A.C.A
IT-SOLUTIONS
DELIVER BETTER SOLUTIONS FASTER

Java Enterprise Security

Stijn Van den Enden

s.vandenenden@aca-it.be

Agenda

- Java EE introduction
- Web module security
- EJB module security
- Runtime configuration
- Other security aspects
- Spring Security
- JBoss SEAM Security

Java EE introduction

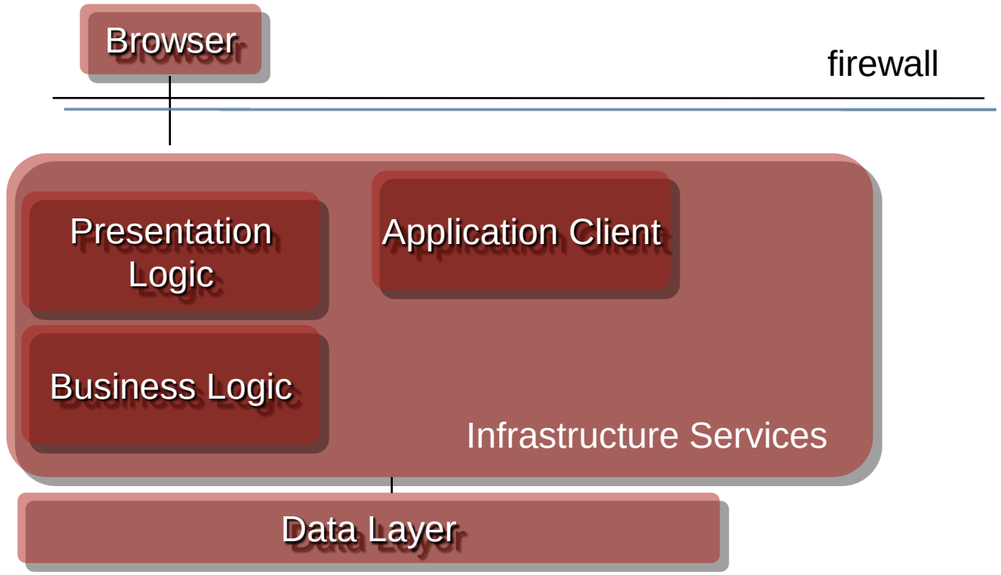
The Java Enterprise platform

Security concepts

Packaging & deployment

Java EE Platform Overview

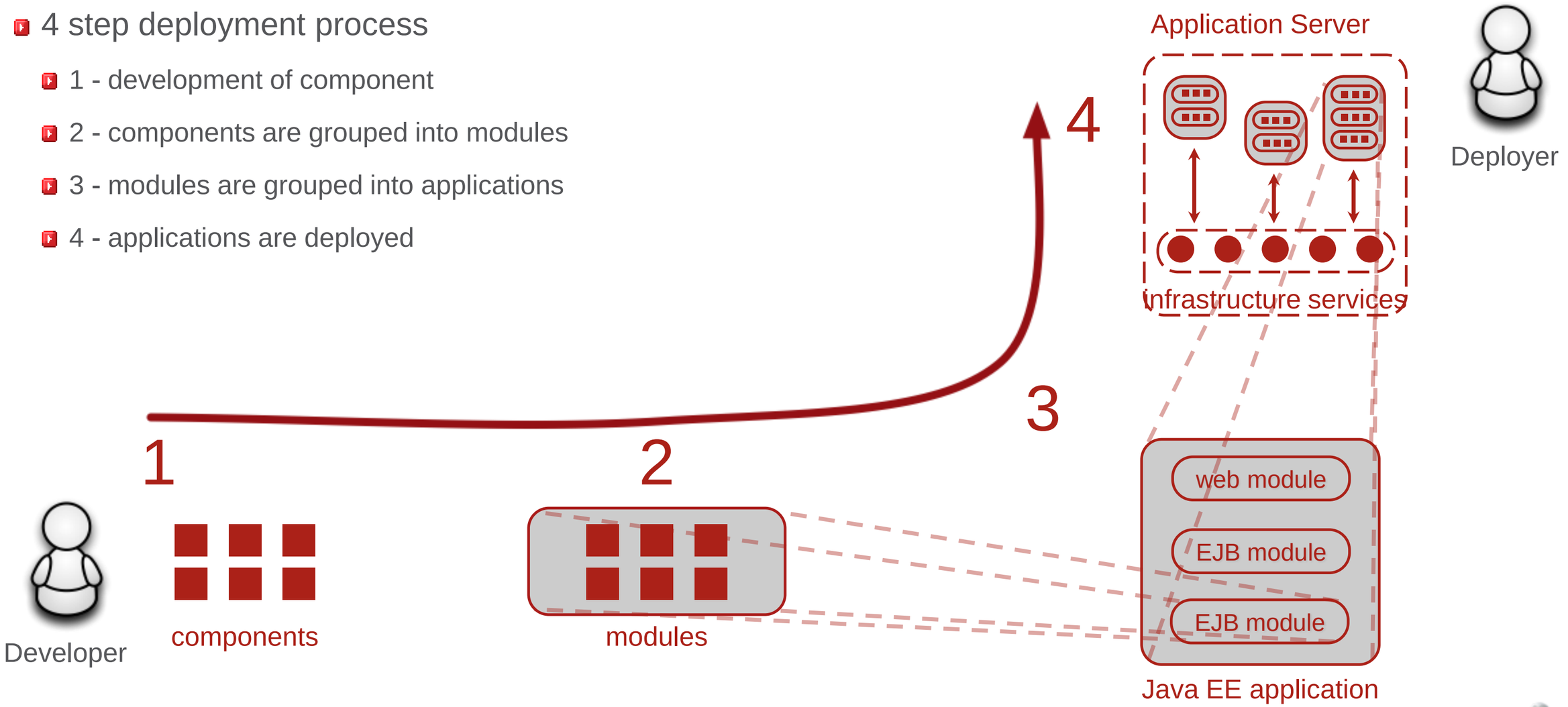
- The Java EE platform is essentially a distributed application server environment in Java, it provides:
  - A multitier (n-tier) distributed application model:
    - User interface
      - Handles all user interaction with the application
    - Presentation logic:
      - Defines what the user interface displays
      - Defines how the requests are handled
    - Business logic:
      - Implements the business rules of the application
    - Infrastructure services
      - e.g. Transaction support, messaging communication
    - Data layer



- This presentation focuses on the security aspect in the different Java EE layers

Java EE Platform Overview

- 4 step deployment process
  - 1 - development of component
  - 2 - components are grouped into modules
  - 3 - modules are grouped into applications
  - 4 - applications are deployed



Web module security

- Authentication
- Authorization
- Declarative vs. programmatic security
- Principal delegation

Terminology

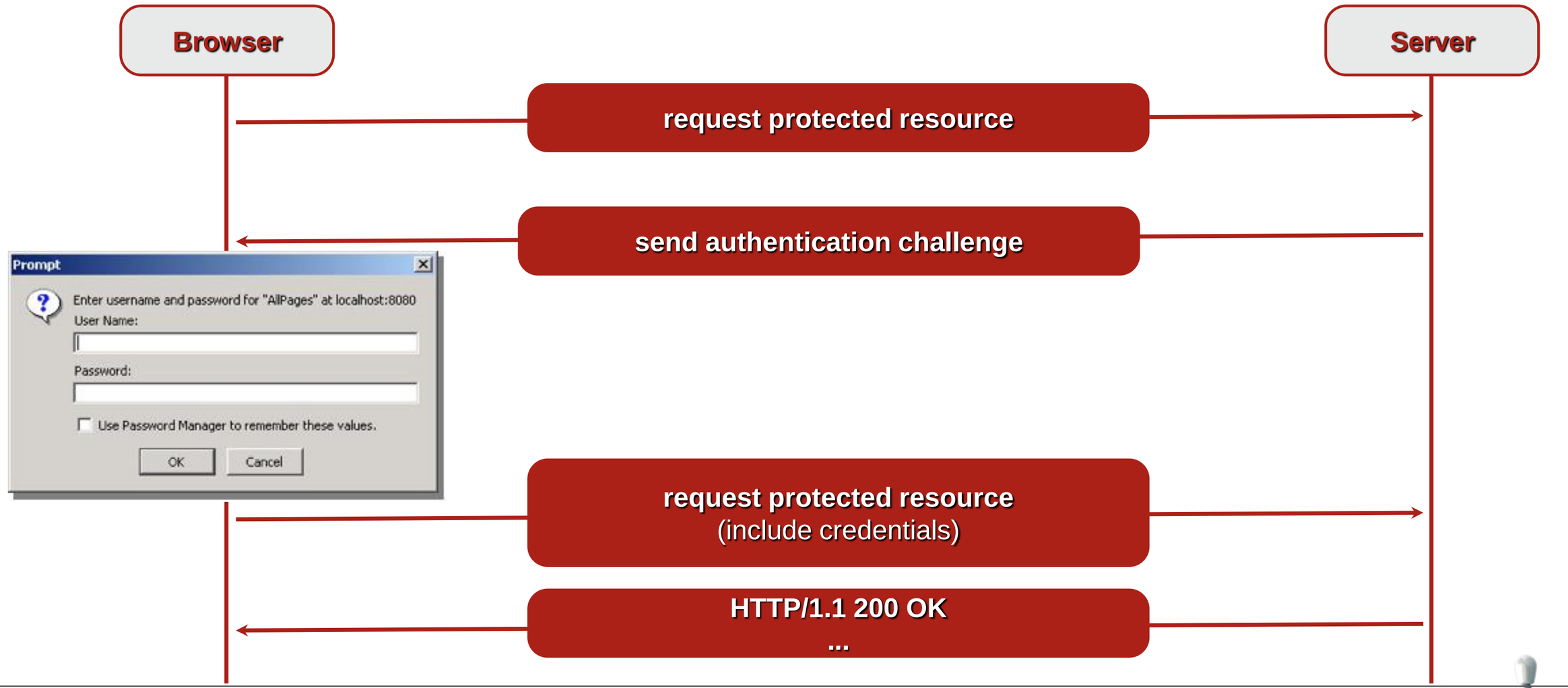
- Authentication = asserting that a user is who he/she claims to be.
- User can be
  - Physical user
  - Services
  - External systems
- A principal is an entity that can be authenticated by an authentication protocol
  - Identified using a principal name and authenticated using authentication data
  - Has one or more roles in an application
  - Is authorized to perform certain actions based on a role
- A credential contains or references information (security attributes) used to authenticate a principal for Java EE product services
  - Acquired upon authentication, or from another principal that allows its credential to be used (delegation)

Declarative vs. programmatic security control

- Declarative security control:
  - Allows deployer to specify security policy through deployment descriptor
  - No need to change application's code
  - Container enforces these security constraints when the application is executed
- Programmatic security control:
  - Applications can enforce security constraints at code level

