswordCallback се използват от NameAndPasswordLoginModule за извличане на потребителското име и парола от заявката (request). EmailCallback се използва от EmailLoginModule за поискване на адреса на електронната поща от потребителя. SSOTokenSetterCallback и SSOTokenGetterCallback се използват съответно от CreateSSOTokenLoginModule и EvaluateSSOTokenLoginModule за подаване на бисквитка с име “SSOToken” към клиента и за проверка на получена такава бисквитка от клиентската заявка.

AuthenticationStatusCallback (приложение 9.8) представлява изброен списък (enumeration) [4], [5] и има две възможни стойности – FAILED и PASSED. Чрез тях логин контекстът съобщава на обработчика на съобщения как е завършила автентикацията, за да може да генерира подходящ отговор към клиента. Една от основните задачи на обработчика на съобщения е да поиска от клиента необходимите данни за автентикация, които са били поискани от логин модулите. Например, ако някой от логин модулите е поискал потребителско име и парола, то обработчикът на съобщения трябва да изведе съответното подканящо съобщение към клиента. Тъй като HttpCallbackHandler работи само и единствено с уеб приложения, то той би могъл да поиска потребителско име и парола от клиента основно по два начина, в зависимост от декларирания тип на автентикация във файла web.xml на уеб модула – BASIC или FORM. Тези два типа автентикация са подробно описани в Servlet спецификацията [7].



Фиг. 10 Диаграма на поведението на обработчика на въпроси.

При декларативна автентикация, информация за конфигурацията, съдържаща се във файла web.xml, обработчикът на съобщения получава от WebApplicationConfig, който му се подава по време на инициализация. При програмна автентикация, обаче,

процесът на автентикация не се контролира от декларираните във web.xml етикети. Въпреки това, средата за автентикация предоставя и в този случай възможност на приложенията да укажат по какъв начин да се изискат името и паролата от клиента. Както вече споменах по-горе, при програмна автентикация, приложенията създават екземпляр на логин контекст, като един от параметрите които подават е обработчик на съобщения (фиг. 10 и 13). При инициализирането на HTTPCallbackHandler, приложението може да подаде желания тип автентикация като параметър. В случай че желаният тип автентикация е FORM, то приложението може да укаже програмно, също и кои са логин страницата му и страницата за грешки, като ги подаде като параметри на обработчика на съобщения.

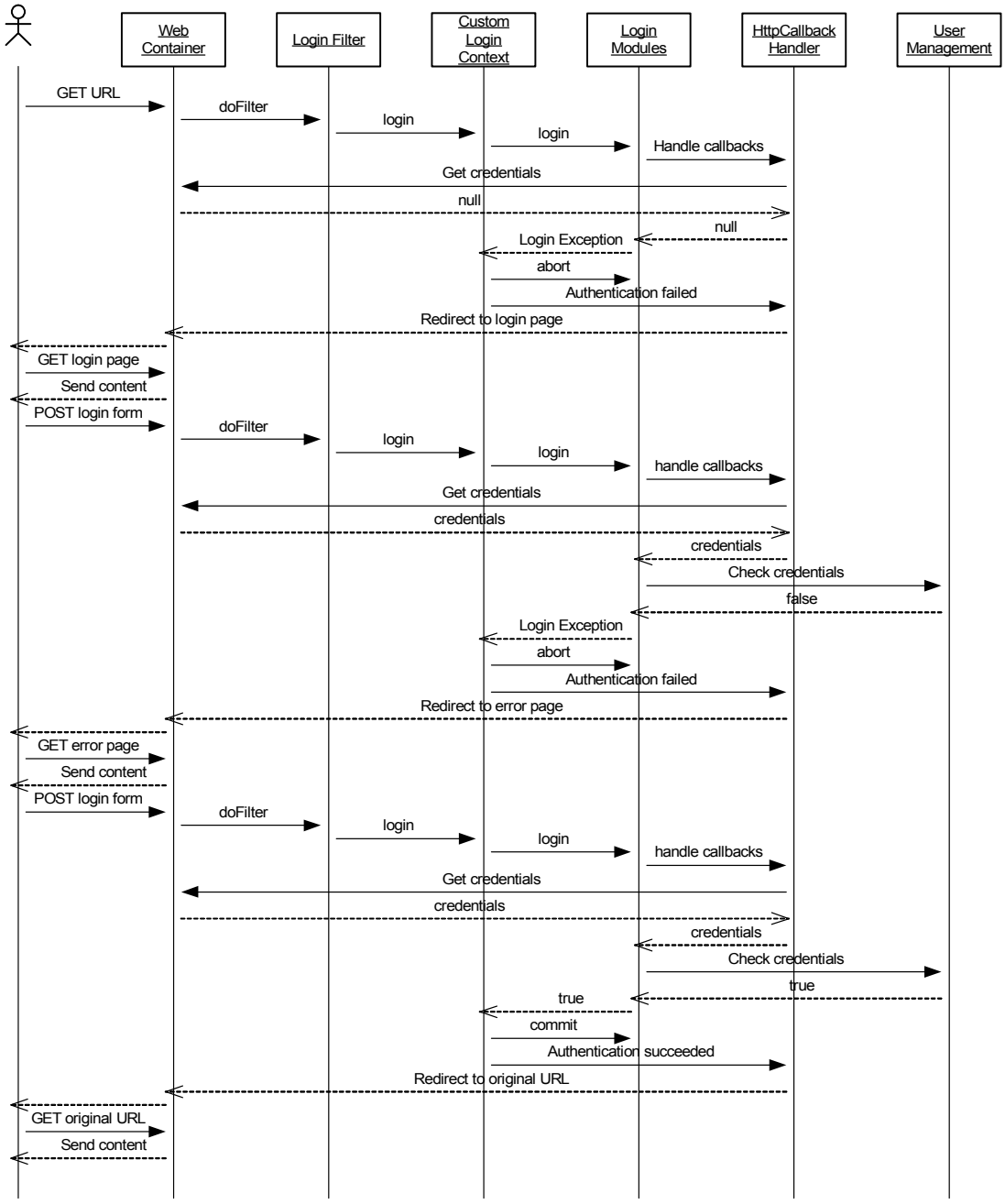
Важно е да се отбележи, че обработчикът на съобщения изисква име и парола по един от двата гореспоменати метода, но само ако те са му били поискани от логин модулите. Ако администраторът на системата е решил да конфигурира за “basic” стека някакъв логин модул, който не работи с потребителско име и парола, то тогава обработчикът на съобщения няма да поиска от клиента тези данни, независимо че декларираният тип на автентикацията е BASIC. В този случай няма смисъл потребителят да се подканя да въвежда име и парола, при положение че няма логин модул, който да обработи тези данни.

