argonavis
tecnologia e arte

JAVA EE 7

segurança

JASPIC JAAC JAAS

Helder da Rocha

O texto foi elaborado como material de apoio para treinamentos especializados em linguagem Java e explora assuntos detalhados nas especificações e documentações oficiais sobre o tema, utilizadas como principais fontes. A autoria deste texto é de inteira responsabilidade do seu autor, que o escreveu independentemente com finalidade educativa e não tem qualquer relação com a Oracle.

O código-fonte relacionado aos tópicos abordados neste material estão em:

`github.com/helderdarocha/javaee7-course`
`github.com/helderdarocha/CursoJavaEE_Exercicios`
`github.com/helderdarocha/ExercicioMinicursoJMS`
`github.com/helderdarocha/JavaEE7SecurityExamples`

www.argonavis.com.br

R672p Rocha, Helder Lima Santos da, 1968-

Programação de aplicações Java EE usando Glassfish e WildFly.

360p. 21cm x 29.7cm. PDF.

Documento criado em 16 de outubro de 2013.

Atualizado e ampliado entre setembro e dezembro de 2016.

Volumes (independentes): *1: Introdução, 2: Servlets, 3: CDI, 4: JPA, 5: EJB, 6: SOAP, 7: REST, 8: JSF, 9: JMS, 10: Segurança, 11: Exercícios*.

1. Java (*Linguagem de programação de computadores*). 2. Java EE (*Linguagem de programação de computadores*). 3. Computação distribuída (*Ciência da Computação*). I. Título.

CDD 005.13'3

Capítulo 10: Segurança

1 Introdução

1.1 O que é Segurança?

Segurança possui dois significados. Em inglês esses significados são representados por palavras diferentes:

- **Safety** é a proteção contra danos não-intencionais: bugs em geral e vulnerabilidades intrínsecas do sistema (threads, ambiente, recursos externos, etc.) *Safety* de uma aplicação depende de fatores como qualidade da plataforma, adoção de boas práticas de programação, realização de testes, validação, monitoração, logging, checksums, etc.
- **Security** é a proteção contra danos intencionais: ataques em geral, acesso e manipulação de dados protegidos, danos ao sistema ou a terceiros. Para garantir *Security* utiliza-se recursos como autenticação, autorização, criptografia, auditoria, filtros, auditoria, hashes, etc.

Neste tutorial iremos discutir Segurança com o significado Security.

1.2 Terminologia

Existem vários termos relacionados à segurança que precisam ser definidos antes de usar a API. Os principais termos são descritos a seguir.

1.2.1 Identificação: Identity

"Quem é você?" Segurança depende da **identificação** dos usuários (pessoas e sistemas). A identificação pode ser um código, um nome, um avatar, um endereço IP, um token que identifique uma entidade de forma unívoca.

1.2.2 Autenticação, autoridades, credenciais, integridade

"Como provar que você é mesmo você?" Uma identidade não tem valor se não for possível provar, dentro de um determinado contexto (**autoridade**), que ela é autêntica. A **autenticidade** de uma identidade depende do parecer dado por uma "autoridade de confiança". Essa autoridade não precisa ser universal, mas deve ser aceita por todos os participantes que dependem da identidade.

O processo de autenticação geralmente envolve a apresentação de **credenciais**: algo que você **possui** (ex: documento), **é** (ex: biometria) ou **sabe** (ex: senha).

A autenticidade também depende da **integridade** das informações. Deve haver uma garantia que os dados transmitidos não foram alterados pelo caminho (o que poderia, potencialmente, forjar uma identidade ou credenciais).

Dentro de uma aplicação Java EE, um usuário autenticado é chamado de **principal**.

1.2.3 Domínio = Identity Store, Realm, Authentication Provider

O **domínio** (security domain) é a **autoridade** que administra usuários e grupos e que determina escopo de políticas de segurança (ex: LDAP, banco de dados, banco de certificados). Também é chamado de *Identity Store* (Repositório de Identidades), *Realm* (Reino), *Zone*, *Region*, *Authentication Provider*, *Login Module*, dependendo do fabricante (esses nomes nem sempre são sinônimos mas geralmente referem-se à mesma coisa: uma **autoridade** que mantém uma coleção de identidades e credenciais.)

A especificação Java EE 7 não abrange domínios de segurança. Utiliza apenas o *nome* do domínio para *selecionar o escopo da autenticação via HTTP*. A configuração dos domínios, definição de usuários, grupos e senhas é realizada de forma proprietária por cada servidor.

1.2.4 Usuários (pessoas, máquinas, sistemas) e Principais

A criação e autenticação de **usuários** não faz parte da especificação Java EE 7. Isto é responsabilidade de um *repositório de identidades* que é configurado usando *ferramentas proprietárias*. É diferente para cada servidor. Java EE apenas padroniza a *escolha* do método de autenticação em um domínio (se é feito através de cabeçalhos HTTP, troca de certificados, sessão, etc.) Java EE não trabalha com o conceito de "usuário", mas com "principal".

Um **java.security.Principal** representa uma *entidade* (usuário, grupo, sistema) *autenticada*. É um objeto que contém sua **identificação** (nome/certificado) verificada através de **credenciais**. Tem um foco diferente em Java SE e Java EE.

- Em Java SE (JAAS) um **Subject** representa um usuário, que pode ter uma coleção de identidades (**Principal**)
- Em APIs Java EE um **Principal** representa um usuário, que pode assumir uma coleção de papéis (**Roles**). A autorização de acesso é feito para papéis e não usuários.

1.2.5 Autorização

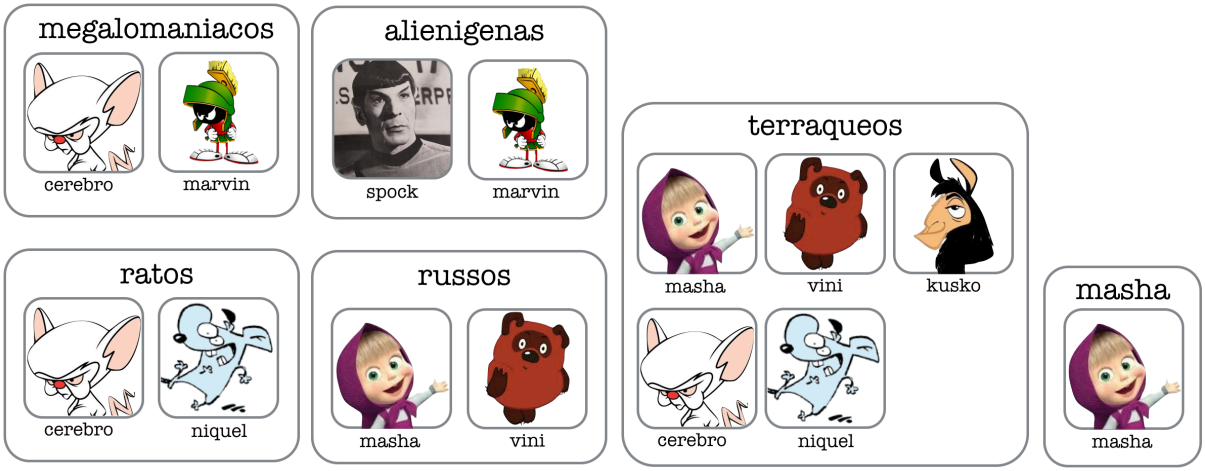
A **autorização** envolve todo tipo de restrição de acesso imposta a um principal ou role. Responde a perguntas como: **Quem** pode acessar? **Quais** recursos? Quais URLs, componentes, classes, métodos, blocos de código, tabelas, registros, fragmentos de XML, imagens, arquivos, pastas, aplicações? Que **operações** podem executar? GET, POST, ler, gravar, criar, remover? Quais **condições**? ″das 9 as 17h″, ″enquanto houver tokens″, ″apenas duas vezes″.

Java EE suporta controle de acesso com base em **identidade** (roles, principais), **recursos** (URLs, caminhos) e **operações** (métodos Java, métodos HTTP). Não há suporte padrão para controle de acesso com base em condições.