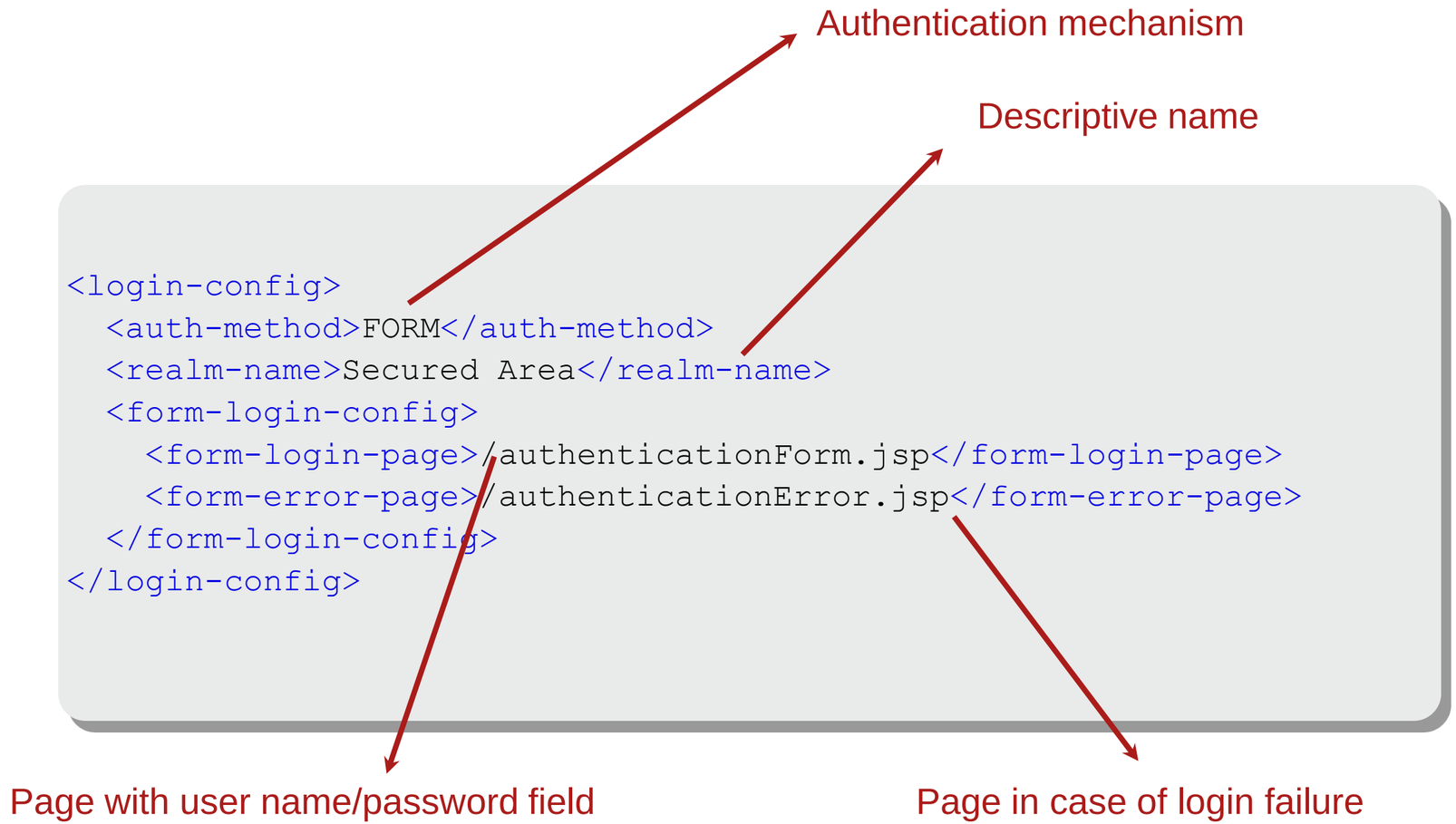
Authentication - overview

- Authentication



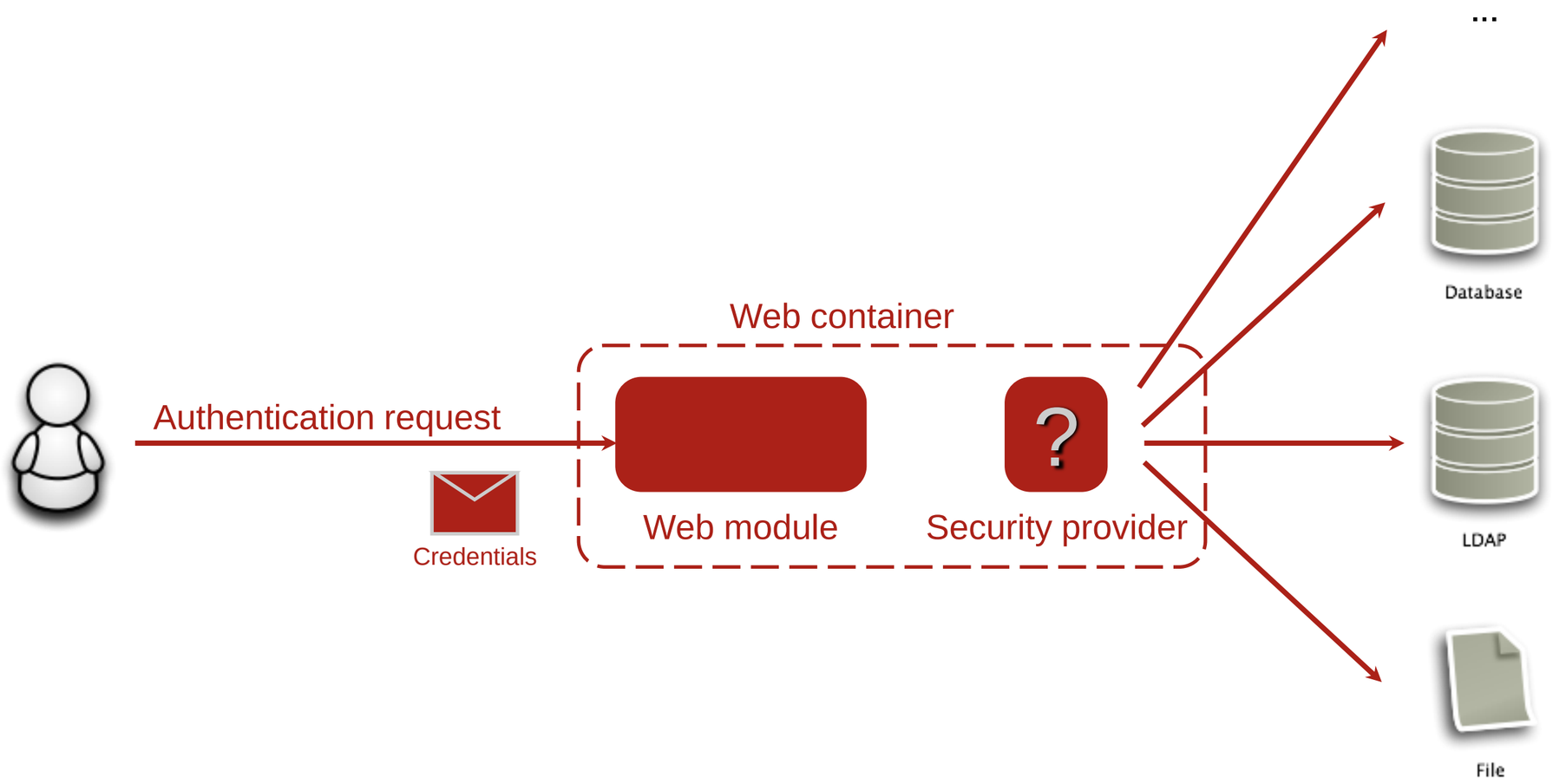
Authentication - declarative security

- Authentication mechanism must be specified in deployment descriptor
- Choice between
  - BASIC
  - DIGEST
  - FORM
  - CLIENT-CERT



Authentication - security providers

- Server must be configured appropriately to handle authentication requests
- User repository can exist in several forms
  - database
  - LDAP repository
  - simple file
  - ...



Authorization - declarative security

- Determines what roles are required

Who ?

What ?

How ?

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>AllPages</web-resource-name>
    <url-pattern>/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
  </auth-constraint>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

Authorization - declarative

```
<web-resource-collection>
    <web-resource-name>AllPages</web-resource-name>
    <url-pattern>/*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
</web-resource-collection>
```

This security constraint involves all pages (/*) which are accessed by either a GET or a POST request.

```
<auth-constraint>
  <role-name>admin</role-name>
</auth-constraint>
```

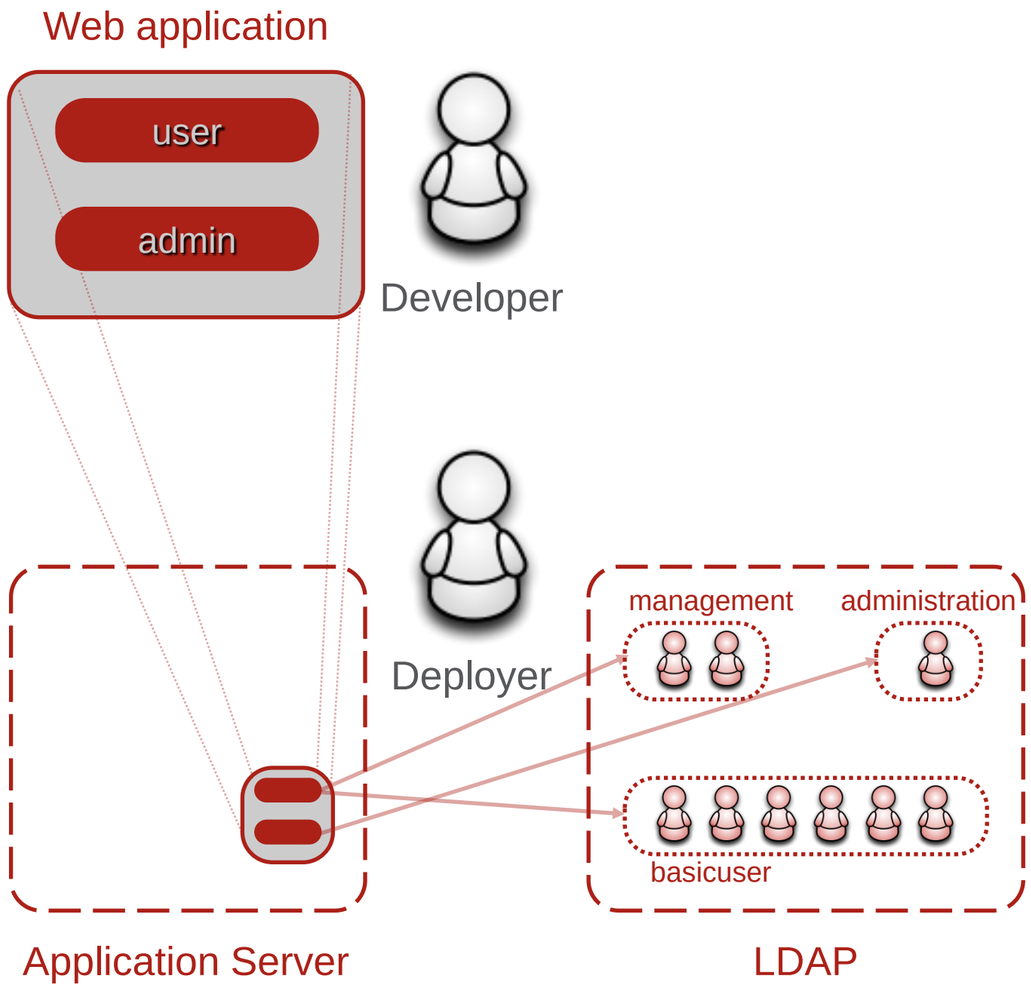
The user needs the “admin” role in order to access the resources covered by this security constraint.

```
<user-data-constraint>
  <transport-guarantee>CONFIDENTIAL</transport-guarantee>
</user-data-constraint>
```