4.3.5.1. Поведение на обработчика на съобщения при FORM логин

При FORM тип автентикация (фиг. 11 и 12), обработчикът на съобщения поисква потребителско име и парола, като препраща клиента към логин страница, където той може да въведе тези данни. Този способ се използва от приложения, които искат да контролират начина, по който изглежда формулярът за попълване на данните.

Преди HttpCallbackHandler да препрати потребителя към логин страницата, той си записва текущото URL в HTTP сесията, за да може след успешна автентикация да го препрати пак към този адрес. Когато потребителят получи логин страницата, той въвежда име и парола. При изпращане на попълнения формуляр, се генерира заявка към сървъра, която е с URL, завършващ на “j_security_check”. Това кара уеб контейнера да извика логин филтъра за да направи автентикация. Логин филтърът взима името на стека от логин модули и го подава на логин контекста, с който извършва автентикация. По време на автентикация обработчикът на съобщения предоставя на логин модулите поисканите данни. При получаване на NameCallback и PasswordCallback, например, HttpCallbackHandler взима името и паролата съответно от параметрите на заявката с имена “j_username” и “j_password”. Логин контекстът съобщава на обработчика на съобщения как е завършила автентикацията чрез

AuthenticationStatusCallback. В зависимост от този резултат, обработчикът на съобщения препраща потребителя към страница съдържаща съобщение за грешка, или към първоначално поисканата страница.

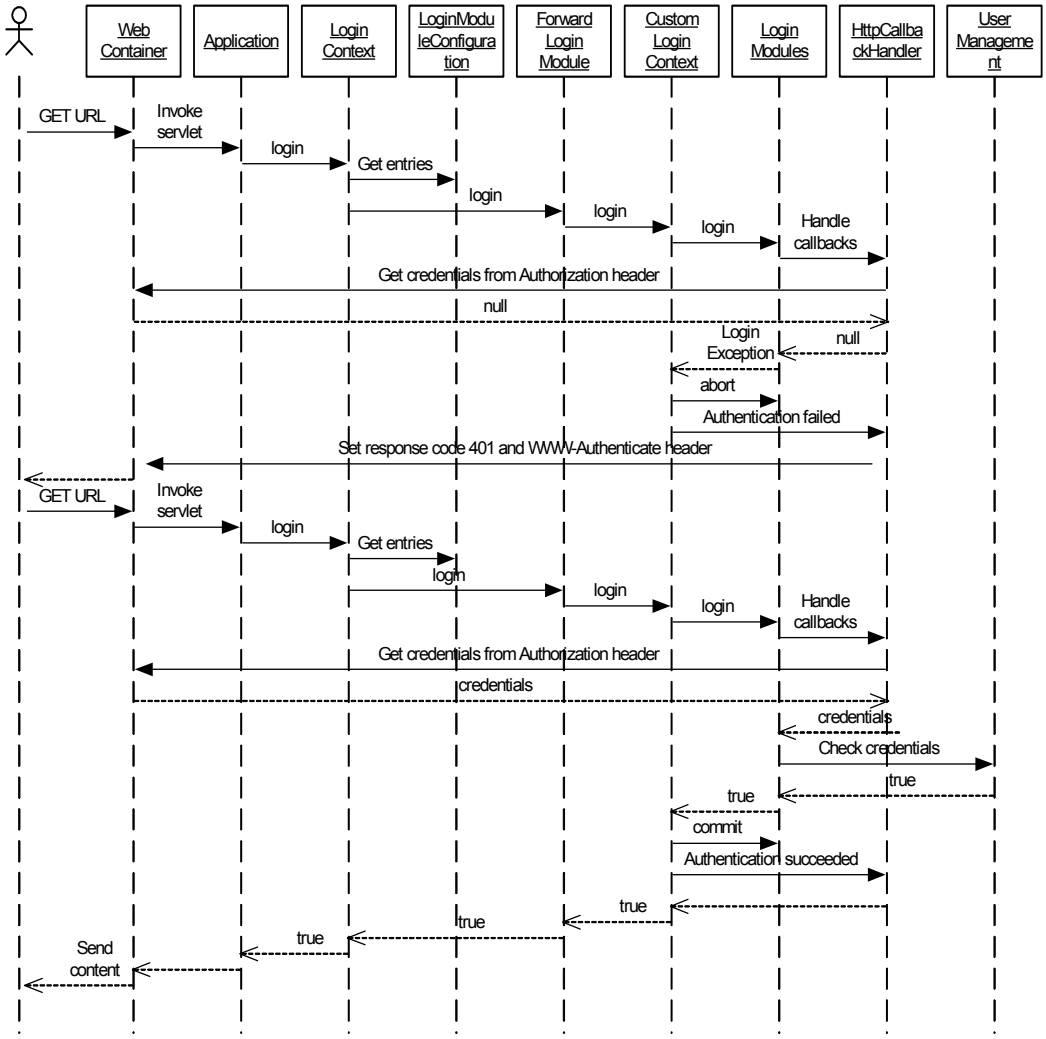


Фиг. 11 Автентикация към уеб приложение с декларативна автентикация, FORM тип автентикация и NameAndPasswordLoginModule.

4.3.5.2. Поведение на обработчика на съобщения при BASIC логин

Когато приложението е с BASIC тип автентикация (фиг. 11 и 13), обработчикът на съобщения да поисква потребителско име и парола от клиента е като изпрати отговор с код 401 (HtthServletResponse.SC_UNAUTHORIZED) и добави заглавен атрибут (header) с име “WWW-Authenticate” и стойност, съдържаща типа на автентикацията и името на областта (realm), в която се извършва тя. Този отговор на сървъра кара навигатора (browser) да покаже на потребителя формуляр с две полета за въвеждане на име и парола.

Ако автентикацията се осъществява с BASIC метод и някой от логин модулите поиска потребителско име и парола, то тогава обработчикът на съобщения взима тези данни от заглавния атрибут “Authorization”. Ако автентикацията не успее, то обработчикът на съобщения изисква (първоначално или поредно) въвеждане на име и парола от страна на потребителя. При успешна автентикация обработчикът на съобщения не прави нищо, тъй като повторната заявка съдържаща въведените данни от клиента е към същото URL, както при първата заявка. Тъй като автентикацията минава успешно, то уеб контейнерът предоставя изискания ресурс.



