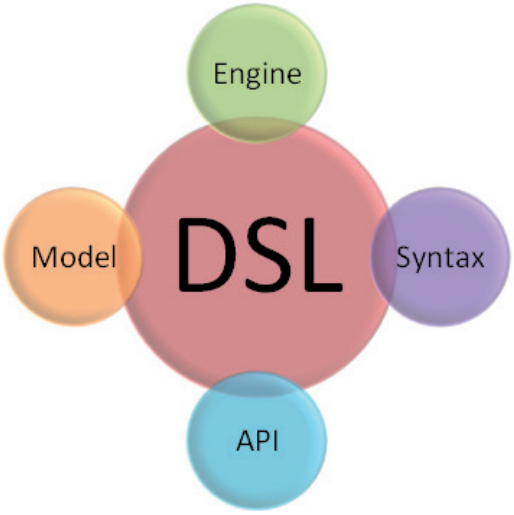
Comunicado Técnico 125
ISSN 1677-8464
Dezembro, 2016
Campinas, SP



Fonte: http://alissonpedrina.blogspot.com.br

Uso de DSL para implementação de regras de autorização em sistemas de informação na arquitetura Java EE

Sérgio Aparecido Braga da Cruz[1]
Daniel Rodrigo de Freitas Apolinário[2]
Isaque Vacari[3]

Introdução

A execução de diferentes operações pelos usuários de grandes sistemas de informação é governada por um conjunto de regras de negócio associadas ao processo implementado pelo sistema. Estas regras refletem políticas ou restrições que determinam um comportamento diferenciado do sistema para cada usuário. A implementação de regras de negócio complexas em sistemas de informação na arquitetura *Java Plataform, Enterprise Edition* (Java EE) pode ser realizada programaticamente por meio de regras de autorização e, geralmente, implicam na construção de longos códigos condicionais que avaliam se uma operação pode ou não ser realizada com base no usuário e no estado atual do sistema (JENDROCK et al., 2014). O objetivo deste trabalho é apresentar uma técnica de implementação destas regras de autorização utilizando *Domain Specific Language* (DSL's) em linguagem Java. As DSL's são utilizadas como linguagens de programação de mais alto nível, expressas utilizando termos mais abstratos e próximos do domínio de um especialista, de modo a tornar mais acessível e facilitado tanto a construção quanto a manutenção de programas por usuários não especialistas em Tecnologia da Informação (TI). A solução apresentada pode ser adaptada para uso em diferentes projetos, podendo ser aplicada para construção de linguagens para outros domínios e aplicações.

Mecanismos de autorização na arquitetura Java EE 7

A arquitetura Java EE 7 modela a estrutura de uma aplicação corporativa típica como uma série de camadas de componentes. A execução desses componentes é controlada por ambientes de execução especiais denominados *containers*, os quais determinam quando e como um componente deve ser instanciado ou destruído e como ele interage com outros componentes e com serviços providos pelo ambiente de execução. O acesso a banco de dados, a mecanismos de comunicação e de segurança são alguns exemplos de serviços que podem ser fornecidos por um *container*. A Figura 1 apresenta como estão organizados os diferentes *containers* de um servidor de aplicação na arquitetura Java EE 7. Os componentes denominados *Enterprise Java Beans* (EJB) são usados para implementar a lógica de negócio de uma aplicação. A Figura 2 exemplifica uma sequência de chamadas de código executada a partir do clique do usuário no botão "Salvar" de um for-

[1] Engenheiro eletricista, doutor em Computação Aplicada, pesquisador da Embrapa Informática Agropecuária, Campinas, SP.
[2] Cientista da computação, especialista em Plataformas de Desenvolvimento Web, analista da Embrapa Informática Agropecuária, Campinas, SP.
[3] Tecnólogo em Processamento de Dados, mestre em Ciência da Computação, analista da Embrapa Informática Agropecuária, Campinas, SP.