The protected resources need to be accessed in a confidential manner (HTTPS)

Authorization - role mapping

- Consider a sample web application with the following security roles
  - user: normal end user
  - admin: can access application screens to change log level, view statistics, ...
- Deployer must deploy this application in an appserver
  - which uses the company LDAP server as its security provider
  - LDAP contains company users and their groups (e.g. administration, management, basicuser)

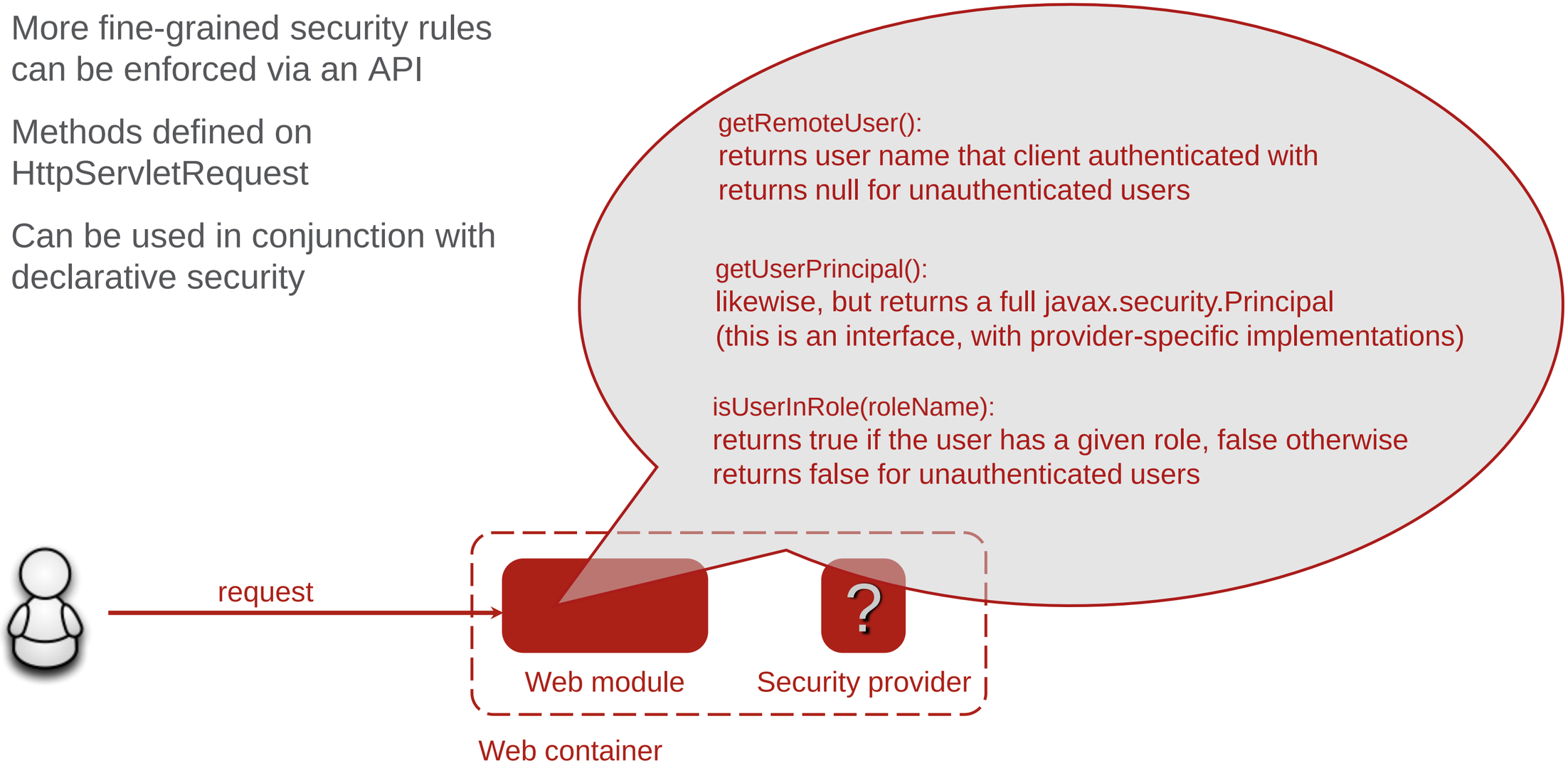


Advantages/drawbacks of declarative security

- Advantages:
  - Simplicity
  - Encapsulation
    - Security is defined in one place in the application, not scattered throughout the application.
- Drawbacks:
  - Granularity
    - Only authorizations at page level can be specified declaratively
    - Not possible to render small page sections based on role
  - Restrictive
    - Only based on the presence of a particular role
    - Not possible to create more fine grained policies (e.g. “access only during business hours”)

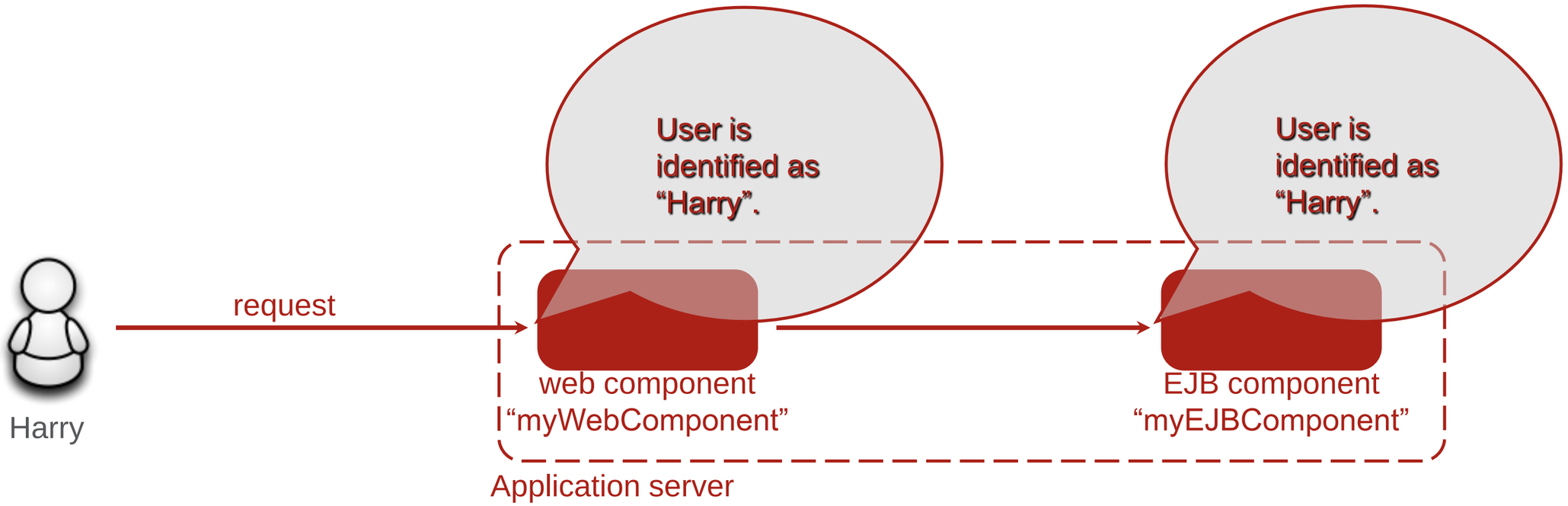
Programmatic security

- More fine-grained security rules can be enforced via an API
- Methods defined on HttpServletRequest
- Can be used in conjunction with declarative security



Principal delegation - default behavior

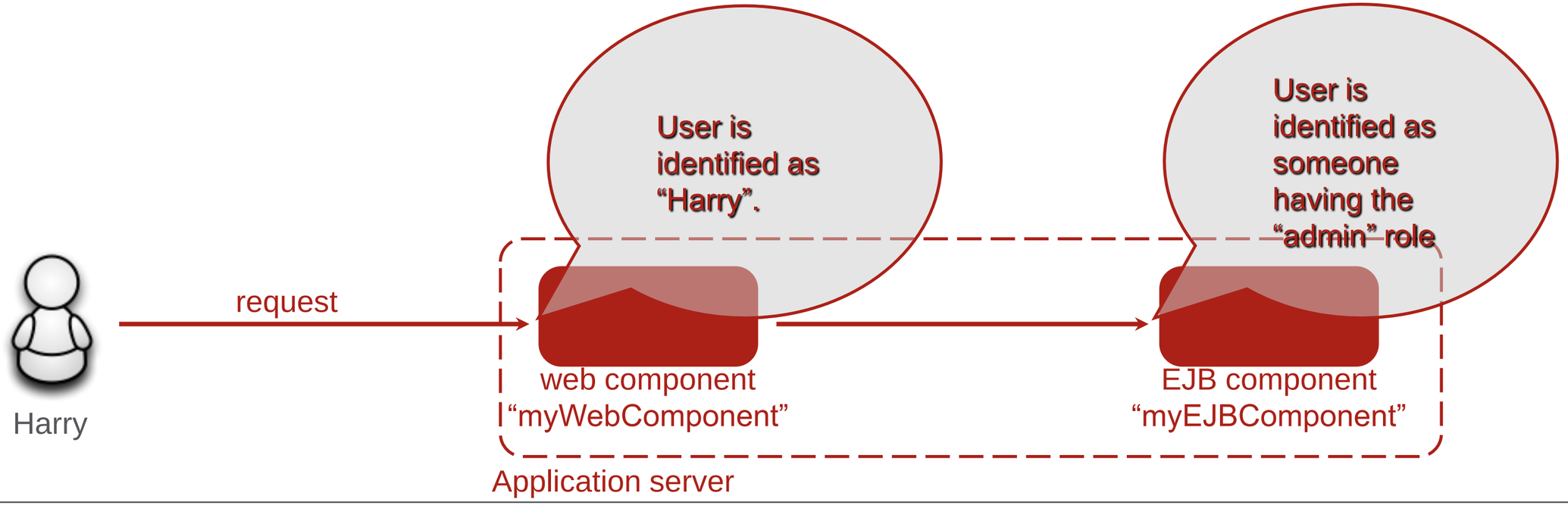
- Harry sends requests to myWebComponent
  - Principal is “Harry”
- myWebComponent calls myEJBComponent
  - Principal is delegated (=> principal is still “Harry”)