1.2.6 Grupos (coleções de usuários) e Roles (permissões)

Usuários podem pertencer a **grupos**. Mas usuários não fazem parte do escopo do Java EE, e grupos também não. Este é mais um conceito é dependente de plataforma. Em Java existem **roles** (papéis), que são **permissões de acesso** a recursos protegidos, e **principais**, que representam grupos e usuários autenticados que **devem ser mapeados a roles**. Java EE não especifica como esse mapeamento é realizado (também é proprietário!) Consequentemente, a definição de grupos, roles, realms, etc. varia entre fornecedores.

Exemplos de grupos de usuários. Um usuário geralmente pode pertencer a vários grupos.



2 Segurança em Java EE 7

Os serviços básicos de segurança em Java EE são **autenticação**, **autorização** e **integridade dos dados**. APIs programáticas e declarativas permitem configurar mecanismos de identificação e autenticação de usuários, controle de acesso a dados restritos, controle de

execução de operações protegidas, e garantias de integridade e confidencialidade para informações que trafegam em rede.

Java EE utiliza-se dos mecanismos de segurança presentes no Java SE, mas também possui APIs que abstraem conceitos de segurança próprios de ambientes distribuídos.

As figuras abaixo comparam os recursos de segurança em Java SE e Java EE:

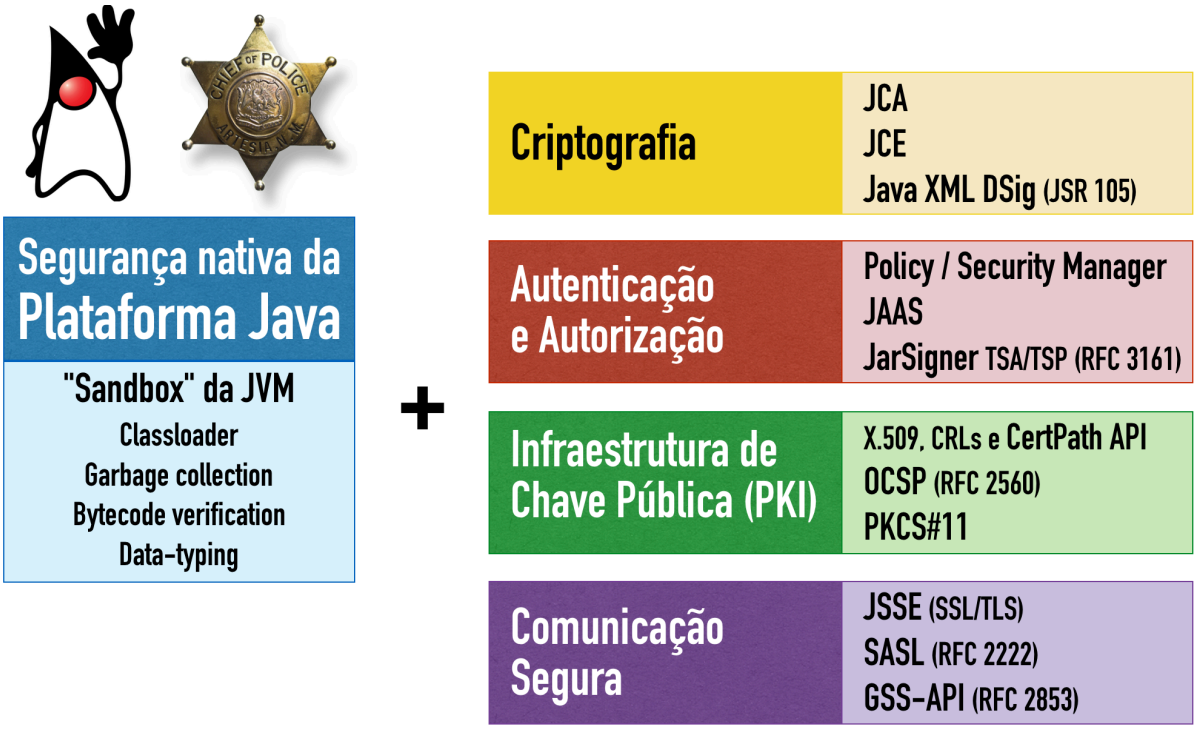


Figura 2.1 Arquitetura de segurança no Java SE

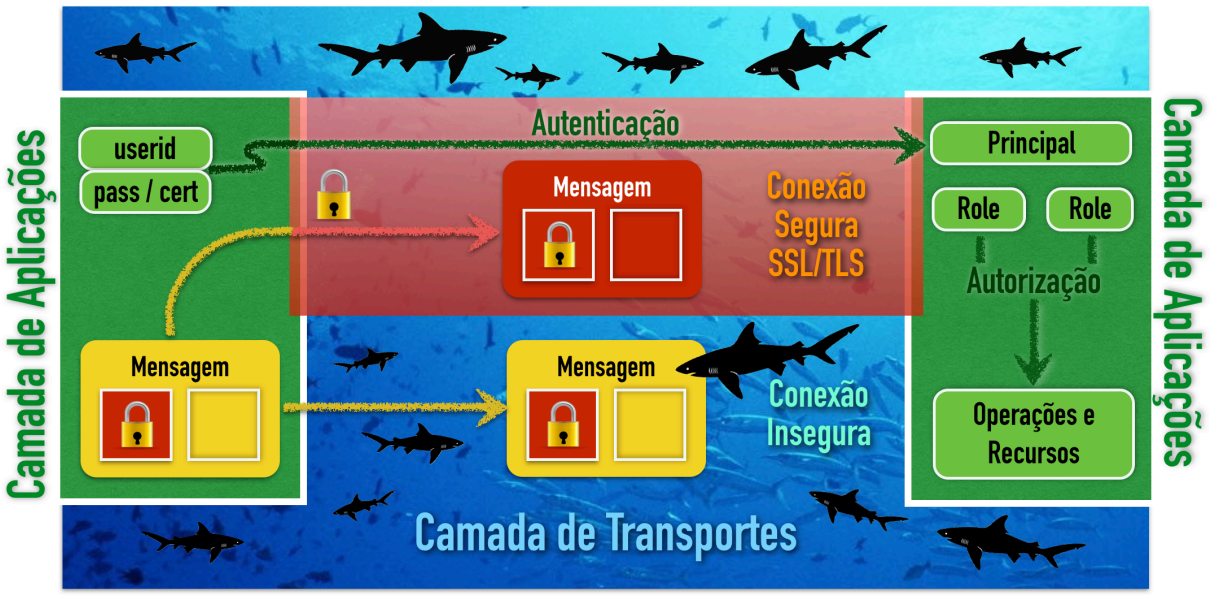


Figura 2.2 Arquitetura de segurança no Java EE

2.1 Camada Web

Na camada Web é possível configurar **autenticação**, realizada de acordo com a especificação HTTP, e **autorização declarativa**, de acordo com a especificação Java EE.