Embrapa

mulário de aplicação web fictícia. Nesta sequência uma classe Java denominada Controller implementa métodos que deverão ser invocados em resposta à interação do usuário com a interface web da aplicação. O *Web Container* realiza a ligação entre a interação do usuário com a interface web e os métodos

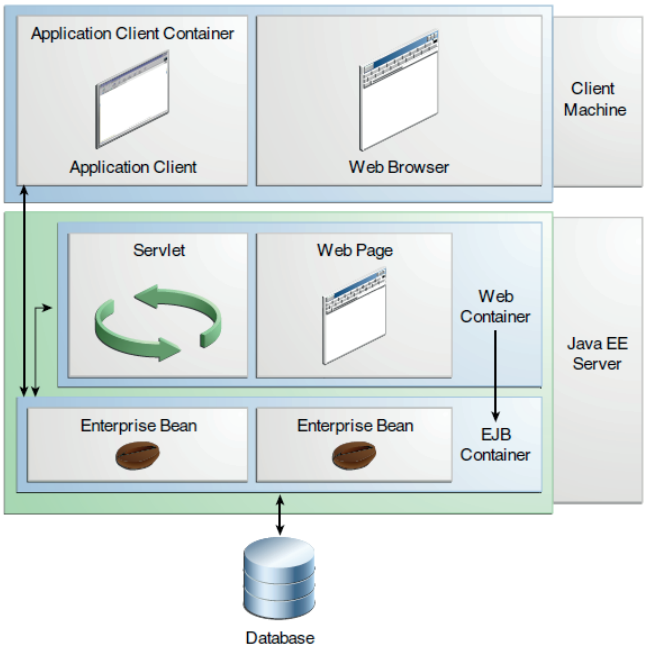


Figura 1. Estrutura de containers na arquitetura Java EE 7.
Fonte: Jendrock et al. (2014).

da classe *Controller*. No exemplo, o clique do usuário no botão "Salvar" está associado à execução do método salvar da classe *Controller*. A implementação do método *salvar()* em *Controller* é baseada na chamada de métodos de classes de camadas de suporte, neste caso EJB's.. No EJB, implementado pelo desenvolvedor do sistema, estão o código que verifica se a operação deve ser realizada (autorização), o código que prepara o dado para persistência e a chamada de métodos de classes de suporte da camada de persistência. A persistência do dado é finalmente realizada pela classe *EntityManger*, que é fornecida pelo EJB *Container* e implementa métodos para acesso ao banco de dados de maneira transparente.

O EJB *Container* fornece suporte a dois tipos de mecanismos de segurança, os quais podem ser utilizados na avaliação da autorização dos métodos em um EJB: segurança declarativa e segurança programática. A segurança declarativa é realizada utilizando o mecanismo de anotação da linguagem Java para descrever em quais condições um método do EJB pode ser executado. Estas descrições possuem um poder limitado para construção de condições uma vez que são baseadas em informações do perfil do usuário logado no sistema.

O EJB *container* avalia as condições descritas nas anotações e desta forma autoriza ou não a execução de um método do EJB. Na segurança programática, o programador fica responsável por construir os testes que irão avaliar se um método do EJB deve ser executado ou não. Para subsidiar esta avaliação, o desenvolvedor deve acessar dados da sessão do usuário e dados sobre o estado do sistema. Este trecho de código normalmente é formado por um conjunto de textos condicionais baseados em expressões booleanas. A Figura 3 apresenta exemplos destes dois mecanismos. Na Figura 3a, a anotação *@DeclareRoles* especifica uma lista de perfis que poderão ser utilizados na avaliação de autorização de algum método da Calculator, neste caso os perfis "*Administrator*", "*Manager*" e "*Employee*". A anotação *@RolesAllowed* especifica que somente usuários com perfil "Administrator" tem autorização para executar o método *setNewRate(int rate)*. A Figura 3b ilustra um exemplo de uso do mecanismo de segurança programática. Neste caso, o desenvolvedor implementa dentro do método *updateEmployeeInfo(EmpInfo info)* da classe EJB *PayrollBean* um teste para verificar se o usuário tem perfil "*payroll*". Caso esta situação não seja verificada, o método termina gerando uma exceção de segurança *SecurityException*.