Principal delegation - “run as” behavior

- Web components can be configured to run under principal having a particular role

```
<servlet>
  <servlet-name>myWebComponent</servlet-name>
  <servlet-class>be.aca.security.MyWebComponent</servlet-class>
  <run-as>
    <role-name>admin</role-name>
  </run-as>
</servlet>
```



EJB module security

Authorization

Declarative vs. programmatic security

Principal delegation

Declarative security

- Declarative security can be applied to EJB components
- Similar concept as for web components
  - For web component, level of granularity was a URL pattern
  - For EJB components, level of granularity is a method


```
<method-permission>
  <role-name>admin</role-name>
  <method>
    <ejb-name>MyEJBComponent</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
<method-permission>
  <role-name>admin</role-name>
  <role-name>manager</role-name>
  <method>
    <ejb-name>MyEJBComponent</ejb-name>
    <method-name>method2</method-name>
  </method>
</method-permission>
```

or

```
@RolesAllowed("admin")
@Stateless public class MyEJBComponentBean
        implements MyEJBComponent {
    public void method1() {
    }

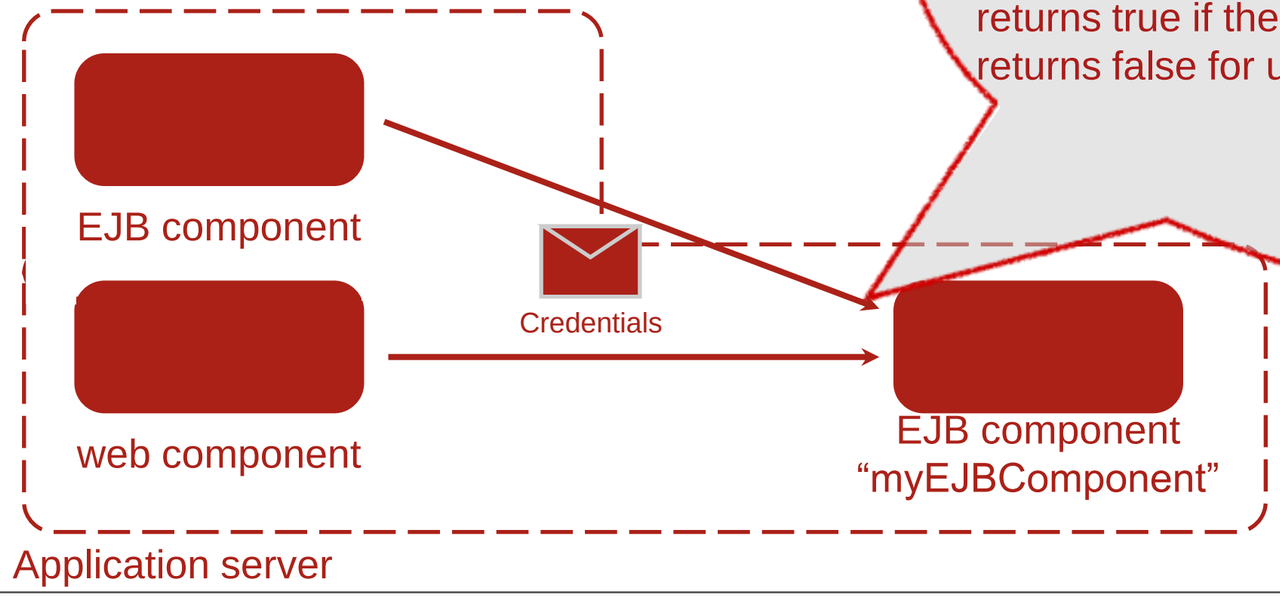
    @RolesAllowed({"admin", "manager"})
    public void method2() {
    }
}
```

Programmatic security

- More fine-grained security rules can be enforced via an API
- Methods defined on EJBContext
  - Every EJB has access to this context
  - getCallerPrincipal & isCallerInRole
  - “caller” is a more appropriate term than “user” in this scenario

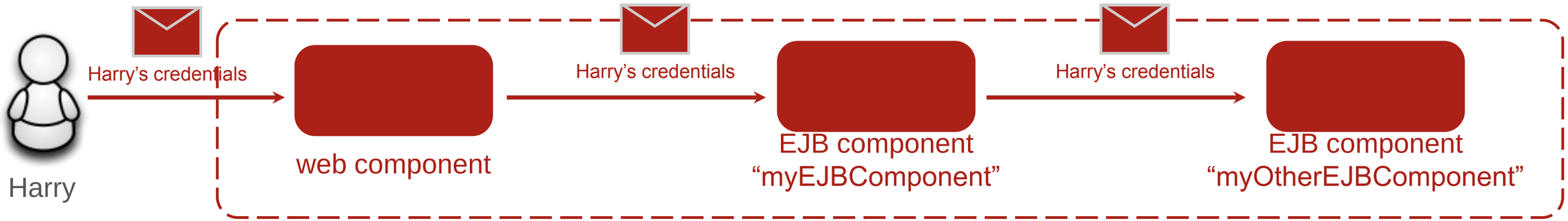
getCallerPrincipal():
returns a full javax.security.Principal instance
(this is an interface, with provider-specific implementations)

isCallerInRole(roleName):
returns true if the user has a given role, false otherwise
returns false for unauthenticated users



Principal delegation - default behavior

- Same concept as for web modules
- Default behavior: propagate current principal
  - Harry's credentials are passed throughout the whole application



Principal delegation

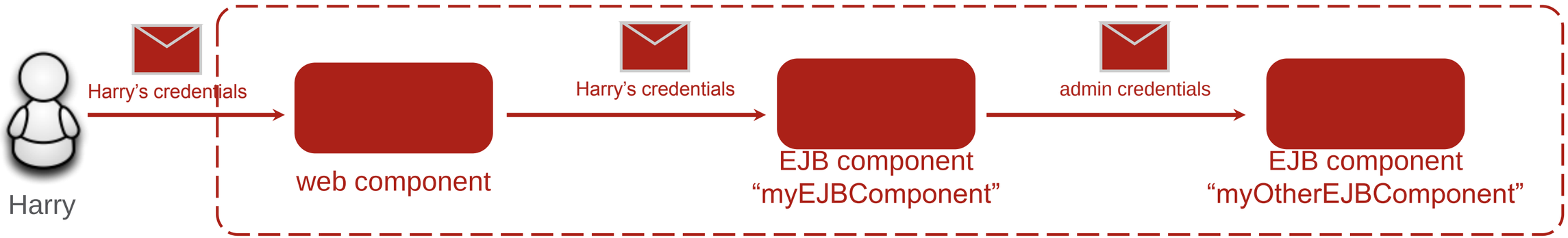
- “run-as” behavior can be used to specify a principal with a given role
  - myEJBComponent runs as principal having the “admin” role

```
<enterprise-beans>  <session>  ...
<security-identity>      <run-as>
<role-name>admin</role-name>      </run-as>
</security-identity>...
</session></enterprise-beans>
```

or

```
@RolesAllowed(“admin”)
@Stateless public class MyEJBComponentBean
        implements MyEJBComponent {
    public void method1() {
    }

    @RolesAllowed({“admin”, “manager”})
    public void method2() {
    }
}
```



Runtime configuration

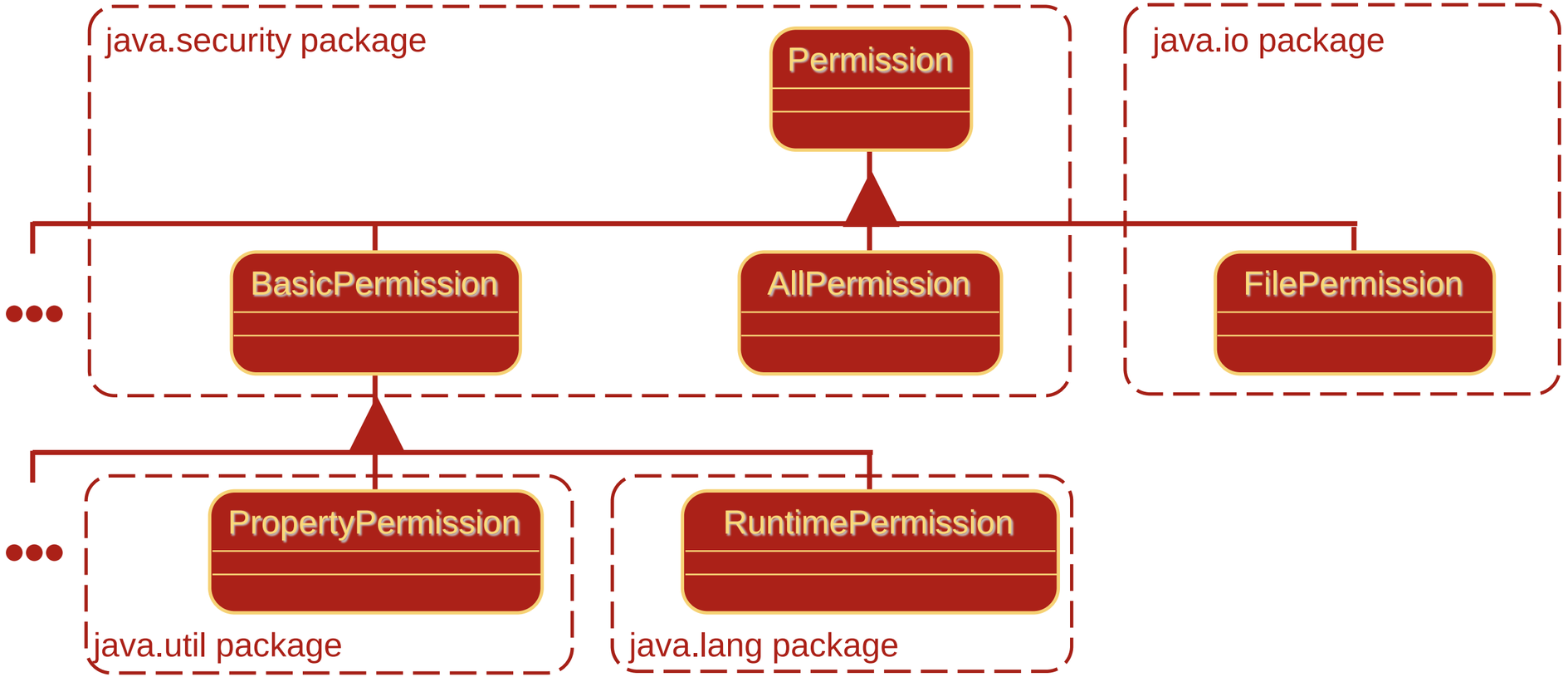
Security policy

Protection domains

Resource configuration

Security policy - permissions

- Basis of the Java security model
- Represents the right to access a particular resource (resource target)
  - optionally also which actions can be done to that resource (resource actions)