Фиг. 12 Автентикация към уеб приложение с програмна автентикация, BASIC тип автентикация и NameAndPasswordLoginModule.

4.3.6. Логин модули

При автентикация с потребителско име и парола се използва NameAndPasswordLoginModule [2]. В login() метода си той поисква съответните данни от обработчика на съобщения чрез NameCallback и PasswordCallback. NameAndPasswordLoginModule проверява върнатите стойности – дали съществува такава потребителска регистрация с такова име и дали съответната парола е вярна. Това става чрез методи, които са специфични за САП. При успешно изпълнение на login метода и при успешна цялостна автентикация, методът commit() създава

NameAndPasswordPrincipal за съответното потребителско име и PasswordCredential, където се слага потребителската парола. След това, същността и документът за самоличност на потребителя се асоциират със субекта. Ако собствената автентикация на логин модула е била успешна, то при излизане (logout), същността и документът за самоличност се махат от субекта.

Средата за автентикация предоставя възможност вместо име потребителят да въведе адреса на електронната си поща. Този механизъм се осъществява чрез EmailLoginModule. Този логин модул е подобен на NameAndPasswordPrincipal. Той взима адреса на електронната поща също чрез NameCallback и след това проверява дали съществува потребител с такива данни. При успешна автентикация добавя в субекта EmailPrincipal, съдържащ името на идентифицирания потребител за подадения адрес.

CertificateLoginModule осъществява автентикация с клиентски сертификат. Той използва CertificateCallback, за да поиска сертификата от обработчика на съобщения. Той, от своя страна, взима сертификата от атрибута “javax.servlet.request.X509Certificate” на потребителската заявка. След това проверява дали съществува потребителска регистрация с такъв сертификат. При успешна автентикация, създава CertificatePrincipal, съдържащ името на идентифицирания потребител, и го добавя към субекта.

За разлика от другите логин модули, CreateSSOTokenLoginModule не прави нищо в метода си login(). В метода commit() (приложение 9.10) той проверява дали някой от предходните логин модули е автентицирал успешно потребителя. Това се прави, като проверява дали в субекта има същност от тип UserIdPrincipal. Ако намери такава същност, то той създава бисквитка с име “SSOToken”, която съдържа името на идентифицирания потребител. След това, той я добавя към отговора към клиента с помощта на SSOTokenSetterCallback, който подава на обработчика на съобщения. Освен това, той създава същност от тип SSOTokenPrincipal и документ за самоличност от тип SSOTokenCredential и ги добавя към субекта. Създадената същност съдържа името на потребителя, а документът за самоличност – стойността на бисквитката.

EvaluateSSOTokenLoginModule извършва автентикация като взима обекта за SSO чрез SSOTokenGetterCallback и извлича потребителското име от него. След това проверява дали съществува потребителска регистрация с такова име. Ако автентикацията е успешна, подобно на CreateSSOTokenLoginModule той създава същност от тип SSOTokenPrincipal и документ за самоличност от тип

SSOTokenCredential и ги добавя към субекта. Както при CreateSSOTokenLoginModule и тук създадената същност съдържа името на потребителя, а документът за самоличност – стойността на бисквитката.

По този начин, като се конфигурират две приложения съответно с CreateSSOTokenLoginModule и EvaluateSSOTokenLoginModule би могло да се осъществи Single Sign-On между тях. Обектът за автоматична автентикация, който създава CreateSSOTokenLoginModule, не е особено надежден метод за установяване на самоличността на потребителя, защото съдържа само потребителското име, кодирано с Base64 алгоритъм и не е подписан. Този логин модул се използва в разработката за демонстрация на допълнителния контролен флаг, който се разглежда по-нататък.

4.3.7. Конфигуриране на стекове от логин модули

За да поддържа новия флаг CLOSING, средата за автентикация се нуждае от собствена реализация на класовете javax.security.auth.login.AppConfigurationEntry и javax.security.auth.login.AppConfigurationEntry.LoginModuleControlFlag. Това се налага тъй като конструкторът на класа AppConfigurationEntry приема като параметър за контролен флаг само обекти от тип AppConfigurationEntry.LoginModuleControlFlag, а LoginModuleControlFlag има само личен (private) конструктор.

Затова, в настоящата разработка, е създаден класа AppConfigurationEntryExtention (приложение 9.1), който наследява AppConfigurationEntry и приема параметри от тип ControlFlag по време на инициализацията си. Класът ControlFlag (приложение 9.2) е реализиран чрез изброен списък [4], [5], който има следния набор от стойности: REQUIRED, REQUISITE, SUFFICIENT, OPTIONAL, CLOSING. Това значително съкращава реализацията на класа която се събира на един ред. В допълнение на това, чрез метода ControlFlag.valueOf(String), който се предоставя стандартно от изброения списък, може лесно да се получи обект от тип ControlFlag за съответния низ (например при четене на конфигурацията от файла). Тъй като ControlFlag не може да наследи AppConfigurationEntry.LoginModuleControlFlag, то се налага да дефинираме нов метод в класа AppConfigurationEntryExtention за взимане на контролния флаг.

4.3.7.1. Приложение

Ето някои случаи, при които се налага използването на флаг CLOSING:

- Съществуват редица операции, които следва да се извършат, независимо кой от механизмите за логин е успял. Това са проверки за потребителя, които сървърът задължително трябва да направи, за да сме сигурни, че потребителят притежава валидна регистрация:

  - Проверка дали потребителската регистрация (user account) не е с изтекъл срок на валидност.

  - Проверка дали потребителската регистрация не е временно забранена от администратора на системата поради някаква причина (например неплатена сметка).

Тези проверки биха могли да се направят в системен логин модул, който да се постави в края на стека с флаг CLOSING.

Друга възможност за реализиране на тази функционалност е тези проверки да се направят и в метода login() на логин контекста (LoginContext) при успешно изпълнение на стека от логин модули. Ако се окаже, че потребителската регистрация е наред, тогава логин контекстът ще върне резултат истина. Ако се окаже, обаче, че потребителската регистрация не е валидна, тогава методът login() на логин контекста ще трябва да хвърли изключение. При това положение, се получава противоречие със спецификацията на логин контекста, според която, ако стекът от логин модули се е изпълнил успешно, то логин контекстът трябва да върне резултат истина.