Quanto mais complexas as regras de negócio associadas à autorização dos métodos do EJB mais complexo se torna o trecho do código necessário para sua avaliação. Apesar da maior facilidade de entendimento para especialistas em TI, este trecho de código pode se tornar ilegível para especialistas do domínio de outras áreas que não têm conhecimento de programação, porém especificam as regras de negócio. Neste trabalho a técnica de construção de uma DSL foi utilizada para aumentar a legibilidade da regra de autorização e desta forma facilitar a sua construção e manutenção.

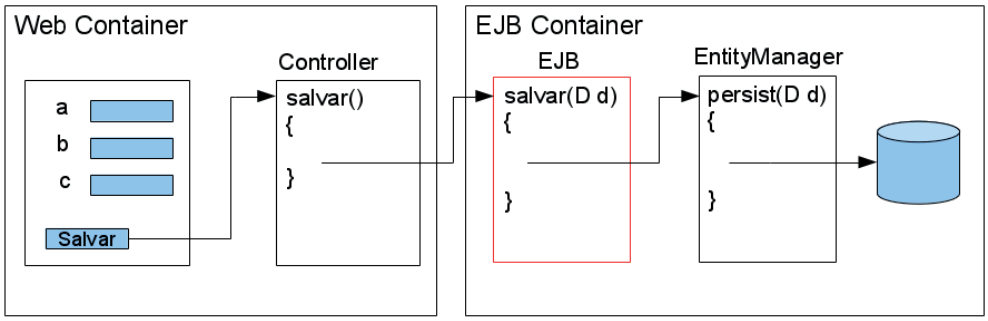


Figura 2. Sequência de chamadas de métodos em aplicação JEE.

```
@DeclareRoles({"Administrator", "Manager", "Employee"})
public class Calculator {

    @RolesAllowed("Administrator")
    public void setNewRate(int rate) {
        ...
    }
}
```

(a)

```
@Stateless public class PayrollBean implements Payroll {
     @Resource SessionContext ctx;

     public void updateEmployeeInfo(EmplInfo info) {

         oldInfo = ... read from database;

         // The salary field can be changed only by callers
         // who have the security role "payroll"
         if (info.salary != oldInfo.salary &&
             !ctx.isCallerInRole("payroll")) {
                 throw new SecurityException(...);
         }
         ...
     }
     ...
 }
```

(b)

Figura 3. Exemplos de implementação de autorização baseados em: (a) segurança declarativa; (b) segurança programática.
Fonte: Jendrock et al. (2014).

DSL na linguagem Java

Uma DSL é uma linguagem com alto nível de abstração que é definida tendo como requisito principal a facilidade de uso por uma comunidade de especialistas de um domínio. Estas linguagens são utilizadas na construção de aplicações de propósito específico em domínios restritos (FOWLER, 2008)(KABANOV et al., 2011). Neste trabalho foi implementada uma DSL utilizando recursos da linguagem Java, ou seja, o trecho de código em DSL fica embutido no código Java e, apesar de aparentar uma sintaxe diferente, continua sendo um código daquela linguagem, que depende de seu ambiente de execução para ser executado. DSL's que possuem esta dependência da linguagem hospedeira são denominados DSL´s internas.

A melhor legibilidade da DSL implementada foi obtida utilizando dois recursos principais. Cada teste necessário para verificação de autorização foi implementado em classes Java distintas por meio de métodos com nomes sugestivos ao que está sendo verificado. Isto facilita o entendimento imediato de um especialista do domínio sobre qual é a condição que está sendo testada. O mecanismo de importação estática de métodos da linguagem Java foi utilizado para permitir a construção de expressões de autorização de forma simplificada em um estilo semelhante à linguagem funcional.