Security policy - permissions

- Examples:

	RuntimePermission	FilePermission
resource target	getClassLoader exitVM getStackTrace queuePrintJob stopThread ...	file or directory name
resource actions	n/a	read write execute delete

Security policy - policies

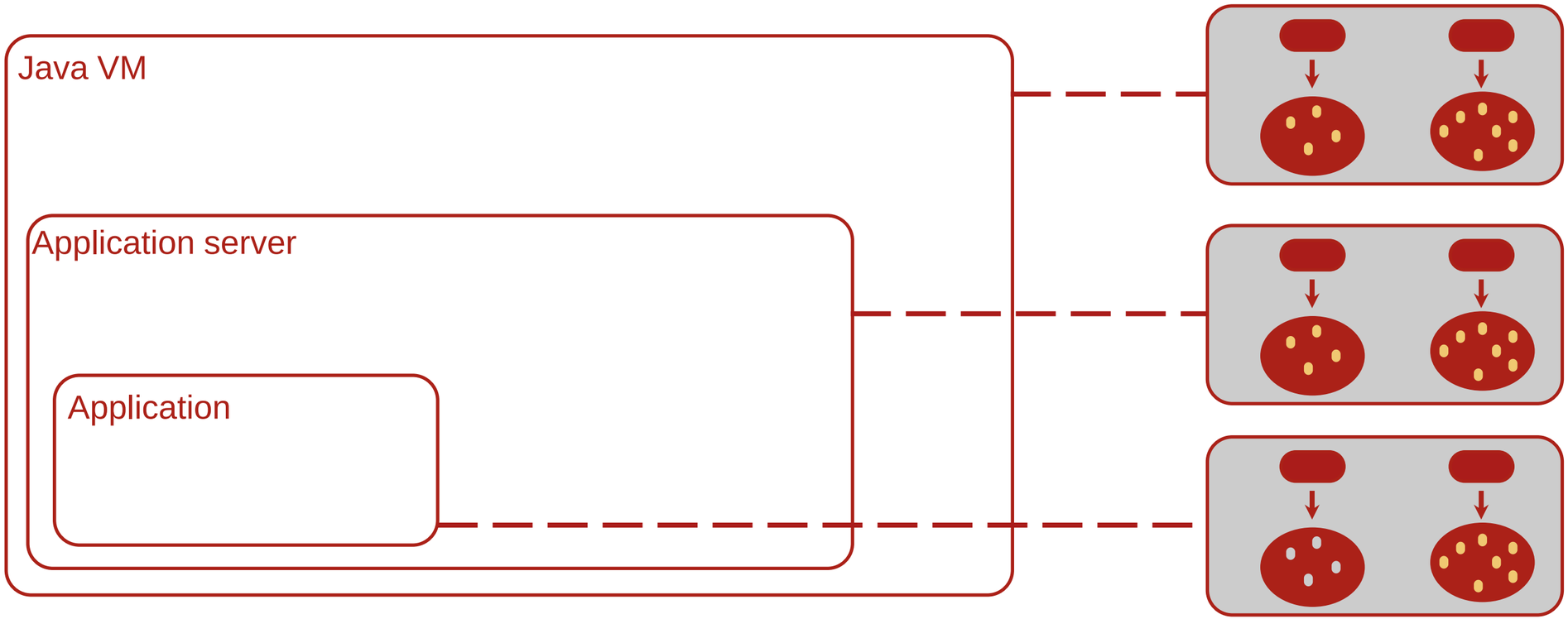
- A security policy defines which codebase can run under which permissions
  - allows definition of sandbox boundaries per codebase

- Described in a security policy file
  - placeholders can be defined for web component (e.g. ${webComponent}), EJB components, ...

```
grant codeBase "file:${webComponent}" {   permission
java.lang.RuntimePermission "stopThread";
   permission java.io.FilePermission "/var/log/myApp.log", "write"; };
```

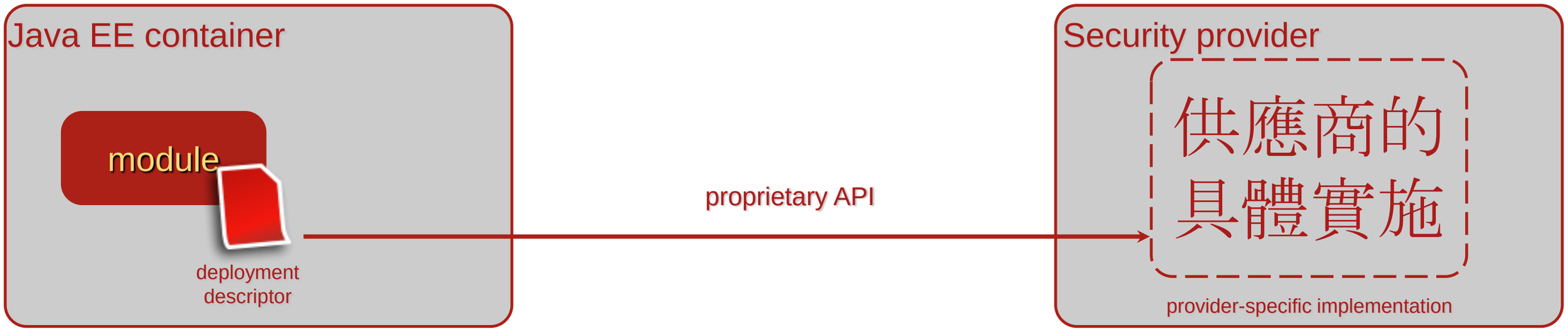
Security policy - policies at different levels

- In a Java EE environment, different security policies apply:
  - at application level
  - at server level
  - at JVM level



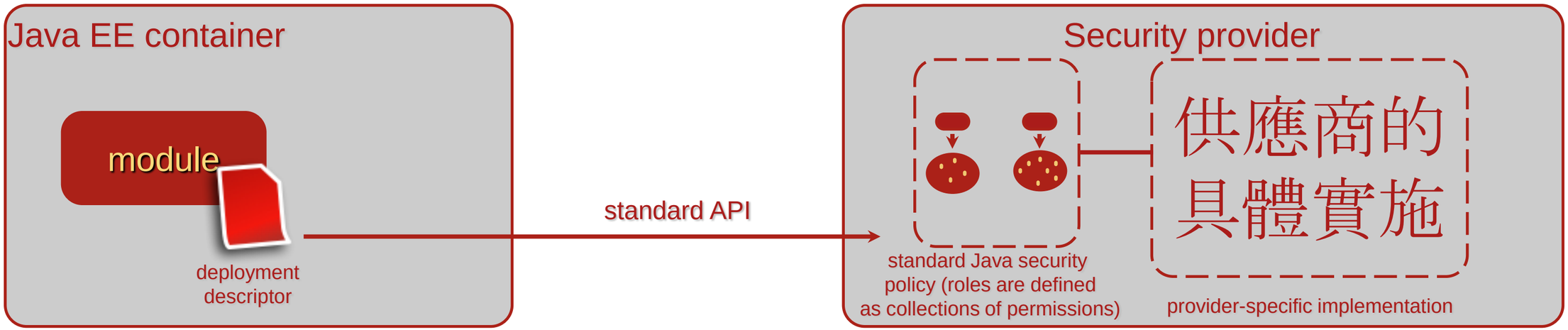
Security policy - JACC

- Security policies can also form the basis of a contract between Java EE containers and security providers
- Before JACC (JSR-115: Java Authorization Contract for Containers), containers were responsible for interacting with security providers
  - developer declares authorization rules in deployment descriptor
  - container maps those to concrete roles offered by a security provider (e.g. LDAP)
  - portability issue: some security providers were supported in one Java EE container, but not in another



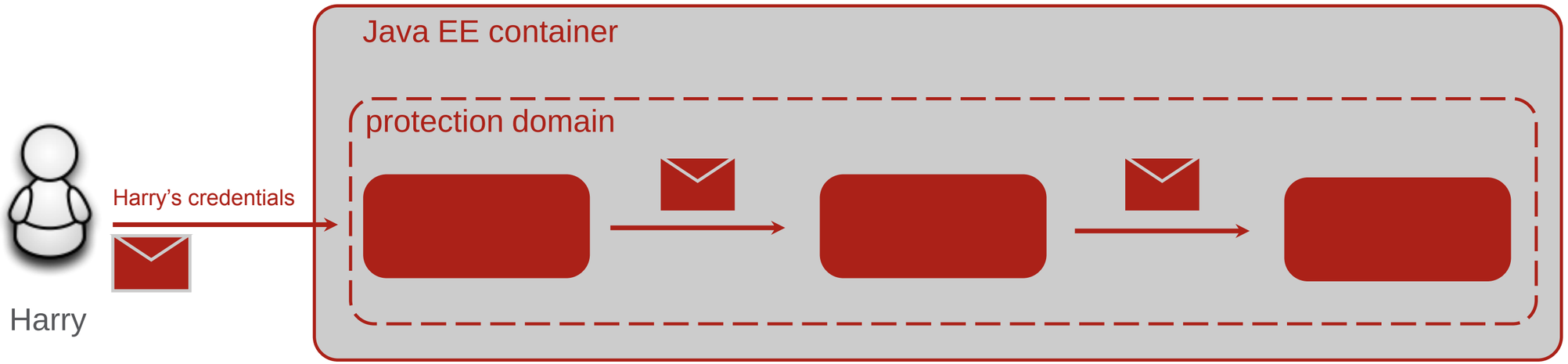
Security policy - JACC

- JACC allows
  - security providers to expose their information as a standard Java security policy
  - containers to easily query that information, regardless of implementation
- Standard contract between containers and security providers
  - any security provider can be “plugged into” any Java EE container
- Similar concept exists for authentication
  - JSR-196: Java Authentication Service Provider Interface for Containers