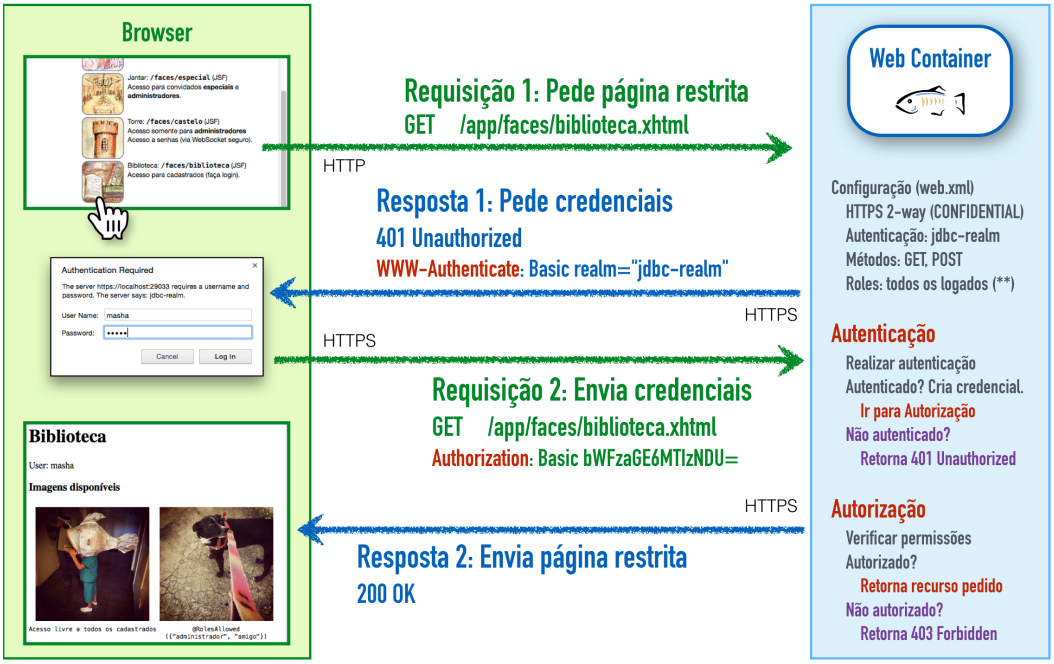


Figura 2.3 Autenticação e autorização na camada Web

2.2 Camada EJB

A camada EJB não suporta autenticação e depende de camadas anteriores (ex: Web) para obter roles e principais usados para controlar acesso (**autorização**) a métodos e recursos.

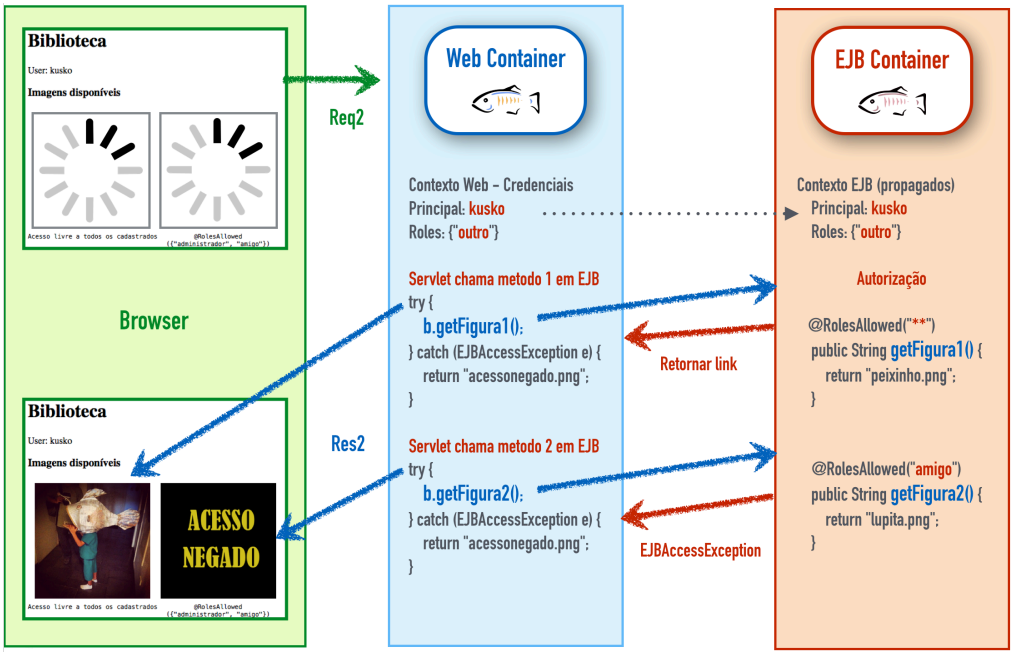


Figura 2.4 Autorização na Camada EJB

2.3 Integridade das informações

Pode-se proteger recursos em Java EE configurando a necessidade do uso de HTTP + camada SSL/TLS, que que atua na camada de transporte fornecendo mecanismos de autenticação, integridade e confidencialidade. Esses mecanismos protegem contra os principais riscos de transferir dados pela rede: a **autenticação** protege contra o risco de não saber se outro é quem diz que é, a **integridade** protege contra o risco da mensagem ser interceptada e alterada, e a **confidencialidade** protege contra o risco da mensagem ser interceptada e lida.

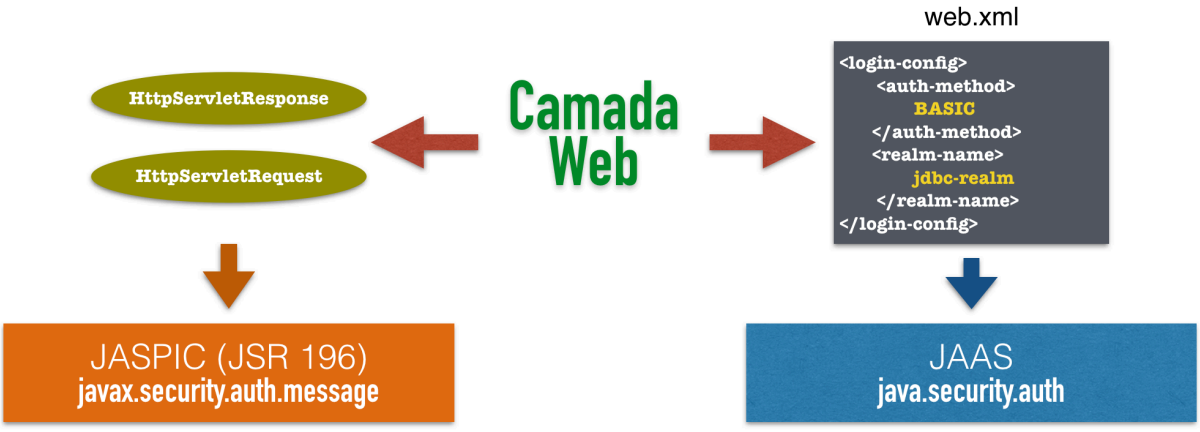
SSL/TLS utiliza-se de vários mecanismos de criptografia: hashes, criptografia de chave pública (assimétrica) e criptografia de chave secreta (simétrica). Também garante proteção à camada de mensagens, mas apenas durante a duração da conexão.

3 Autenticação

Java EE oferece duas alternativas de configuração de autenticação: uma API **declarativa**, para a camada Web, e uma API **programática** (JASPIC). A maior parte dos servidores oferece apenas a primeira opção, implementada através de módulos proprietários.

A API declarativa é usada na camada Web (web.xml) para selecionar o **domínio** ou *realm* (identity store que depende de recursos proprietários para configuração) e o **método** de autenticação. São quatro métodos disponíveis: BASIC, FORM, DIGEST, CLIENT.

A API programática (**JASPIC**) é usada para criar **módulos** de autenticação. É a solução ideal para usar autenticação OAuth, SSO e via redes sociais utilizando padrões Java EE, mas, apesar de ser a API oficial, poucos fabricantes de servidores de aplicação oferecem uma implementação (a maioria oferece apenas APIs e ferramentas proprietárias baseadas em **JAAS**, o que torna a realização da autenticação é *dependente* de fabricante.)