Estudo de Caso

A Figura 4a ilustra uma expressão DSL utilizada para verificar se uma operação de atualização de uma entrada no cadastro de tecnologias poderá ser realizada pelo usuário logado no sistema. A avaliação desta regra retorna um valor true ou false, indicando se a operação de atualização pode ser realizada ou não. A Figura 4b ilustra como seria a construção da mesma expressão se não fosse utilizado o mecanismo de importação estática de métodos do Java.

```
import static br.embrapa.geocloud.dsl.AuthExpressionBuilder.and;
import static br.embrapa.geocloud.dsl.AuthExpressionBuilder.inAdministrativeRole;
import static br.embrapa.geocloud.dsl.AuthExpressionBuilder.isTechnologyOwner;
import static br.embrapa.geocloud..dsl.AuthExpressionBuilder.or;

...

AuthExpression defaultExpr =
                or(
                        inAdministrativeRole("ADMIN"),
                        and(
                                inAdministrativeRole("TECHADMIN"),
                                isTechnologyOwner()
                        )
                );

...

if(defaultExpr.eval())
{
    ....
}
else
{
  ...
}
```

(a)

```
AuthExpression technologyOwner = AuthExpressionBuilder.isTechnologyOwner();
AuthExpression inAdministrativeRoleTechadmin = AuthExpressionBuilder.inAdministrativeRole("TECHADMIN");
AuthExpression and = AuthExpressionBuilder.and(inAdministrativeRoleTechadmin, technologyOwner);
AuthExpression inAdministrativeRoleAdmin = AuthExpressionBuilder.inPerfilAdministrativo("ADMIN");
AuthExpression defaultExpr = AuthExpressionBuilder.or(inAdministrativeRoleAdmin, and);
```

(b)

Figura 4. Regra de autorização para operação de *update* em cadastro de tecnologias (a) em DSL e (4) na sua forma Java explícita.

A classe *AuthExpressionBuilder* funciona como uma *Factory* (GAMMA et al., 1995), possuindo uma série de métodos estáticos, os quais constroem uma instância de objeto para cada tipo de teste a ser realizado. Cada um destes objetos é instância de uma classe que encapsula todas as consultas necessárias ao estado atual da aplicação, o que torna o código das regras de autorização mais claro. Um diagrama simplificado das classes que implementam a DSL de autorização é apresentado na Figura 5. No diagrama podemos visualizar os métodos estáticos de *AuthExpressionBuilder* com nomes correspondendo aos testes a serem realizados. Cada método retorna um objeto de uma das classes *InAdministrativeRole, IsTechnolyOwner, Or, And* e *Not. As classes InAdministrativeRole* e *IsTechnolyOwner* são subclasses da classe *Authorization* o que permite o acesso a informações sobre a sessão do usuário (*SessionData*), utilizadas na avaliação dos testes. Todos os objetos criados por *AuthExpressionBuilder* implementam a interface *AuthExpression*, ou seja, pos-