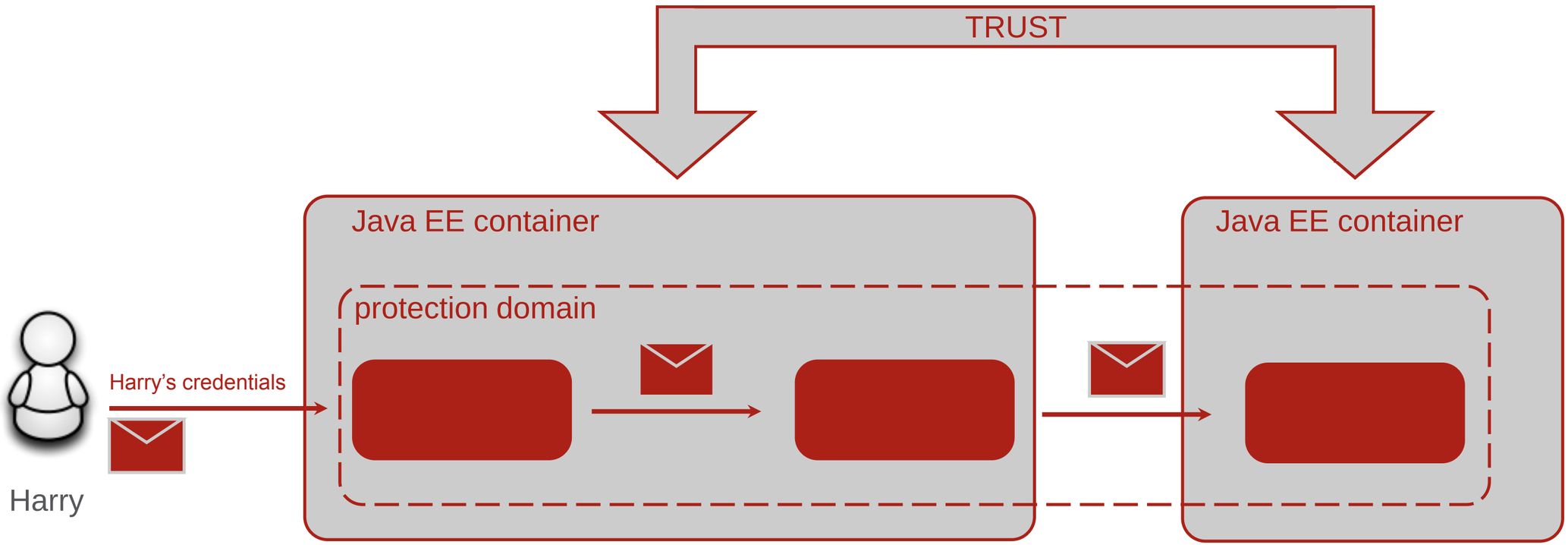
Protection domains - concept

- A protection domain is a set of components that are assumed to trust each other
  - no authentication is needed between those entities
- In Java EE, the container provides an authentication boundary between external clients and its hosted components
- inside the container's boundaries, components have the freedom to
  - either propagate the caller's identity
  - or choose an identity (based on knowledge of authorization constraints imposed by the called component)



Protection domains - across container boundaries

- Boundaries of protection domains don’t necessarily align with container boundaries
- Possible to establish trust relationship between containers in order to define a global protection domain
- No authentication needed for inter-container invocations

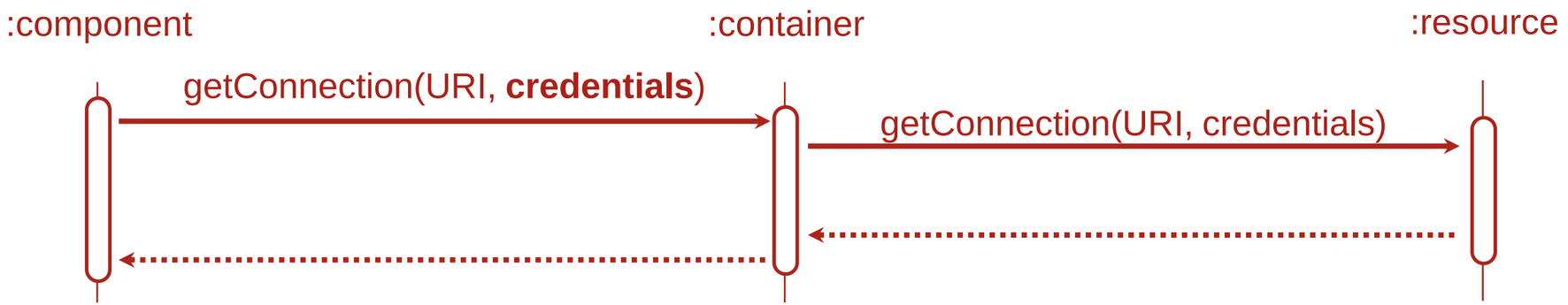


Resource configuration - concept

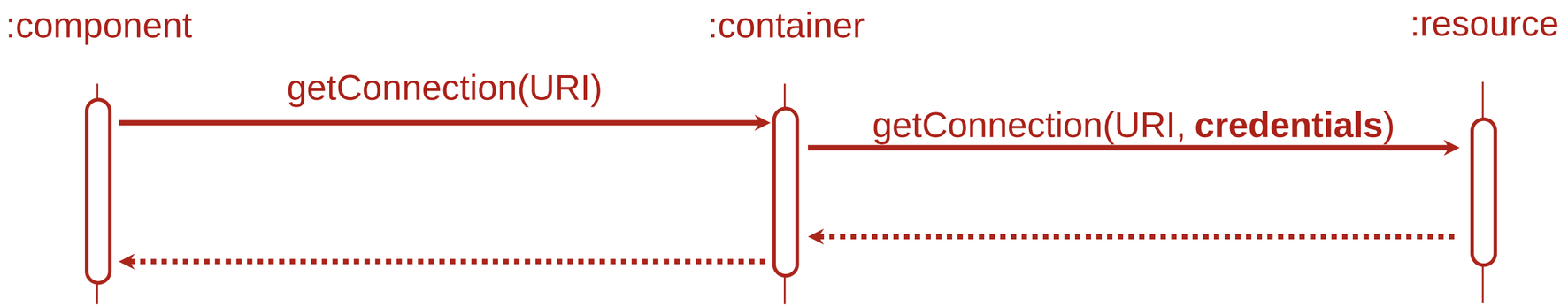
- A container can make many types of resource available to its components, e.g.
  - data sources
  - MOM (message oriented middleware)
  - mail sessions
  - ...
- Each of these may require authentication

Resource configuration - authentication

- Resource authentication can be
  - component managed (component passed credentials when asking for a connection)



  - container-managed (container has been configured with the required credentials for the resource)

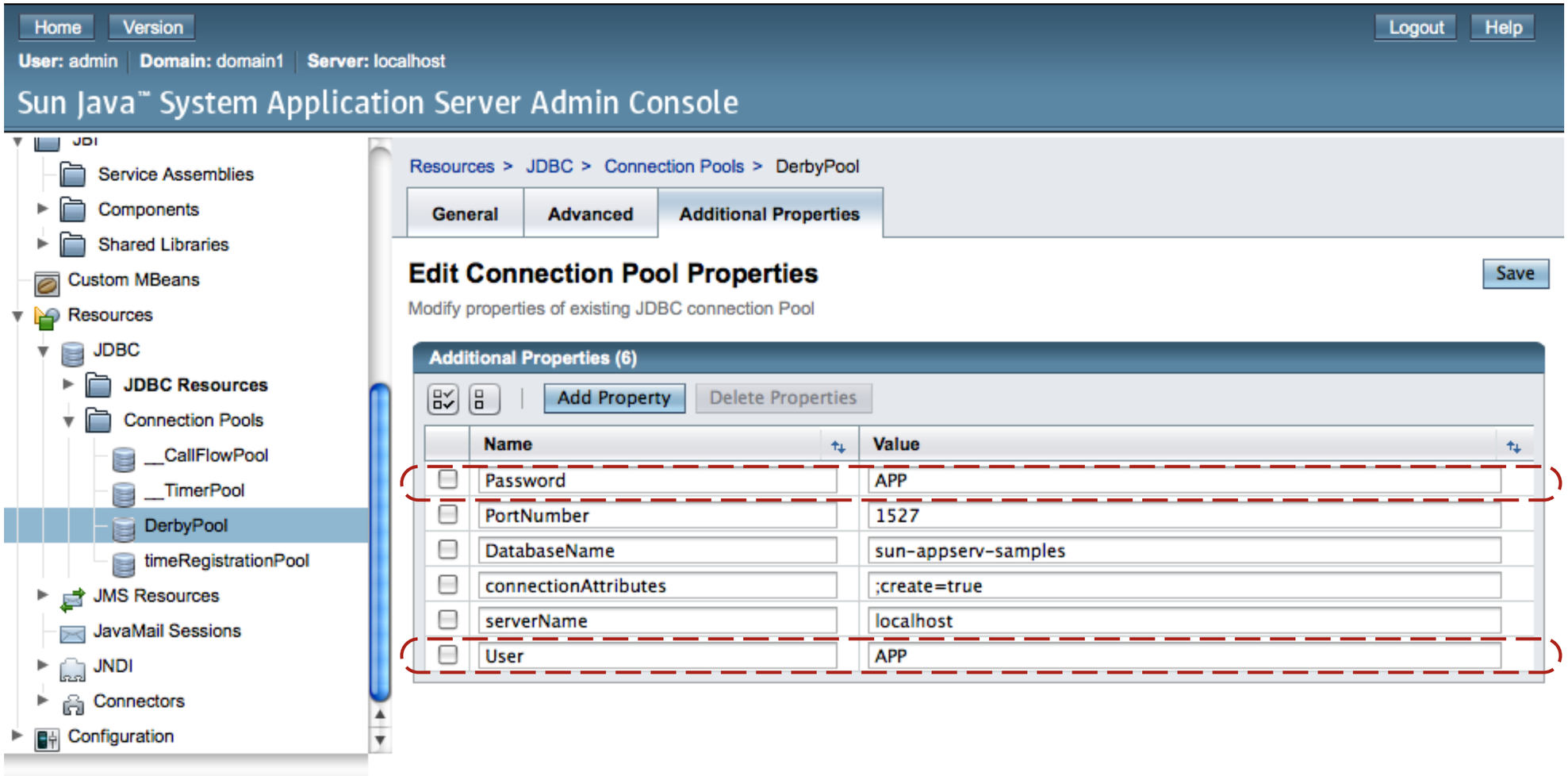


Resource configuration - component-managed authentication

- Component-managed authentication doesn’t specify how the component acquires the necessary credentials
  - e.g. read from a configuration file
  - e.g. obtained from an authenticated client
    - client certificate
    - user name/password
    - ...