3.1 Autenticação Web

Os quatro métodos de autenticação via HTTP estão descritos nos diagramas abaixo:

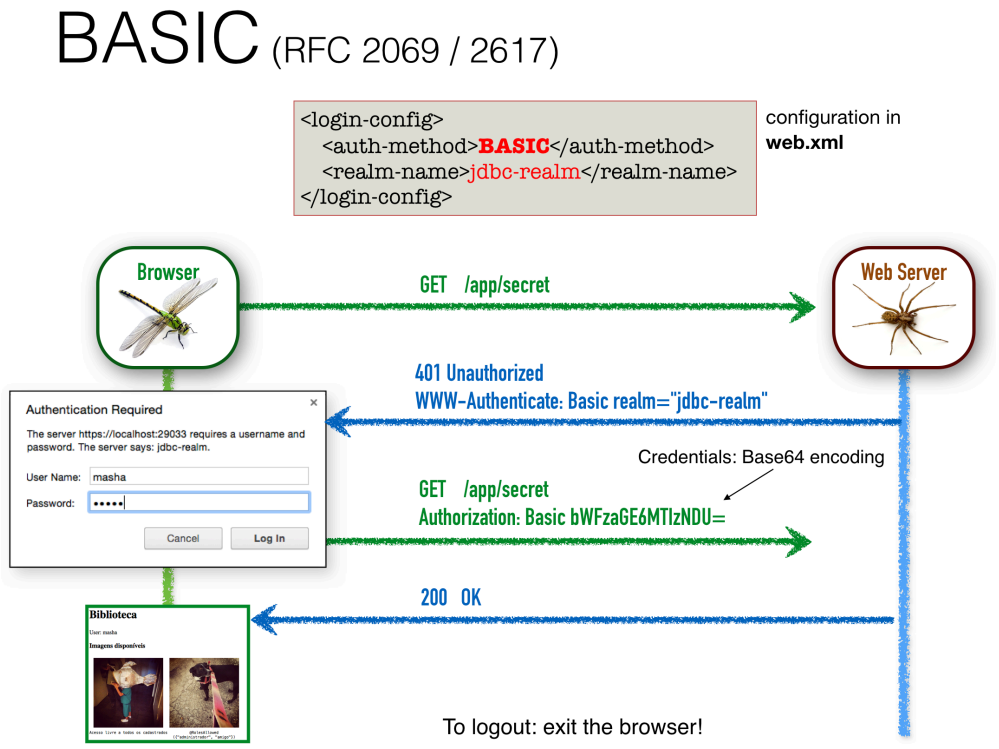


Figura 3.1 Método BASIC utiliza cabeçalhos HTTP pra trocar dados com criptografia fraca.

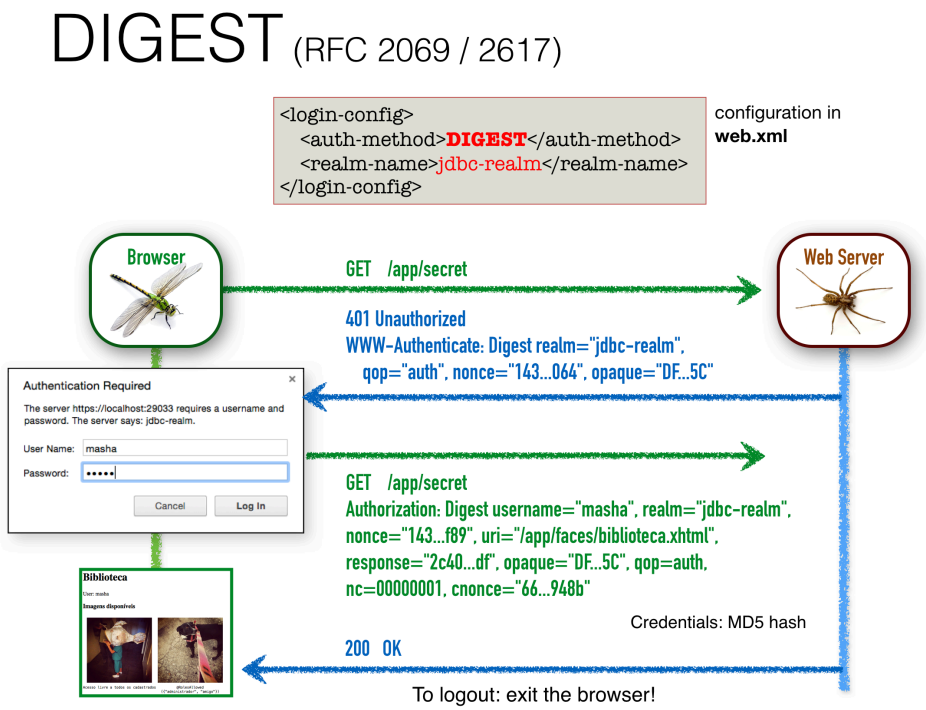


Figura 3.2 Método DIGEST utiliza hash assimétrico MD5 que é mais seguro.

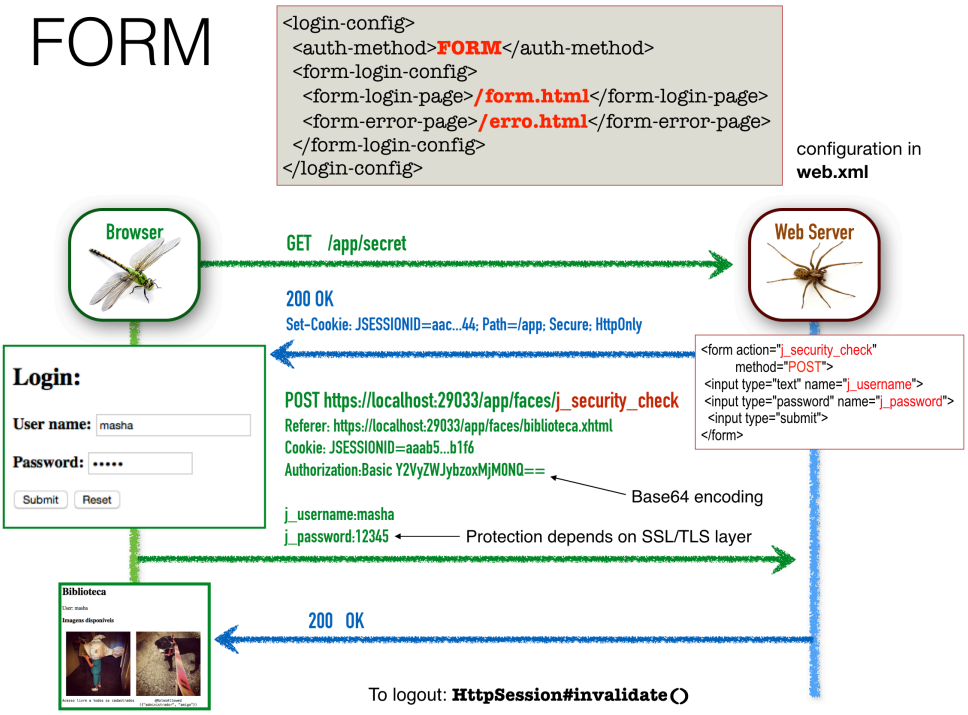


Figura 3.3 Método FORM é como BASIC (utiliza criptografia fraca e cabeçalhos HTTP) mas instrui o cliente a usar uma página para entrada de credenciais (em vez de abrir uma janela padrão).

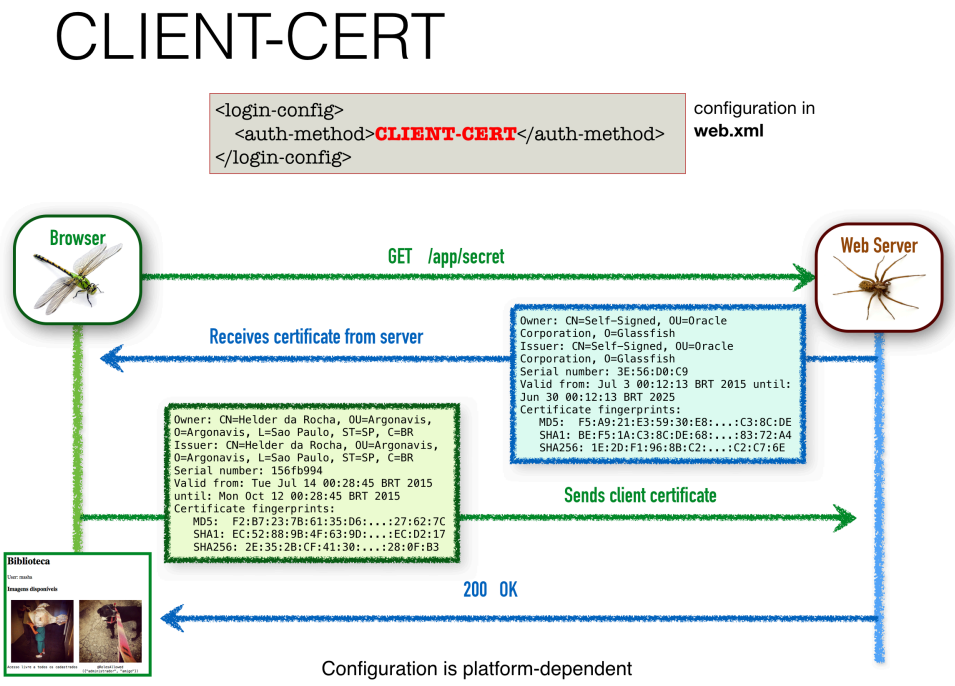


Figura 3.4 Método CLIENT-CERT realiza autenticação através da troca de certificados (que precisam ser gerados anteriormente)