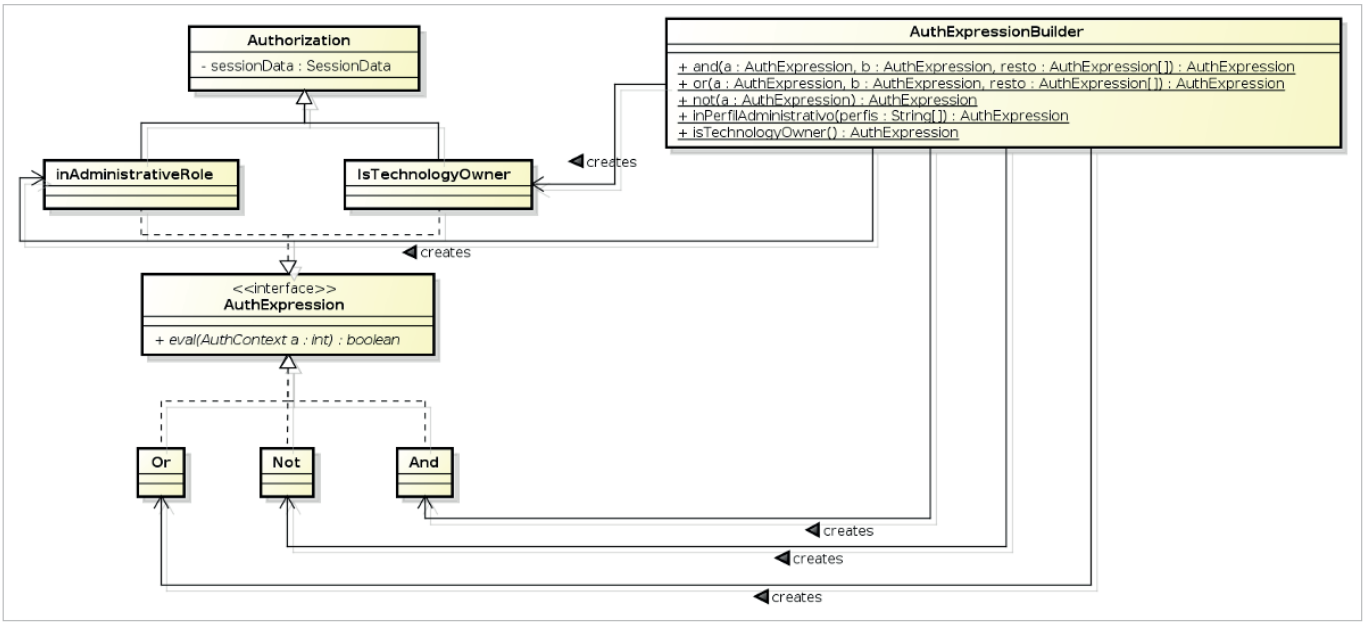


Figura 5. Implementação de regra de autorização usando segurança programática padrão.

suem o método *eval()* que retorna um valor booleano de acordo com a avaliação do teste de autorização.

A Figura 4a, apresentada anteriormente, ilustra como deve ser o uso da DSL. Em um primeiro momento, a expressão deve ser construída utilizando os métodos de *AuthExpressionBuilder*. Como resultado, um objeto em memória é criado com a estrutura dos testes a serem realizados. Em um segundo momento, o método *eval()* da expressão é chamado, o que dispara a execução de todos dos métodos *eval()* da estrutura. Como resultado um valor *true* ou *false* é retornado de acordo com o estado atual da sessão do usuário disponível em *sessionData*.

Na Figura 6, está um trecho de código ilustrando como seria a implementação desta mesma regra sem a utilização da DSL. Além da maior dificuldade em identificar a regra de autorização necessária para a permitir ou não a atualização de uma entrada no cadastro de tecnologias, a manutenção e a atualização desta regra está sujeito a uma série de erros, uma vez que haverá uma maior dificuldade por não especialistas em TI em entender o código e a regra implementada, bem como existirá um maior esforço do especialista em TI em codificá-la.