Resource configuration - container-managed authentication

- Example of datasource definition (+ authentication) in Glassfish:



Standard Java EE security - wrap up

- Standard Java EE security
  - can handle most common security requirements
  - can be configured/implemented quite easily
- But...
  - Configuration of security provider is container's responsibility
    - may not be the most optimal approach
    - dependent on whatever security providers the container offers
      - less of a problem in modern containers where JACC is supported
  - Authentication support may be too limited
    - remember-me, auto-login
    - single sign-on
  - Authorization support may be too limited
    - only role-based access is supported

Standard Java EE security - wrap up

- Often DIY-frameworks are used to overcome the limitations
  - e.g. Using a Servlet Filter approach to validate whether a user is authenticated
- Other frameworks offer more elaborate security solutions:
  - Spring Security
  - JBoss SEAM
- A standard like XACML provides more possibilities
  - XACML : eXtensible Access Control Markup Language
  - OASIS standard
  - allows description and enforcement of fine-grained authorization rules in an XML syntax
  - added benefit of interoperability (Java & .NET)

Appserver comparison chart

- Server vendors offer security solutions on top of what the Java EE spec mandates

	Glassfish v2	JBoss 5	Apache Geronimo 2
SSO support	Yes: JSR-196 is supported since v2. OpenSSO (https://opensso.dev.java.net) is a JSR-196 compliant implementation. Has many extensions (e.g. OpenID, CAS, HTTP Negotiate/SPNEGO).	JSR-115 (JACC) supported. Working on JSR-196 (JASPI). JBoss Federated SSO project can be used for SSO solutions.	JSR-115 (JACC) supported. JSR-196 (JASPI) almost complete. OpenID support through openid4java library.
XACML	XACML support provided through OpenSSO project	JBossXACML	(unknown)

Spring Security

History & features

High level concepts

Authentication

Authorization

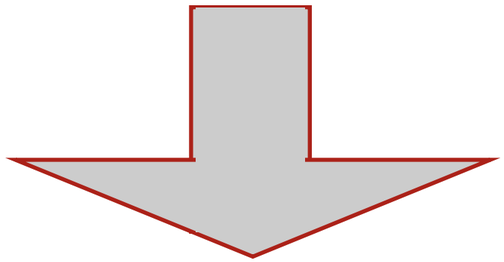
History

- Started out as the Acegi project
  - Acegi 1.0.0 released in May 2006
- Now integrated into the Spring framework as Spring Security
  - www.springsource.org/spring-security
  - Current version: 2.0.4

Features

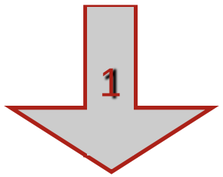
- Spring Security supports the standard Java EE authentication schemes
- Support for remember-me authentication
  - remembering users across sessions
  - uses long-term cookies
- Support for single sign-on redirects
  - redirects user to a central access manager for single sign-on support
- Method invocations can be secured
  - Leverage the advantages of Spring AOP to apply security concepts
- …

High level concepts



Security interceptor

Intercepts calls to make sure nothing goes by unauthorized
Implementation depends on resource type (servlet filter for web modules, aspect for method invocations)



Authentication manager

Validates the provided principal and credentials

Access Decision manager

Checks whether the caller is authorized to perform this action

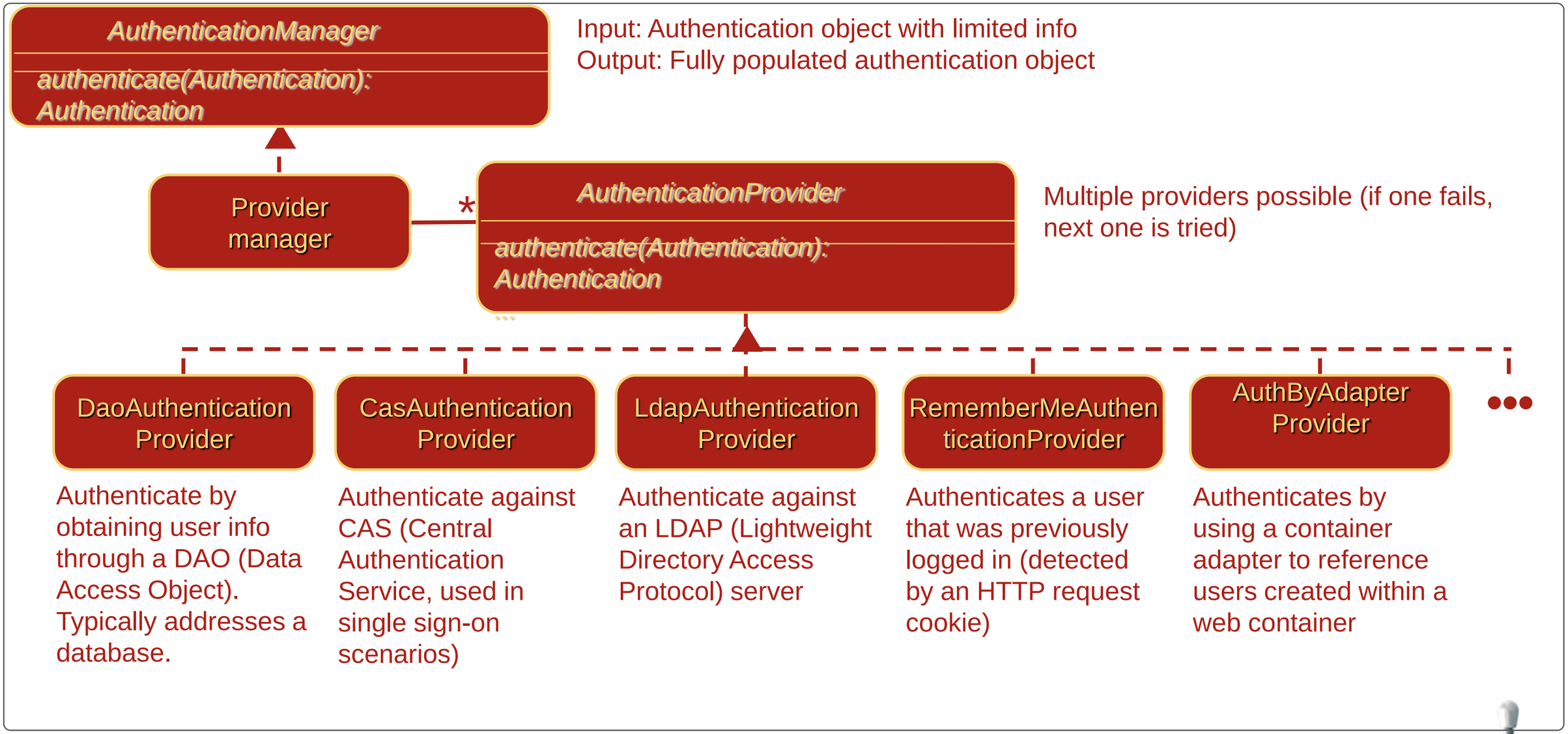
Run-As manager

Optionally replaces the caller's authentication by one that allows access to resources further down the road

After-Invocation manager

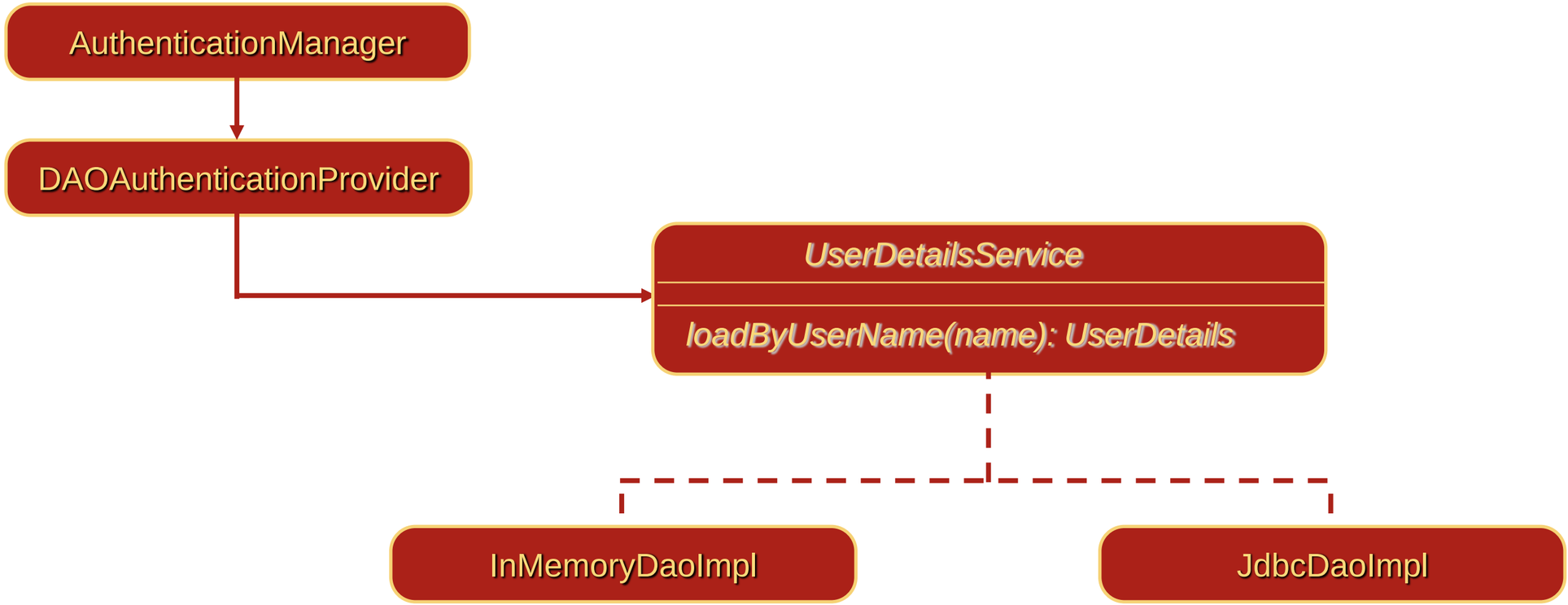
Optionally intervenes in the response being returned (e.g. to mask sensitive data)

Authentication



DAO Authentication provider

- DAOAuthenticationProvider uses a UserDetailsService instance to decide where user info is stored



DAO Authentication provider - in memory

```
<bean id="authenticationManager" class="org.springframework.security.providers.ProviderManager">
        <property name="providers">
                <list>
                        <ref bean="daoAuthenticationProvider" />
                </list>
        </property>
</bean>

<bean id="daoAuthenticationProvider" class="org.springframework.security.providers.dao.DaoAuthenticationProvider">
        <property name="userDetailsService" ref="authenticationDao" />
</bean>

<bean id="authenticationDao" class="org.springframework.security.userdetails.memory.InMemoryDaoImpl">
        <property name="userMap">
                <value>
                        user1=password1,ROLE_ADMINISTRATION
                        user2=password2,ROLE_SALES,ROLE_MARKETING
                        user3=password3,disabled,ROLE_ADMINISTRATION
        </value>
        </property>
</bean>
```