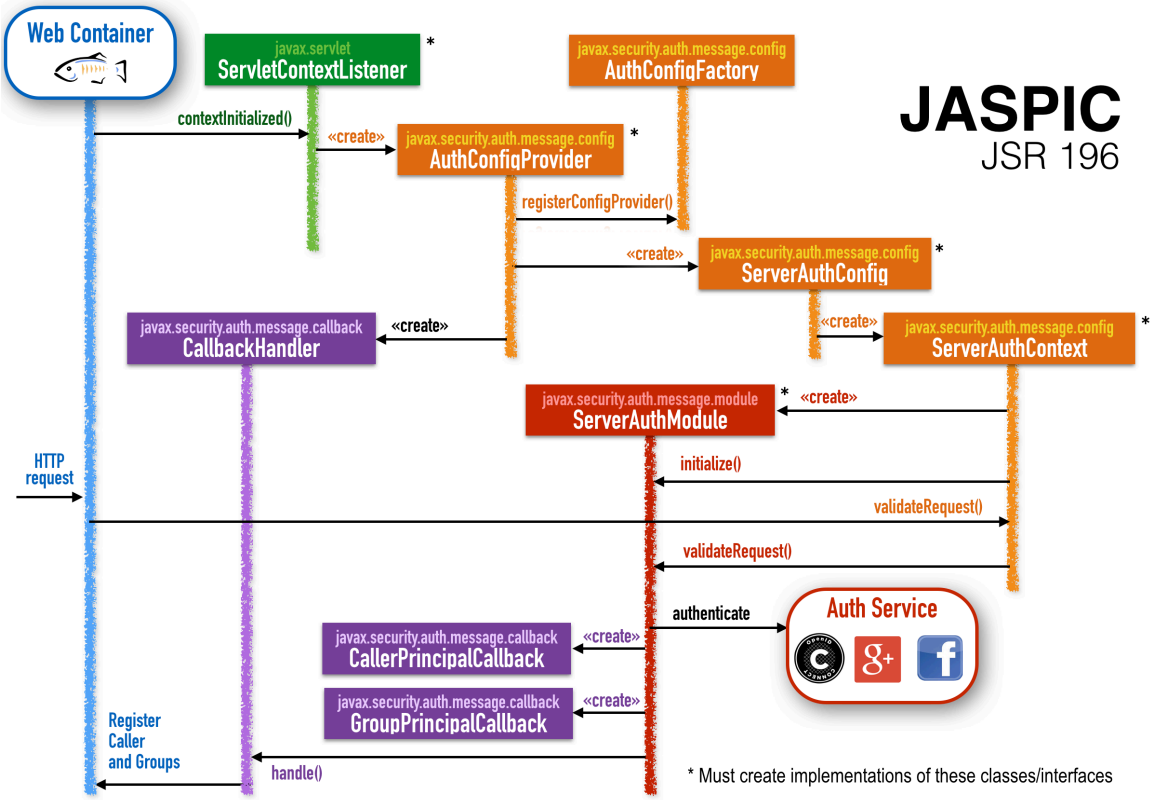
3.2 JASPIC

JASPIC é uma API padrão Java EE que oferece mais flexibilidade que os métodos HTTP por permitir a autenticação através do HttpServletRequest. É uma API programática. Pode ser

usada para construir filtros de autenticação usando serviços externos (redes sociais, OAuth, etc.) Os métodos de autenticação (em HttpServletRequest) são:

- String **getAuthType**(): retorna string com mecanismo de autenticação (BASIC, DIGEST, FORM, CLIENT-CERT) ou null
- void **authenticate**(HttpServletResponse response): autenticar usando mecanismo configurado
- void **login**(String nome, String senha): autenticar usando login e senha
- void **logout**(): faz com que getRemoteUser(), getCallerPrincipal() e getAuthType() retorne null.

No Java EE 7 não há uma forma simples de usar JASPIC. Para construir uma solução com usar autenticação via HttpServletRequest é necessário implementar e configurar várias interfaces. O diagrama abaixo ilustra o processo de autenticação. Todas as classes e interfaces marcadas com asterisco (*) abaixo precisam ser implementadas:



3.3 Autenticação em WebServices

A autenticação em WebServices é a mesma usada nas aplicações Web, ou seja, os métodos HTTP (BASIC, FORM, etc.) ou JASPIC podem ser usados para configurar a autenticação de serviços SOAP ou REST.

3.3.1 JAX-WS

O JAX-WS contém algumas propriedades padrão adicionais que permitem o envio de credenciais via mensagens SOAP (isto implementa padrões SOAP). As propriedades são USERNAME_PROPERTY e PASSWORD_PROPERTY e estão disponíveis em javax.xml.ws.BindingProvider.

O código abaixo mostra um cliente usando essas propriedades para realizar autenticação:

```
DungeonWebService service = new DungeonWebService();
Dungeon port = service.getDungeonPort();

BindingProvider bp = (BindingProvider)port;
bp.getRequestContext().put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
"https://localhost:29033:/app/dungeon");

bp.getRequestContext().put(BindingProvider.USERNAME_PROPERTY, "masha");
bp.getRequestContext().put(BindingProvider.PASSWORD_PROPERTY, "12345");
String secret = port.secretMessage();
```

No serviço é possível recuperar as credenciais através do MessageContext:

```
@WebService(serviceName = "dungeon")
public class DungeonWebService {

    @Resource WebServiceContext ctx;

    @WebMethod public String secretMessage() {
        MessageContext mctx = ctx.getMessageContext();
        String user = mctx.get(BindingProvider.USERNAME_PROPERTY);
        String pass = mctx.get(BindingProvider.PASSWORD_PROPERTY);
```

3.4 Segurança em WebSockets

A especificação de WebSockets não define um mecanismo próprio de autenticação, mas o *handshake*, que é o processo de inicialização, é feito através de HTTP ou HTTPS, podendo aproveitar o contexto de segurança definido na configuração Web existente (métodos BASIC, FORM, etc. ou JASPIC).

Mas é possível garantir transporte seguro (SSL) em conexões WebSocket através da seleção do protocolo da URL (ws – sem SSL, ou wss – com SSL). Se houver tentativa de realizar uma conexão ws em um WebSocket que foi inicializado com handshake HTTPS, um erro de segurança é lançado em JavaScript. Já uma tentativa de usar wss em um WebSocket iniciado com HTTP (sem SSL) provoca um erro HTTP 401 Unauthorized.