Друг вариант за реализиране на тази функционалност, е да се задължи всеки създател на логин модули да добавя тези проверки в своя логин модул. Това от една страна би затруднило хората, които пишат логин модули, а от друга страна ще създаде условия за непоследователност в поведението на сървъра – ако не всички логин модули са направили тези проверки правилно, то в зависимост от механизма на автентикация, сървърът ще позволява или не влизането на потребители с невалидна регистрация.

- Друг случай на употреба на новия флаг е, когато искаме да реализираме Single Sign-On (SSO). Това е механизъм, при който при успешна автентикация към дадено приложение, се генерира някакъв обект, който да служи за разпознаване и автентикация на потребителя при следващи негови опити за логин към други приложения. По този начин, потребителят въвежда ръчно своята идентифицираща информация само при първата автентикация към системата. При следващи заявки към други приложения може да се използва генерирания

обект и потребителят автоматично да се автентицира. За да реализираме този механизъм, трябва добавим логин модула, който създава SSO обекта в края на стека от логин модули и да му сложим флаг CLOSING. Приложенията, които искаме да използват SSO, трябва да конфигурираме с логин модул, който разпознава и проверява SSO обекта. Ето и примерни конфигурации на стекове от логин модули за приложения използващи Single Sign-On:

```
login-module-stack-name {
  ClientCertificateLoginModule SUFFICIENT;
  NameAndPasswordLoginModule SUFFICIENT;
  CreateSSOTokenLoginModule CLOSING;
};
```

В горните два случая бихме могли да използваме таблицата със споделени състояния, която се подава на логин модулите по време на инициализация. Чрез тази таблица, логин модулите биха могли да си комуникират и всеки един от тях да реши дали да изпълни предназначението си или просто да върне резултат “лъжа”, в зависимост от състоянието на предхождащите го логин модули. При това решение се забелязват два основни недостатъка:

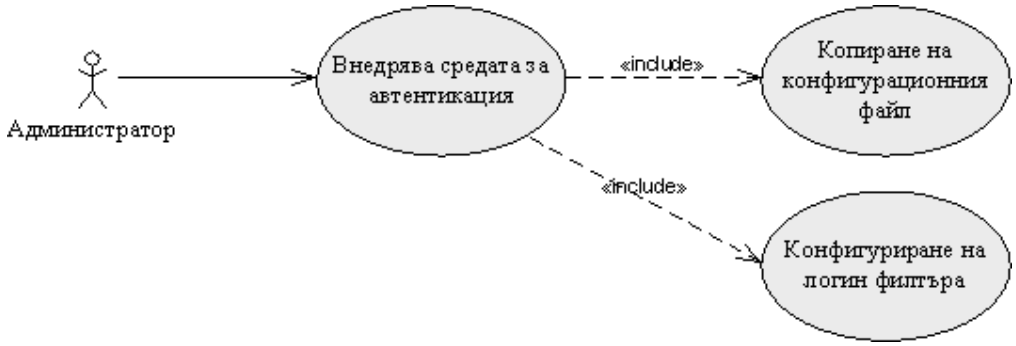
- Първо, трябва всички логин модули в стека, които си комуникират по този начин да познават протокола (правилата) на комуникация.
- Второ, администраторът на системата трябва да познава в детайли поведението на логин модулите, за да може да ги конфигурира правилно в стека.

- Пример за нужда от CLOSING логин модул в стека имаме и при JBoss. Те използват логин модули както за автентикация, така и за проверка на потребителските роли (authorization). В този случай, логин модулът, който проверява ролите (RoleCheckLoginModule), трябва да се постави в края на стека и да се изпълни само и единствено след успешна автентикация, дори и в стека да има SUFFICIENT механизъм за автентикация.

4.4. Случаи на употреба

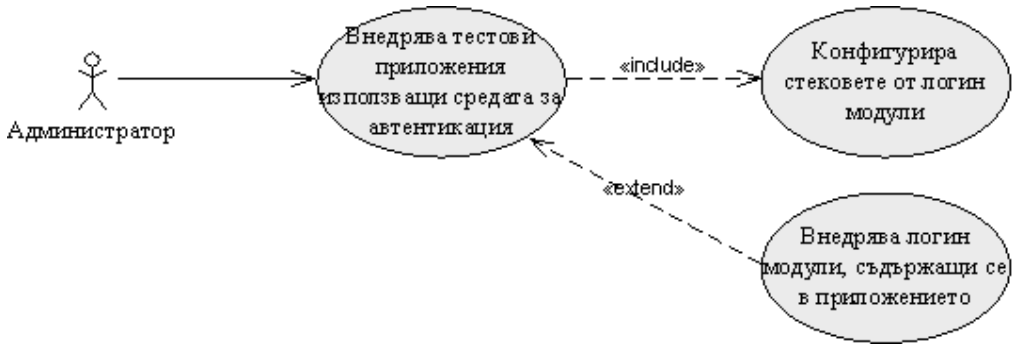
Така изработената среда за автентикация има три основни типа потребители – администратор, програмист и краен потребител.

Администраторът на системата е човека, който внедрява и конфигурира средата за автентикация (фиг. 14). При това, той трябва да копира конфигурационния файл, съдържащ стековете от логин модули в работната директория на сървъра и да конфигурира логин филтъра в global-web-j2ee-engine.xml.



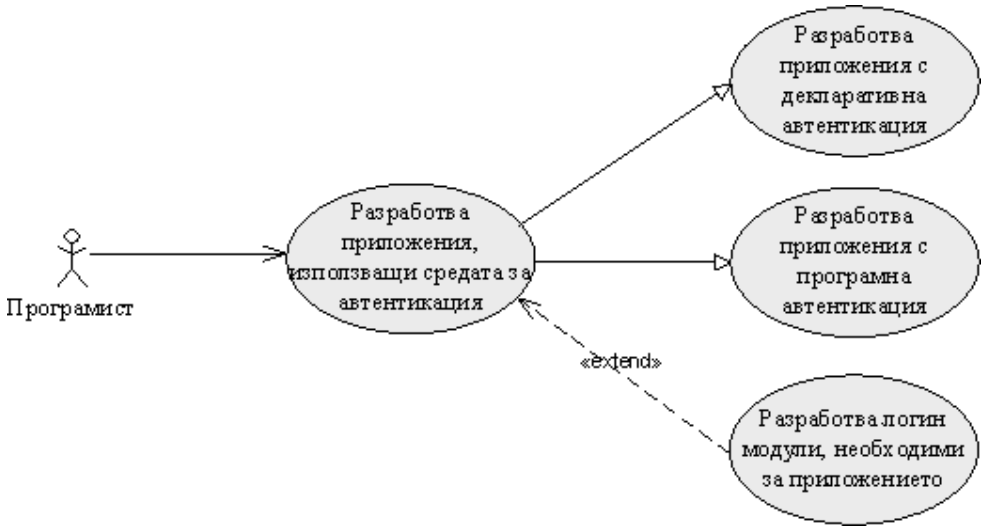
Фиг. 13 Администраторът внедрява средата за автентикация.