DAO Authentication provider - database

```
<bean id="authenticationManager" class="org.springframework.security.providers.ProviderManager">
        <property name="providers">
                <list>
                        <ref bean="daoAuthenticationProvider" />
                </list>
        </property>
</bean>

<bean id="daoAuthenticationProvider" class="org.springframework.security.providers.dao.DaoAuthenticationProvider">
        <property name="userDetailsService" ref="authenticationDao" />
</bean>

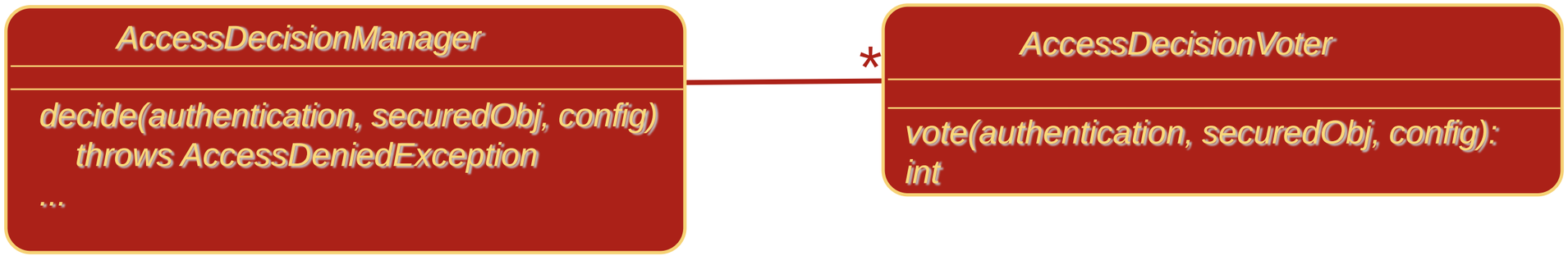
<bean id="authenticationDao" class="org.acegisecurity.userdetails.jdbc.JdbcDaoImpl">
        <property name="dataSource" ref="dataSource" />
</bean>
```

In-memory UserDetailsService **replaced by DAO**-based one

This assumes default queries (can be customized of course)

SELECT username, password, enabled FROM users WHERE username = ?
SELECT username, authority FROM authorities WHERE username = ?

Authorization

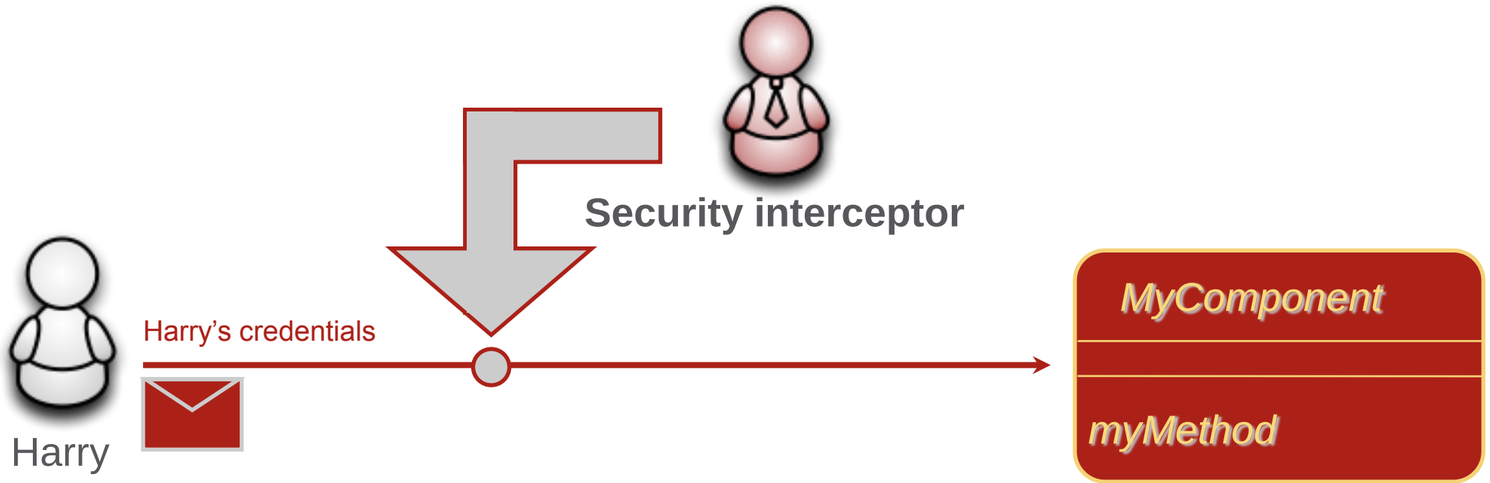


Multiple subclasses provided:
- **AffirmativeBased**: grants access if at least one voter grants access
- **ConsensusBased**: grants access if a consensus of voters grant access
- **UnanimousBased**: grants access if all voters grant access

Possible return values:
ACCESS_GRANTED (1)
ACCESS_ABSTAIN (0)
ACCESS_DENIED (-1)

Authorization - securing method invocations

- Consider a class MyComponent, with a method myMethod.
- The role “admin” is required for invoking myMethod.
- A Security interceptor is typically
  - a servlet filter for authorization of web resource access
  - an aspect for authorization of method access
- In case of the latter:



Authorization - securing method invocations - example

```
<bean id="autoProxyCreator" class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
        <property name="interceptorNames">
                <list>
                        <value>securityInterceptor</value>
                </list>
        </property>
        <property name="beanNames">
                <list>
                        <value>myComponent</value>
                </list>
        </property>
</bean>

<bean id="securityInterceptor" class="org.springframework.security.intercept.method.MethodSecurityInterceptor">
        <property name="authenticationManager">
                <ref bean="authenticationManager" />
        </property>
        <property name="accessDecisionManager">
                <ref bean="accessDecisionManager" />
        </property>
        <property name="objectDefinitionSource">
                <value>
                        be.aca.MyComponent.myMethod*=ROLE_ADMIN
           </value>
        </property>
</bean>
```

generate a securityInterceptor proxy for myComponent

invocation of methods prexifed with myMethod()
requires a ROLE_ADMIN role

JBoss Seam

- Authentication
- Identity Management
- Authorization
- Permission Management
- Extra features

Authentication

- JAAS – based
  - Simplified alternative

```
<security:identity authenticate-
      method="#{authenticator.authenticate}" />
```

```
<div>
 <h:outputLabel for="name" value="Username"/>
 <h:inputText id="name" value="#{credentials.username}"/>
</div>
<div>
 <h:outputLabel for="password" value="Password"/>
 <h:inputSecret id="password" value="#{credentials.password}"/>
</div>
<div>
<h:commandButton value="Login" action="#{identity.login}"/>
</div>
```

```
@Name("authenticator")
public class Authenticator {
   @In EntityManager entityManager;
   @In Credentials credentials;
   @In Identity identity;
   public boolean authenticate() {
      try {
         User user = (User) entityManager.createQuery(
          "from User where username = :u and password = :p"
)
            .setParameter("u", credentials.getUsername())
            .setParameter("p", credentials.getPassword())
            .getSingleResult();
         if (user.getRoles() != null) {
            for (UserRole mr : user.getRoles())
               identity.addRole(mr.getName());
         }
         return true;
      }  catch (NoResultException ex) {
         return false;
      }
   }
}
```

Security

- Authentication
  - Securing pages

```
<pages login-view-id="/login.xhtml">

    <page view-id="/members/*" login-required="true"/>

    <exception class="org.jboss.seam.security.NotLoggedInException">
        <redirect view-id="/login.xhtml">
            <message>You must be logged in to perform this action</message>
        </redirect>
    </exception>

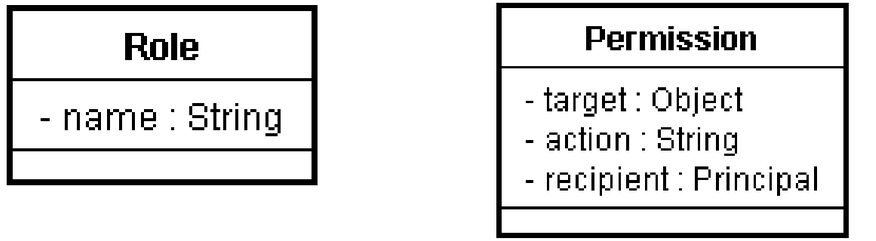
 <event type="org.jboss.seam.security.notLoggedIn">
    <action execute="#{redirect.captureCurrentView}"/>
 </event>

 <event type="org.jboss.seam.security.postAuthenticate">
    <action execute="#{redirect.returnToCapturedView}"/>
 </event>

</pages>
```

Authorization

- Role based access control



- Restrictions can be applied on:
  - Components
  - User interface
  - Pages
  - Entities

Authorization

- Restrictions can be applied on
  - Components
    - @Restrict
    - Typesafe alternative
      - @Insert
      - @Update
      - @Delete
      - @Read
      - @Admin
      - Custom Annotations

```
@Restrict @Name("account")
public class AccountAction {
    @In Account selectedAccount;

    // @Restrict("#{s:hasPermission('account', 'insert')}")
    @Insert
    public void insert() {
      ...
    }

    @Restrict("#{s:hasRole('admin')}")
    public void delete() {
      ...
    }

    @Restrict("#{s:hasPermission(selectedAccount,'modify')}")
    public void modify() {
        selectedAccount.modify();
    }

}
```

Authorization

- User interface
  - Rendered property
    - #{not identity.loggedIn}
    - #{s:hasPermission(event,'delete')
    - #{s:hasRole('admin')}
  - Pages
    - /settings.xhtml:render

```
<page view-id="/settings.xhtml">
    <restrict/>
</page>
```

```
<page view-id="/reports.xhtml">
    <restrict>#{s:hasRole('admin')}</restrict>
</page>
```

Authorization

- Entities
  - ACL-style
  - Read/write/update/delete permission for <entity-classname>:<id>
  - For all operations or only for some via entity lifecycle methods: prePersist, postLoad, preUpdate, preRemove
    - ```
      @PrePersist @Restrict
      public void prePersist() {}
      ```


      ```

Permission Management

- Permissions are revolved using:
  - PermissionResolvers
    - RuleBasePermissionResolver
      - Drools

```
package be.aca.jeeonsteroids;
  import org.jboss.seam.security.permission.PermissionCheck;
  import org.jboss.seam.security.Role;

  rule CanUserDeleteEvent
  when
    c: PermissionCheck(target == "event", action == "delete")
    Role(name == "admin")
  then
    c.grant();
  end
```

    - PersistentPermissionResolver
      - PermissionManager API
      - JPA

Extra Features

- Seam comes with a set of common features to implement your security needs
  - CAPTCHA support
  - RememberMe / AutoLogin
  - Fine grained declarative support for SSL

```
<pages login-view-id="/login.xhtml">

    <page view-id="/login.xhtml" scheme="https"/>
    ...
</pages>
```

  - OpenId integration

Conclusions

- Standard Java EE security can handle common security requirements
  - configured/implemented quite easily
- But...
  - Authentication support may be too limited
  - Authorization support may be too limited
- More elaborate security solutions exist:
  - Spring Security
  - JBoss SEAM
  - Check them out before implementing a DIY-framework
    - all too often complexity is well hidden (e.g. Facelets and Servlet filters)

Questions ?

DELIVER BETTER SOLUTIONS FASTER