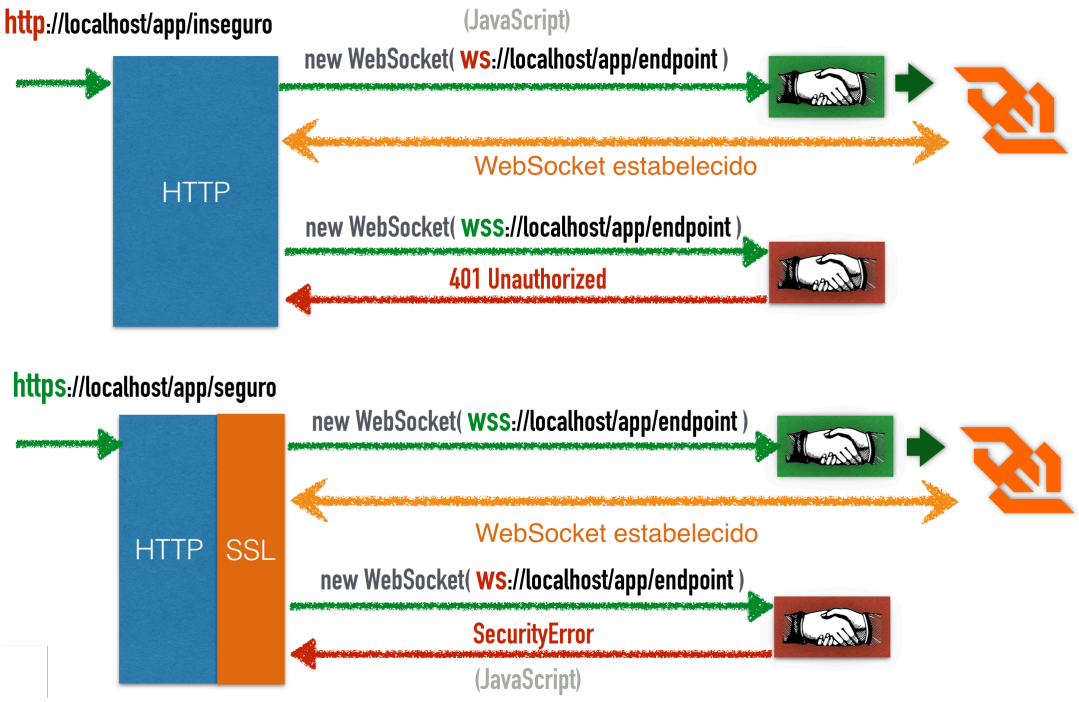


Figura 3.5 WebSocket e comunicação SSL

4 Autorização

A autorização em Java EE é baseada em roles (papéis, permissões). É preferencialmente declarativa, relativamente fácil de usar e, diferentemente da autenticação, é 100% Java EE.

A configuração declarativa da autorização pode ser realizada através de anotações e/ou deployment descriptors XML, ou através de APIs programáticas. A autorização é diferente para cada tecnologia Java EE usada (a API JACC, que seria uma solução para este problema não é implementada pela maior parte dos servidores de aplicação Java EE 7):

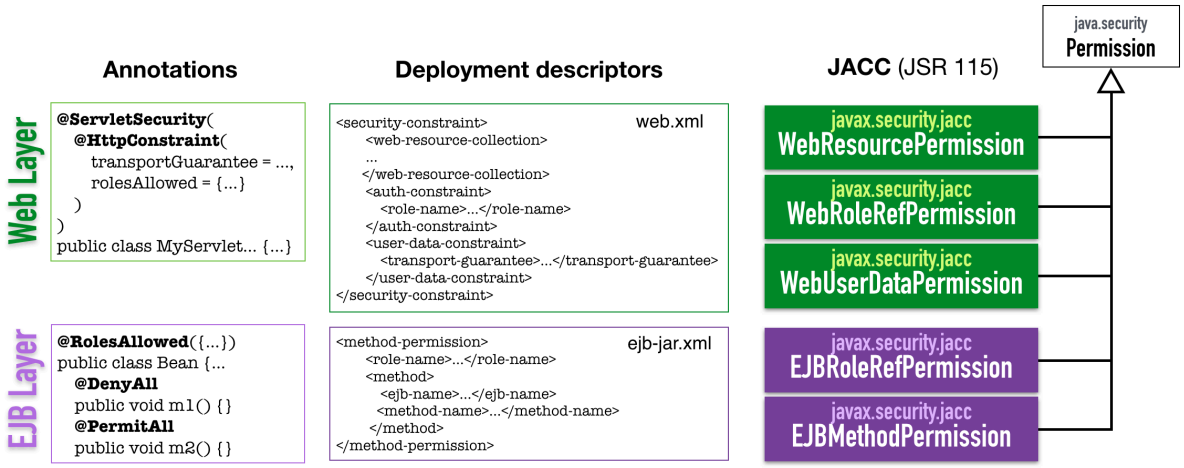


Figura 4.1 Autorização em Java EE

4.1 Autorização em aplicações Web

Em aplicações Web é possível controlar o acesso através de URIs. O controle declarativo é sempre baseado em permissões (roles) que são declarados no web.xml usando <security-role> (ou via anotações nos WebServlets).

```
<web-app> ...
        <security-role>
                <role-name>administrador</role-name>
        </security-role>

        <security-role>
                <role-name>amigo</role-name>
        </security-role>

        <security-role>
                <role-name>especial</role-name>
        </security-role>
    ...
</web-app>
```

A configuração dos roles precisa ser feita previamente e é dependente de plataforma. O mapeamento dos roles com usuários/grupos é dependente de servidor. Em alguns servidores, eles são automaticamente mapeados a grupos. Em outros, eles dependem de uma configuração proprietária (arquivos como glassfish-web.xml, jboss-web.xml, etc.) A ilustração abaixo mostra um arquivo de configuração para o Glassfish onde são mapeados grupos e usuários a roles distintos.

Exemplo de mapeamentos possíveis em **Glassfish 4.1**

Aplicação: JavaEESecurity.war (veja código-fonte)

```
<glassfish-web-app>
    <context-root>JavaEESecurity</context-root>
    <security-role-mapping>
        <role-name>administrador</role-name>
        <principal-name>cerebro</principal-name>
        <group-name>alienigenas</group-name>
    </security-role-mapping>
    <security-role-mapping>
        <role-name>especial</role-name>
        <principal-name>vini</principal-name>
        <principal-name>masha</principal-name>
    </security-role-mapping>
    <security-role-mapping>
        <role-name>amigo</role-name>
        <principal-name>masha</principal-name>
        <principal-name>vini</principal-name>
        <principal-name>niquel</principal-name>
    </security-role-mapping>
    <security-role-mapping>
        <role-name>outro</role-name>
        <principal-name>kusko</principal-name>
    </security-role-mapping>
</glassfish-web-app>
```

/WEB-INF/glassfish-web.xml

4.1.1 Security constraints

O security constraint determina restrições de acesso a partes de um site. Um bloco <security-constraint> contém três partes:

- Um ou mais <web-resource-collection>
- Pode conter um <auth-constraint> (lista de roles)
- Pode conter um <user-data-constraint> (SSL/TLS).

```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>Área restrita</web-resource-name>
                <url-pattern>/secreto/*</url-pattern>
      </web-resource-collection>
        <auth-constraint>
                <role-name>administrador</role-name>
       </auth-constraint>
</security-constraint>
```

4.1.2 Web resource collection

O Web resource collection grupa recursos e operações protegidas através da combinação de padrões de URL e métodos HTTP.

```
<security-constraint>
        <web-resource-collection>
                <web-resource-name>Área restrita</web-resource-name>
                <url-pattern>/secreto/*</url-pattern>
                <url-pattern>/faces/castelo.xhtml</url-pattern>
                <http-method>GET</http-method>
                <http-method>POST</http-method>
        </web-resource-collection>
    ...
</security-constraint>
```

Métodos HTTP não informados são restritos, mas métodos não informados estão descobertos. Use <deny-uncovered-http-methods/> ou outro bloco negando acesso a todos exceto <http-method-omission>

4.1.3 Authorization constraint

Através de <auth-constraint>, uma <web-resource-collection> é acessível apenas aos <role-name> declarados. Se não houver <auth-constraint> o recurso pode ser acessado por qualquer cliente, sem nenhum controle de acesso (livre). Se o <auth-constraint /> for vazio, ninguém tem acesso.

```
<web-app>
        ...
        <security-constraint>
                <web-resource-collection> ... </web-resource-collection>
                <web-resource-collection> ... </web-resource-collection>
                <auth-constraint>
                        <role-name>administrador</role-name>
                        <role-name>especial</role-name>
                </auth-constraint>
        </security-constraint>
                ...
        <security-constraint> ....</security-constraint>
</web-app>
```