A *DSL* não reduz a quantidade de operações necessárias para avaliação da regra de autorização, porém ela permite organizar e modularizar este código de modo que ele possa ser reutilizado de forma transparente. Os objetos de teste criados por *AuthExpressionBuilder* precisam, em algum momento, realizar consultas ao estado do sistema para avaliar se uma condição é verdadeira ou falsa, porém, o código necessário para esta avaliação fica “escondido” do nível em que as regras de autorização são declaradas.

```
@Resource private SessionContext ctx;

public void updateTechnology(Technology technology)
{
        String login = null;
        try {
                Principal principal = ctx.getCallerPrincipal();

                if (principal != null && !principal.getName().equalsIgnoreCase("anonymous")) {
                        login = principal.getName();
                }
        }
        catch (IllegalStateException e) {
                e.printStackTrace();
        }

        boolean auth = login!=null;

        if(!auth)
        {
                String msg = "Usuário não esta logado";
                throw new SecurityException(msg);
        }

        //Consulto tabelas do sistema para verificar se usuario logado pertence a
        //equipe administrativa ou de administracao de tecnologias "ADMIN","THECHADMIN"
        auth = ... ;
        if (!auth) {
                String msg = "Usuário %s não pertence a equipe administrativa";
                throw new SecurityException(String.format(msg, login));
        }

        if(technology!=null)
        {
                auth = (technology.getRecordOwner().getLogin().equals(login));
                if(!auth)
                {
                        String msg = "Usuário %s não é responsável pela Tecnologia. ";
                        throw new SecurityException(String.format(msg, login));
                }
        }
        else
        {
                String msg = "Tecnologia não identificada";
                throw new SecurityException(String.format(msg));
        }

        //Atualizo tecnologia
        ...
}
```

Figura 6. Implementação de regra de autorização usando segurança programática padrão.

Conclusão

Existem ferramentas que permitem a incorporação de regras de negócios em um sistema de informação, porém, além de representar uma nova dependência para construção do sistema, podem gerar impacto devido à necessidade de capacitação e adaptação do sistema aos requisitos de instalação da nova ferramenta. A DSL implementada neste trabalho utiliza recursos da linguagem Java e pode mais facilmente ser customizada para outros tipos de aplicação. Como o seu uso está embutido no código fonte, qualquer alteração nas regras de autorização pode ser registrada no controlador de versão utilizado na gestão do código do sistema. A gestão das regras de autorização possui um impacto importante na segurança de um sistema e quanto maior a transparência no acesso e na forma em que estas regras são avaliadas, melhor é a possibilidade de realização de auditorias e testes para sua validação. O código-fonte pode ser adaptado para refletir com maior detalhe qualquer condição que se deseje testar.

Referências

FOWLER, M. **DslQandA**. 2008. Disponível em: <http://martinfowler.com/bliki/DslQandA.html>. Acesso em: 28 out. 2016.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns**: elements of reusable object-oriented software. Reading: Addison-Wesley, 1995. 395 p. (Addison-Wesley professional computing series).

JENDROCK, E.; CERVERA-NAVARRO, R.; EVANS, I.; HAASE, K.; MARKITO, W. **Java platform**: the Java EE tutorial 7. California: Oracle Corporation, 2014. Disponível em: <https://docs.oracle.com/javaee/7/tutorial/>. Acesso em: 15 dez. 2016.

KABANOV, J.; HUNGER, M.; RAUDJÄRV, R. On designing safe and flexible embedded DSLs with Java 5. **Science of Computer Programming**, v. 76, n. 11, p. 970-991, Nov. 2011. DOI: doi:10.1016/j.scico.2010.04.005. Acesso em: 15 dez. 2016.

Comunicado Técnico, 125

Embrapa Informática Agropecuária
Endereço: Caixa Postal 6041 - Barão Geraldo
13083-886 - Campinas, SP
Fone: (19) 3211-5700
www.embrapa.br/informatica-agropecuaria
www.embrapa.br/fale-conosco/sac/

MINISTÉRIO DA AGRICULTURA, PECUÁRIA E ABASTECIMENTO

1ª edição publicação digital - 2016

Expediente

Supervisão editorial: *Stanley Robson de Medeiros Oliveira, Suzilei Carneiro*

Normalização bibliográfica: *Maria Goretti Gurgel Praxedes e Victor Paulo Marques Simão*

Revisão de texto: *Adriana Farah Gonzalez*

Editoração eletrônica: *Suzilei Carneiro*

CGPE 13503