Другото основно задължение на администратора е да внедрява приложения, които от своя страна използват средата за автентикация (фиг. 15). Ако приложението е с програмна автентикация, може да се наложи да се добави съответния стек от логин модули в конфигурационния файл. Освен това, приложението може да съдържа собствени логин модули, които да бъдат използвани от него.



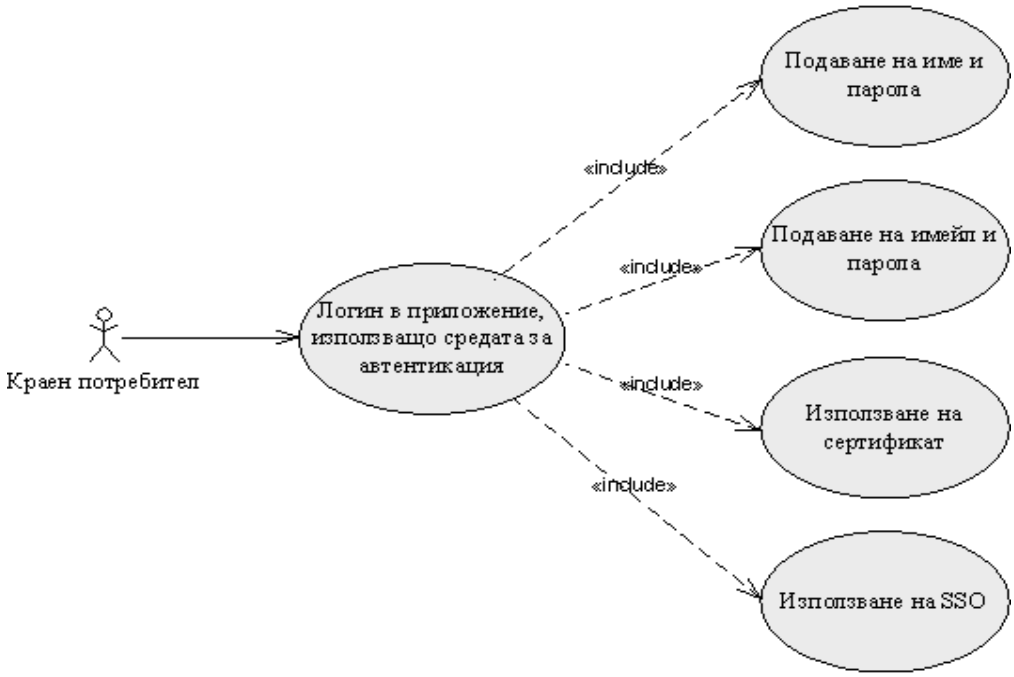
Фиг. 14 Администраторът внедрява приложения, използващи средата за автентикация.

Средата за автентикация може да бъде използвана и от програмисти, които да създават приложения, използващи средата за автентикация (фиг. 16). Тези приложения могат да използват както декларативна така и програмна автентикация. Освен това, програмистът би могъл да добави и собствени логин модули, които да бъдат използвани от приложението за автентикация.



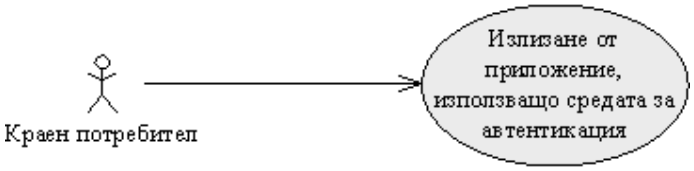
Фиг. 15 Програмистът разработва приложение, използващо средата за автентикация.

Крайният потребител е човека, използващ услугите и приложенията, които се предлагат от съответния сървър. За да направи това, той трябва първо да се автентицира, като подаде своите лични данни и документи, поискани от сървъра (фиг. 17).



Фиг. 16 Крайният потребител се автентицира към приложението.

След като потребителят приключи работата си, той излиза от приложението.



Фиг. 17 Излизане от приложението.

5. Тестване

5.1. Инсталиране и конфигуриране

За да се инсталира системата и за да се внедри и конфигурира средата за автентикация са нужни следните стъпки:

- Изтеглете SAP NetWeaver Application Server 7.10 от следния адрес: http://www.sdn.sap.com. Той може да се изтегли от всеки. Трябва само да си направите регистрация.

- Разархивирайте съдържанието на изтегления ZIP файл.

- Инсталирайте сървъра като стартирате файла setup.exe. По време на инсталация трябва само да въведете парола за администраторския потребител. Препоръчвам ви да въведете парола “abc123”, за да сте сигурни, че паролата е съобразена с политиката за съдържанието и дължината на паролите, която е конфигурирана първоначално.

- След успешна инсталация, на работния плот се появява връзка за стартиране на SAP Management Console. Там можете да видите кога сървърът е стартиран.

- След стартиране на сървъра, стартирайте telnet към него на порт 50008. (Start -> run -> telnet localhost 50008). Въведете име „Administrator” и избраната по време на инсталация парола.

- Внедрете приложението AuthenticationFramework.ear и тестовото приложение Test.ear по следния начин: В телнет конзолата напишете командата “add deploy”. След това изпълнете последователно командите:

  deploy <path_to_AuthenticationFramework.ear>/AuthenticationFramework.ear
  и
  deploy <path_to_Test.ear>/Test.ear.

- Копирайте LoginModuleConfiguration.txt в работната директория на сървъра - SAP\<SID>\JC00\j2ee\cluster\server0.

- Конфигурирайте филтъра LoginFilter в global-web.xml: Стартирайте offlinecfgeditor.bat, намиращ се в директория SAP\<SID>\JC00\j2ee\configtool.

Щракнете върху иконата с молива, за да преминете в режим на писане. Отворете global-web.xml, намиращ се в Configurations/cluster_config/system/custom_global/cfg/services/servlet_jsp/persistent. Изтрийте цялото му съдържание и копирайте съдържанието от предоставения global-web.xml.