4.1.4 Transport guarantee

A configuração <transport-guarantee> de <user-data-constraint> declara as garantias mínimas para comunicação segura SSL/TLS:

- NONE: (ou ausente) não garante comunicação segura
- INTEGRAL: proteção integral, autenticação no servidor
- CONFIDENTIAL: proteção integral, autenticação: cliente e servidor

```
<web-app> ...
        <security-constraint>
                ...
                <user-data-constraint>
                        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
                </user-data-constraint>
        </security-constraint>

        <security-constraint>
                ...
                <user-data-constraint>
                        <transport-guarantee>INTEGRAL</transport-guarantee>
                </user-data-constraint>
        </security-constraint>
</web-app>
```

4.1.5 Servlet 3.x @ServletSecurity

No Servlet 3.x é possível realizar as mesmas configurações no código do servlet usando as anotações @ServletSecurity e @HttpConstraint. Um @ServletSecurity pode conter um ou mais @HttpConstraint. Cada HttpConstraint define os roles que podem usar o servlet

(atributo rolesAllowed) e as garantias do transporte (atributo transportGuarantee), que pode ser CONFIDENTIAL, INTEGRAL ou NONE.

```
@WebServlet(name = "ServletSecreto", urlPatterns = {"/secreto"})
@ServletSecurity(
    @HttpConstraint( transportGuarantee = TransportGuarantee.CONFIDENTIAL,
                     rolesAllowed = {"amigo", "administrador"}
    )
)
public class ServletSecreto extends HttpServlet { ... }

@WebServlet(name = "LabirintoServlet", urlPatterns = {"/labirinto"})
@ServletSecurity(httpMethodConstraints={
    @HttpMethodConstraint("GET"),
    @HttpMethodConstraint(value="POST", rolesAllowed={"outro"}),
    @HttpMethodConstraint(
            value="TRACE",
            transportGuarantee = TransportGuarantee.NONE,
            rolesAllowed = {"amigo", "administrador"}
    )
})
public class LabirintoServlet extends HttpServlet { ... }
```

4.1.6 Autorização com HTTPServletRequest

O controle programático da autorização é possível através da API básica que consiste de um método para obter o Principal e outro para testar se o usuário autenticado faz parte do role.

- **getUserPrincipal()**: obtém java.security.Principal (getName() obtém nome)
- **isUserInRole("role")**: testa se o usuário autenticado faz parte do role

```
String loggedUser = request.getUserPrincipal().getName();

if( !request.isUserInRole("administrador")) {
       if (!loggedUser.equals("cerebro")) {
              throw new SecurityException("...");
       }
}
```

4.2 Autorização em JSF e CDI

Em aplicações JSF é possível acessar a API básica usando Expression Language:

```
<h:panelGroup rendered="#{request.userPrincipal == 'masha'}">
    <p>Conteúdo visto pelo usuário masha</p>
</h:panelGroup>
<h:panelGroup rendered="#{request.isUserInRole('especial')}">
```

```
    <p>Conteúdo visto por quem possui o role especial</p>
</h:panelGroup>
```

Se o código for executado dentro de um Managed Bean, pode-se usar ExternalContext para chamar os mesmos métodos:

```
ExternalContext ctx = FacesContext.getCurrentInstance().getExternalContext();
boolean inRole  = ctx.isUserInRole("role");
String user     = ctx.getUserPrincipal().getName();
```

Ou ainda injetar o Principal via CDI:

```
@Inject java.security.Principal;
```

4.3 Autorização em WebSockets

Contexto de segurança em javax.websocket.Session retorna apenas o java.security.Principal.

```
@ServerEndpoint("/secretpoint")
public class SecretEndPoint {
    @OnMessage
    public void onMessage(String message, Session session) {
        String user = session.getUserPrincipal().getName();
        ...
    }
}
```

É possível redefinir o handshake do WebSocket para passar outros dados de contexto via HttpServletRequest.

4.4 Autorização em EJB

Em componentes Java EE, roles são declarados usando @DeclareRoles. Roles declarados devem estar mapeados a usuários ou grupos no servidor de aplicações. A autorização é realizada através de três anotações:

- **@RolesAllowed** é usado em classes e métodos relaciona quem pode usar a classe e método
- **@PermitAll** libera acesso a todos
- **@DenyAll** restringe acesso a todos

O exemplo abaixo ilustra o uso dessas anotações:

```
@DeclareRoles({"amigo", "administrador"})
@Stateless public class ImagemBean {
     @RolesAllowed("administrador")
     public void removerUsuario(Usuario u) {  ... }
```

```
    @RolesAllowed("amigo")
    public List<Usuario> listaDeUsuarios() { ... }

    @PermitAll
    public String getInfo() { ... }

    @RolesAllowed("**")
    public String getFreeStuff() { ... }

    @DenyAll
    public String loggingData() { ... }
}
```

4.4.1 Segurança em ejb-jar.xml

A mesma configuração é possível em XML, mas é raramente usado por desenvolvedores, já que toda a segurança pode ser configurada por anotações.

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee" ... >
       <enterprise-beans>
               <session>
                       <ejb-name>testeBean</ejb-name>
                       <ejb-class>testebean.TesteBean</ejb-class>
                       <session-type>Stateless</session-type>
                       <transaction-type>Container</transaction-type>
              </session>
        </enterprise-beans>
        <assembly-descriptor>
               <security-role>
                       <role-name>administrador</role-name>
               </security-role>
               <method-permission>
                      <role-name>administrador</role-name>
                      <method>
                            <ejb-name>testeBean</ejb-name>
                           <method-name>*</method-name>
                      </method>
               </method-permission>          ...
        </assembly-descriptor>
</ejb-jar>
```

4.4.2 Propagação de identidade

A anotação

```
@RunAs("role"):
```

Declarada antes da declaração de classe permite que o componente execute usando um role específico, que ele tenha um role mesmo que não tenha sido propagado pelo container cliente.

```
@MessageDriven
@RunAs("administrador")
public class TestEJB {  ... }
```

Isto também pode ser configurado em ejb-jar.xml:

```
<message-driven>
     <ejb-name>TestEJB</ejb-name>
     <ejb-class>br.com.ejb.examples.TestEJB</ejb-class>
    <security-identity>
            <run-as>
                 <role-name>administrador</role-name>
            </run-as>
      </security-identity>
</message-driven>
```

4.4.3 Autorização com EJBContext

Uma API programática de autorização com dois métodos está disponível através dos métodos de javax.ejb.EJBContext. Esses métodos são herdados por SessionContext e MessageDrivenContext (podem ser usados em Session Beans e Message-Driven Beans).

Os métodos são similares aos métodos usados no contexto Web, mas têm nomes diferentes. O Caller no contexto EJB é o User no contexto Web. O efeito é o mesmo:

```
String loggedUser = ctx.getCallerPrincipal().getName();

if( !ctx.isCallerInRole("administrador")) {
       if (!loggedUser.equals("cerebro")) {
              throw new EJBAccessException("...");
       }
}
```

4.5 Autorização em Web Services

Para WebServices REST e SOAP, os mecanismos *declarativos* de autorização aproveitam a estrutura do contexto Web (<security-constraint>), que pode ser usado para controlar acesso a URIs, mas cada API possui contextos com APIs programáticas distintas.

4.5.1 API de segurança em JAX-RS

Há uma API programática de segurança contendo alguns métodos em javax.ws.rs.core.SecurityContext

- **getAuthenticationScheme()**: Retorna o string contendo o esquema de autenticação (mesmo que HttpServletRequest#getAuthType()).
- **isSecure()**:Retorna true se a requisição foi feita em canal seguro.
- **getUserPrincipal()**: Retorna java.security.Principal com usuário autenticado.
- **isUserInRole(String role)**: Retorna true se usuário autenticado pertence ao role.

Resources JAX-RS implementados como EJBs podem usar mesmos mecanismos de autorização declarativos:

```
@Stateless @Path("usuario")
@RunAs("administrador")
public class UsuarioFacadeREST {

    @DenyAll
    @POST
    @Consumes({"application/xml", "application/json"})
    public void create(Usuario entity) { ...
    }

    @RolesAllowed({"administrador"})
    @PUT @Path("{id}")
    @Consumes({"application/xml", "application/json"})
    public void edit(@PathParam("id") Integer id, Usuario entity) { ...
    } ...
}
```

4.5.2 API de segurança em JAX-WS

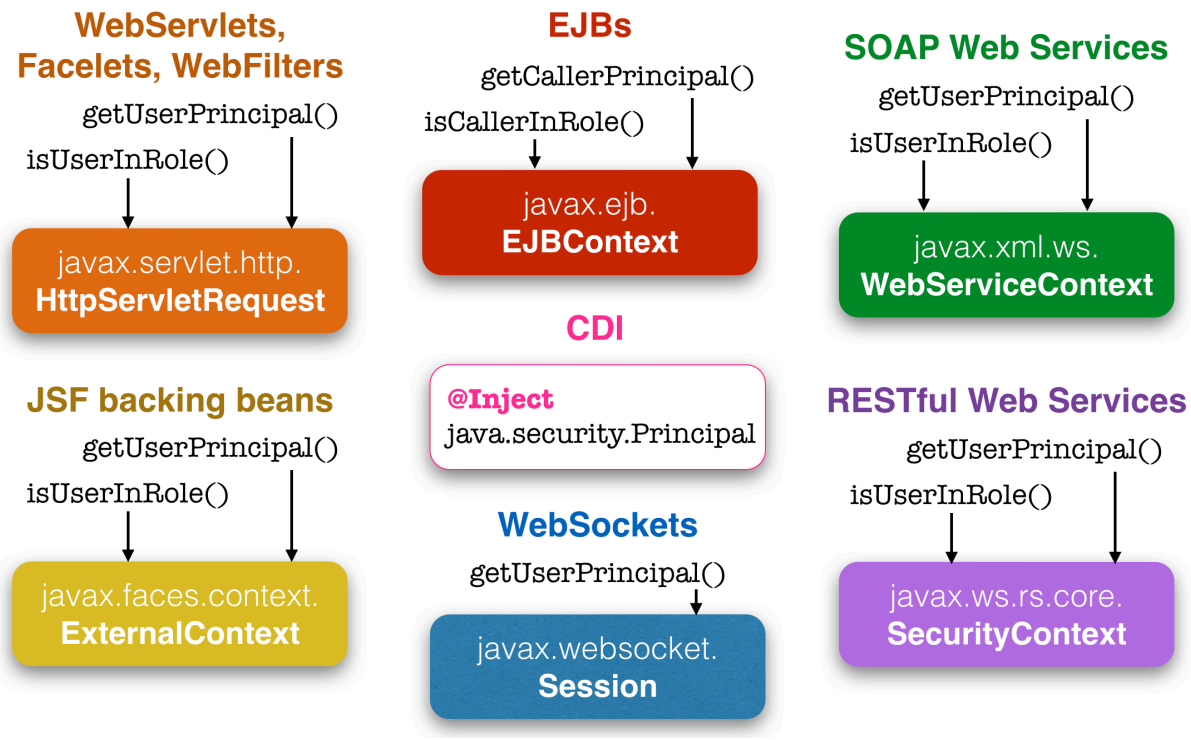
Há uma API programática de segurança contendo apenas os dois métodos básicos em javax.xml.ws.WebServiceContext:

- **getUserPrincipal()**: Retorna java.security.Principal com usuário autenticado.
- **isUserInRole("role")**: Retorna true se usuário autenticado pertence ao role.

O suporte a criptografia na camada de mensagens (ainda) não faz parte do Java EE.

4.6 Resumo: como obter principais e testar roles

Não há uma forma unificada de obter principais e testar roles em Java EE. Cada tecnologia possui métodos e técnicas diferentes. A ilustração abaixo é um resumo de todos os métodos.



5 Referências

5.1 Documentação oficial

[1] Eric Jendrock et al. The Java EE 7 Tutorial, Release 7 (Oracle, September 2014). http://docs.oracle.com/javaee/7/tutorial/ O capítulo 50 contém um tutorial sobre Segurança em Java EE com Glassfish (os exemplos e as configurações funcionam apenas no Glassfish).

[2] Oracle. Java SE Security. http://www.oracle.com/technetwork/java/javase/jaas/index.html Uma visão geral das APIs Java SE de segurança.

[3] Oracle. JAAS Authentication tutorial. http://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/tutorials/GeneralAcnOnly.html

[4] Oracle. JAAS Authorization tutorial. http://docs.oracle.com/javase/8/docs/technotes/guides/security/jaas/tutorials/GeneralAcnAndAzn.html

[5] Oracle. Java Security Overview. http://docs.oracle.com/javase/8/docs/technotes/guides/security/overview/jsoverview.html

[6] Especificações Java EE http://www.oracle.com/technetwork/java/javaee/tech/index.html

[7] Ron Monzillo. Java Authentication SPI for Containers JSR 196, Version 1.1, Oracle, 2013

[8] Ron Monzillo. Java Authorization Contract for Containers JSR 115, Version 1.5, Oracle, 2013

[9] Shing Wai Chan e Rajiv Mordani. Java Servlet Specification, Version 3.1, Chapter 13: Security, Oracle, 2013

[10] Marina Vatkina. JSR 345 Enterprise JavaBeans, Version 3.2. EJBCore Contracts and Requirements. Chapter 12 Security Management. Oracle, 2013.

[11] Danny Coward. Java API for WebSocket. Version 1.0. Chapter 8: Server Security. Oracle, 2013

[12] Jitendra Kotamraju. WebServices for Java EE, Version 1.3. Chapter 9: Security. Sun, 2009.

[13] Jitendra Kotamraju. The Java API for XML-Based Web Services (JAX-WS) 2.2 Rev. a. Section 4.2.2.1 (Standard properties). Oracle, 2011.

[14] Santiago Pericas-Geertsen e Marek Potociar. JAX-RS: Java API for RESTful Web Services. Version 2.0. Section 9.2.5 Security Context. Oracle, 2013

[15] Pete Muir et al. Contexts and Dependency Injection for Java EE, Version 1.1. Section 3.8 (Additional built-in beans). Oracle, 2013.

5.2 Livros

[16] Arun Gupta. Java EE 7 Essentials O'Reilly, 2013. Código-fonte com vários exemplos usando recursos de segurança em Java EE 7: https://github.com/javaee-samples/javaee7-samples

5.3 Artigos e apresentações sobre Java EE 7

[17] Marian Muller. "Please Log In": Authentication and Authorization in Java SE and Java EE. JavaONE 2013. https://www.youtube.com/watch?v=pBDqIavwMYw Tutorial (video) sobre recursos de segurança em Java SE e Java EE, cobrindo JAAS e JASPIC.

[18] Arjan Tijms. Implementing container authentication in Java EE with JASPIC Nov/2012. http://arjan-tijms.omnifaces.org/2014/03/implementing-container-authorization-in.html Tutorial abrangente sobre JASPIC 1.0 (veja também posts do autor sobre JACC).

[19] Ron Monzillo. Using JACC to determine a caller's roles. Ron Monzillo's Weblog. Aug 18 2008. https://blogs.oracle.com/monzillo/entry/using_jacc_to_determine_a Aplicações práticas de alguns recursos da API JACC.

[20] Ron Monzillo. Adding Authentication Mechanisms to the GlassFish Servlet Container. Ron Monzillo's Weblog. Apr 30, 2008. https://blogs.oracle.com/enterprisetechtips/entry/adding_authentication_mechanisms_to_the Um tutorial de JASPIC.