- Създайте тестова потребителска регистрация:
  - Инсталирайте приложението за администрация – стартирайте install.bat, който се намира в директория SAP\<SID>\JC00\j2ee\NWAdmin.
  - В навигатора въведете адреса: http://localhost:50000/useradmin. Въведете име “Administrator” и паролата, която сте избрали по време на инсталация. От менюто в ляво изберете “Create User”. Попълнете задължителните полета във формуляра и натиснете бутона “Create”. След това отворете нов прозорец и се опитайте да влезете в същото приложение с този тестов потребител. Това се прави с цел при тази първа автентикация да се смени паролата на потребителя с някоя, която само той знае.

5.2. Тестови приложения

Тестовото приложение Test.ear се състои от пет уеб модула и един конфигурационен файл. Конфигурационният файл е application.xml. В него се описват всички съдържащи се уеб модули и съответния им контекстен корен (context root). Модулите, използвани в тестовото приложение могат да се разделят на 2 групи: “Модули с декларативна автентикация” и “Модули с програмна автентикация”.

5.2.1. Модули с декларативна автентикация

Тези модули съдържат по два файла - web.xml и authentication.jsp. Във файла web.xml е декларирана област на сигурност за всички ресурси от съответния модул. Това означава, че при опит за достъп на произволен ресурс от страна на клиента, уеб контейнерът ще инициира автентикация. Ако клиентът вече е направил успешна автентикация, то се проверява дали потребителят има права за този ресурс. Във файла web.xml на всеки модул е декларирано, че се допускат потребители, притежаващи всякакви роли. Разликата при тези модули е само в използвания тип автентикация:

- test_basic - използва BASIC тип автентикация.

- test_form - използва FORM тип автентикация.

- test_cert - използва CLIENT-CERT тип автентикация.

Файлът authentication.jsp показва на екрана кой е потребителя, който достъпва този ресурс и коя е същността, която го идентифицира. Тази информация се взима от HttpServletRequest съответно чрез методите getRemoteUser() и getUserPrincipal().

5.2.2. Модули с програмна автентикация

Тези модули нямат никакви особености в своя web.xml. Тук имаме два JSP файла – authentication.jsp и logout.jsp. Файлът authentication.jsp, подобно на модулите с декларативна автентикация, показва името и идентифициращата същност на потребителя. За разлика от предишните модули, обаче, потребителят успява да достъпи това JSP, без да е автентициран. Тук самото приложение се грижи за автентикацията като прави екземпляр на логин контекст и му извиква метода login(). При инициализирането на логин контекста му се подават два параметъра – име на стек от логин модули и екземпляр на обработчик на въпроси.

За улеснение на тестването, името на конфигурацията на логин модулите се взима от параметър на заявката с име “LoginModuleStackName”. По този начин, можем лесно да тестваме програмна автентикация с различни конфигурации от логин модули. Ако не се подаде такъв параметър при достъпването на приложението, на логин контекста се подава стойност “other”.

След успешна автентикация се показват на потребителя вече споменатите по-горе данни. Освен това, authentication.jsp показва съдържанието на целия субект, който взима от вече създадения логин контекст.

При двата модула за програмна автентикация съществува следната разлика при инициализирането на обработчика на съобщения:

- test_programmatic_basic: new HttpCallbackHandler(request, response, "basic", null, null, "/test_programmatic_basic"));

- test_programmatic_form: new HttpCallbackHandler(request, response, "form", "/form_pages/LoginPage.jsp", "/form_pages/ErrorPage.jsp", "/test_programmatic_form"));

В authentication.jsp има също и връзка към logout.jsp, където се извършва

програмно излизането от приложението. Това се извършва като се извика метода logout() на логин контекста, създаден по време на автентикация. След това, logout.jsp показва кой е текущия потребител.

5.3. Тестови сценарии

Заглавната страница на средата за автентикация може да се достъпи на следния адрес: http://localhost:50000/login/index.jsp. Всички тестови сценарии могат да се изпълнят като се достъпят връзки намиращи се на тази страница. Включени са следните тестови сценарии:

- Достъпване на защитена страница с BASIC тип автентикация.
- Достъпване на защитена страница с FORM тип автентикация.
- Достъпване на незащитена страница, която извършва програмна автентикация с NameAndPasswordLoginModule чрез BASIC механизъм.
- Достъпване на незащитена страница, която извършва програмна автентикация с NameAndPasswordLoginModule чрез FORM механизъм.
- Достъпване на незащитена страница, която извършва програмна автентикация с EvaluateSSOTokenLoginModule.
- Достъпване на незащитена страница, която извършва програмна автентикация с NameAndPasswordLoginModule, EmailLoginModule с флагове SUFFICIENT и CreateSSOTokenLoginModule с флаг CLOSING.
- Излизане от приложение с програмна автентикация.

6. Заключение и насоки за бъдещо развитие на разработката

Поглеждайки назад виждаме, че задачите, поставени в първа глава са успешно изпълнени:

- Проучени са възможностите, които JAAS спецификацията предлага, както и по какъв начин се използва от някои от сървърите за приложения.
- Реализирана е собствена среда за автентикация базирана на JAAS, която изпълнява всички изисквания, посочени в увода.
- Изработени са тестови приложения, с които може да се тества и демонстрира работата на средата.

Така изработената среда за автентикация може да се използва като модел за реализация на среда за автентикация от всички сървъри за приложения:

- Тя разкрива начин, по който да се поддържа използването на стандартните JAAS интерфейси за програмна автентикация.
- Предлага добавянето на нов контролен флаг към стандарта, с което биха се предоставили по-широки възможности за конфигуриране и комбиниране на механизмите за автентикация.

Като възможности за развитие и разширяване на настоящето решение могат да се идентифицират следните насоки:

- Да се добави поддръжка и за други видове приложения и протоколи. Това би могло да се осъществи, като се изработят други обработчици на съобщения за съответните протоколи.
- Да се добавят и други логин модули, които да осъществяват допълнителни механизми за автентикация – например: Kerberos, RSA SecureID Card, Smart Card. Наред с това, може да се добави и поддръжката на други видове съобщения, с които да се извличат потребителските данни необходими на логин модулите.
- Възможно е и да се промени и разшири стандарта, така че в него да се покриват

и въпросите свързани с интегрирането на средата за автентикация с контейнерите на сървърите. По този начин ще се създаде възможност за повече гъвкавост и подмяна на една среда за автентикация с друга. Подобна е идеята, залегнала в JSR-196 (Java Specification Request), чиято спецификация все още се обсъжда.

7. Използвана литература

1. Java™ Authentication and Authorization Service (JAAS) Reference Guide for the J2SE Development Kit 5.0:
   http://java.sun.com/j2se/1.5.0/docs/guide/security/jaas/JAASRefGuide.html

2. Java™ Authentication and Authorization Service (JAAS) LoginModule Developer's Guide:
   http://java.sun.com/j2se/1.5.0/docs/guide/security/jaas/JAASLMDevGuide.html

3. JAAS Authentication Tutorial:
   http://java.sun.com/j2se/1.5.0/docs/guide/security/jaas/tutorials/GeneralAcnOnly.html

4. “Java 5.0 Tiger: A developer’s Notebook”, David Flanagan, Brett McLaughlin

5. Java™ 2 Platform Standard Edition 5.0 API Specification:
   http://java.sun.com/j2se/1.5.0/docs/api/index.html

6. Java™ Platform, Enterprise Edition, v 5.0 API Specification:
   http://java.sun.com/javaee/5/docs/api/

7. JSR-000154 JavaTM Servlet 2.4 Specification:
   http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html

10. “Writing Servlet 2.3 Filters”, Stephanie Fesler:
    http://www.onjava.com/pub/a/onjava/2001/05/10/servlet_filters.html

9. The JBoss 4 Application Server J2EE Reference:
   http://docs.jboss.com/jbossas/guides/j2eeguide/r2/en/html_single/#ch8.chapter

10. WebLogic Authentication:
    http://edocs.bea.com/wls/docs92/secintro/concepts.html#wp1122439

11. “JAAS and the HTTP Session Life Cycle”, Rajesh Shah:
    http://dev2dev.bea.com/pub/a/2005/08/session_lifecycle.html

12. “JAAS Fundamentals”, Bill Kemp, Rich Helton:
    http://dev2dev.bea.com/pub/a/2003/04/Kemp_Helton.html

13. The authentication process:
    http://edocs.bea.com/wls/docs92/secintro/archtect.html#wp1047592

14. Securing Web Applications:
http://e-docs.bea.com/wls/docs81/security/thin_client.html#1028088

15. WebLogic Server 8.1 API Reference:
http://e-docs.bea.com/wls/docs81/javadocs/index.html

16. Eclipse development platform: http://www.eclipse.org

17. SAP Network: http://www.sdn.sap.com

8. Речник на термините

- application – приложение
- application server – сървър за приложения
- authentication – автентикация
- authentication provider – доставчик на услуга за автентикация
- authentication type – тип автентикация
- authorization – упълномощаване
- browser – навигатор
- callback – съобщение
- callback handler – обработчик на съобщения
- configuration – конфигурация
- context root – контекстен корен
- credentials – документи за самоличност
- domain – област
- enumeration – изброен списък
- header – заглавен атрибут
- Java Authentication and Authorization Service (JAAS) – Java услуги за автентикация и упълномощаване
- login context – логин контекст
- login module – логин модул
- logout – излизане
- pluggable authentication module (PAM) – заменяем модул за автентикация
- principal – същност
- principal validation provider – доставчик на услуга за проверка на същности
- private – личен
- privileged action – привилегировано действие
- protected area – защитена област
- realm – област
- remote invocation – отдалечено извикване
- request – заявка
- response – отговор
- security domain – област на сигурност
- security manager – мениджър по сигурността
- session id cookie – бисквитката представляваща идентификатор на сесията
- Single Sign-On (SSO) – автоматичен достъп след първоначално автентициране
- subject – субект

- tag – маркер, етикет
- user account – потребителската регистрация
- web – уеб
- web container – уеб контейнер
- web module – уеб модул

9. Приложение с фрагменти от кода

9.1. Клас AppConfigurationEntryExtension

```
public class AppConfigurationEntryExtension extends AppConfigurationEntry {
  private ControlFlag controlFlag = null;

  public AppConfigurationEntryExtension(String loginModuleName,
                                        String flag,
                                        Map<String,?> options) {
    super(loginModuleName, LoginModuleControlFlag.REQUIRED, options);

    if (flag != null) {
        flag = flag.toUpperCase();
        controlFlag = ControlFlag.valueOf(flag);
    } else {
        throw new IllegalArgumentException("The login module flag cannot be null.");
    }
  }

  public LoginModuleControlFlag getControlFlag() {
    return null;
  }

  public ControlFlag getControlFlagExtension() {
    return controlFlag;
  }
}
```

9.2. Изброен списък ControlFlag

```
public enum ControlFlag {REQUIRED, REQUISITE, SUFFICIENT, OPTIONAL, CLOSING};
```

9.3. Клас LoginModuleConfiguration, метод AppConfigurationEntry[] getAppConfigurationEntry(String context)

```
public AppConfigurationEntry[] getAppConfigurationEntry(String context) {
   AppConfigurationEntry entry = null;
   Hashtable<String, String> options = new Hashtable<String, String>(1);

  options.put(ForwardLoginModule.LOGIN_MODULES_STACK_NAME, context);
  entry = new AppConfigurationEntry(
                                FORWARD_LOGIN_MODULE,
                                AppConfigurationEntry.LoginModuleControlFlag.REQUIRED,
                                options);

  return new AppConfigurationEntry[] {entry};
}
```

9.4. Клас ForwardLoginModule

9.4.1. Метод initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)

```
public void initialize(Subject subject,
                       CallbackHandler callbackHandler,
                       Map sharedState,
                       Map options) {

  try {
    String loginModuleStack = (String) options.get(LOGIN_MODULES_STACK_NAME);
    loginContext = new CustomLoginContext(loginModuleStack, subject, callbackHandler);
  } catch (Exception e) {
    loginContext = null;
    throw new SecurityException(e.toString());
  }
}
```

9.4.2. Метод boolean login()

```
public boolean login() throws LoginException {
if (loginContext != null) {
   loginContext.login();
} else {
   return false;
}
  return true;
}
```

9.5. Клас CustomLoginContext, метод Principal getPrincipal()

```
private Principal getPrincipal() {
  if (this.principal != null) {
    return this.principal;
  } else if (this.subject != null) {
    Set set = this.subject.getPrincipals(UserIdPrincipal.class);
    Iterator iterator = set.iterator();

    if (iterator.hasNext()) {
        this.principal = (Principal) iterator.next();
    }

    if (this.principal == null) {
        set = subject.getPrincipals();
        iterator = set.iterator();

      if (iterator.hasNext()) {
        this.principal = (Principal) iterator.next();
      }
    }
  }

  return this.principal;
}
```

9.6. Клас AuthStackProcessAction, метод Object run()

```
public Object run() throws LoginException {
  boolean processed = false;
  boolean localProcessed = false;
  boolean onlyFinal = false;
  int i = 0;
  LoginException exception = null;
  LoginException weakException = null;

  for (; i < modules.length; i++) {
    if ((!onlyFinal || (configuration[i].getControlFlagExtension() == ControlFlag.CLOSING))) {
      localProcessed = false;

      try {
        switch (type) {
          case CustomLoginContext.OPERATION_LOGIN: {
            localProcessed = modules[i].login();
            break;
          }
          case CustomLoginContext.OPERATION_ABORT: {
            localProcessed = modules[i].abort();
            break;
          }
          case CustomLoginContext.OPERATION_COMMIT: {
            localProcessed = modules[i].commit();
            break;
          }
          case CustomLoginContext.OPERATION_LOGOUT: {
            localProcessed = modules[i].logout();
            break;
          }
        }

        if (localProcessed) {
          weakException = null;
          processed = true;
```

```
    if ((type == CustomLoginContext.OPERATION_LOGIN)
          || (type == CustomLoginContext.OPERATION_COMMIT)
          && (configuration[i].getControlFlagExtension() == ControlFlag.SUFFICIENT)) {

      onlyFinal = true;
    }
  }
} catch (LoginException e) {
  if (configuration[i].getControlFlagExtension() == ControlFlag.REQUISITE) {
    if ((type == CustomLoginContext.OPERATION_ABORT) ||
       (type == CustomLoginContext.OPERATION_LOGOUT)) {

      if (exception == null) {
        exception = e;
      }
    } else {
      throw e;
    }
  } else if ((configuration[i].getControlFlagExtension() == ControlFlag.REQUIRED) ||
             (configuration[i].getControlFlagExtension() == ControlFlag.CLOSING)) {

    if (exception == null) {
      exception = e;
    }
  } else {
    if (weakException == null) {
      weakException = e;
    }
  }
} catch (Throwable e) {
  LOCATION.traceThrowableT(Severity.ERROR,
                           "Login module {0} from authentication stack {1} errors
                            while authenticating the caller.",
                           new Object[] {modules[i].getClass().getName(),
                                         this.authStack},
                           e);
```

```
        throw new SecurityException("Error during login: " + e.toString());
      }
    }
  }

  if (exception != null) {
    if (exception instanceof LoginException) {
      throw (LoginException) exception;
    } else {
      throw new LoginException(exception.toString());
    }
  }

  if (!processed && (weakException != null)) {
    if (weakException instanceof LoginException) {
      throw weakException;
    } else {
      throw new LoginException(weakException.toString());
    }
  }

  if (!processed) {
    if (LOCATION.beDebug()) {
      LOCATION.debugT("No login module succeeded. Authentication stack: {0} size: {1}.",
                      new Object[] {this.authStack, new Integer(modules.length)});
    }

    throw new LoginException("No login module succeeded.");
  }

  return null;
}
```

9.7. Клас LoginFilter

9.7.1. Метод init(FilterConfig config)

```
public void init(FilterConfig config) throws ServletException {
  ServletContext servletContext = config.getServletContext();
  ApplicationContext applicationContext =
                        ((ServletContextImpl) servletContext).getApplicationContext();

  this.applicationConfiguration = applicationContext.getWebApplicationConfiguration();
}
```

9.7.2. Метод doFilter(ServletRequest request, ServletResponse response, FilterChain chain)

```
public void doFilter(ServletRequest request,
                     ServletResponse response,
                     FilterChain chain) throws IOException, ServletException {

 HttpServletRequest httpRequest = (HttpServletRequest) request;
 HttpServletResponse httpResponse = (HttpServletResponse) response;

 authStack = applicationConfiguration.getAuthType();
 if (authStack == null) {
     authStack = "other";
 }

 try {
  LoginContext loginContext = new CustomLoginContext(
       authStack,
       new Subject(),
       new HttpCallbackHandler(httpRequest, httpResponse, applicationConfiguration));

  loginContext.login();

  if (httpResponse.isCommitted()) {
    return;
```

```
    }

    chain.doFilter(request, response);
  } catch (LoginException le) {
    location.traceThrowableT(Severity.ERROR, "LoginFilter exception", le);
  }
}
```

9.8. Изброен списък AuthenticationStatusCallback

```
public enum AuthenticationStatusCallback implements Callback {FAILED, PASSED};
```

9.9. Клас UserIdPrincipal

```
public class UserIdPrincipal implements java.security.Principal, java.io.Serializable {
 private int hashCode = -1;
 private String name = null;

 public UserIdPrincipal(String name) {
  this.name = name;
  this.hashCode = name.hashCode();
 }

 public boolean equals(Object otherPrincipal) {
  if (otherPrincipal.getClass().isInstance(this)) {
   if (this.hashCode == otherPrincipal.hashCode()) {
     if (name.equals(((UserIdPrincipal) otherPrincipal).getName())) {
       return true;
     }
   }
  }

  return false;
 }
```

```
  public String getName() {
    return this.name;
  }

  public int hashCode() {
    return this.hashCode;
  }

  public String toString() {
    return this.getClass().getName() + ": " + this.name;
  }
}
```

9.10. Клас CreateSSOTokenLoginModule, метод boolean commit()

```
  public boolean commit() throws LoginException {
     Set<UserIdPrincipal> principals = subject.getPrincipals(UserIdPrincipal.class);

     if (principals != null && !principals.isEmpty()) {
         UserIdPrincipal[] array = new UserIdPrincipal[0];
         array = (UserIdPrincipal[]) principals.toArray(array);

         UserIdPrincipal principal = array[0];
         String name = principal.getName();
         String cookieValue = null;

         try {
            cookieValue = new String(Base64.encode(name.getBytes()));
         } catch (Exception e) {
            throw new LoginException(e.toString());
         }

         boolean result = setCookie(cookieValue);

         if (result) {
```

```
            if (principals == null || principals.isEmpty()) {
                SSOTokenPrincipal ssoPrincipal = new SSOTokenPrincipal(name);
                subject.getPrincipals().add(ssoPrincipal);
            }

            SSOTokenCredential credential = new SSOTokenCredential("SSOToken",
                                                                   cookieValue);
            subject.getPrivateCredentials().add(credential);

            successful = true;
        }

        return result;
    } else {
        shouldBeIgnored = true;
        return false;
    }
}
```