GERAÇÃO DE CÓDIGO BASEADA EM METAMODELOS PARA SEGURANÇA EM SISTEMAS DE INFORMAÇÃO WEB[1]

José Antonio S. Neto, Rodrigo Vilar

Departamento Ciências Exatas (DCX) – Universidade Federal da Paraíba(UFPB)
Rua da Mangueira, s/n, Companhia de Tecidos Rio Tinto
CEP 58297-000 - Rio Tinto - PB - Brasil

{jose.silvaneto,rodrigovilar}@dce.ufpb.br

Abstract. *Several frameworks use the Scaffolding technique, which consists of generating the system functional code quickly, to increase developers productivity. However most of Scaffolding techniques do not give the suitable support to the implementation of security requirements. This work searches to analyze the viability of a Scaffold capable of generating a security code automatically without the need of customization posteriorly. For this, we will develop a template for security Scaffolding, use a generator with fine granularity templates and finally, perform an experiment to compare the readability of the security code with the developed template.*

Resumo. Diversos frameworks utilizam a técnica *Scaffolding*, que consiste em gerar rapidamente o código funcional do sistema, para aumentar a produtividade dos desenvolvedores. No entanto, a maioria das técnicas de *Scaffolding* não dá o suporte adequado para a implementação dos requisitos de segurança. Este trabalho busca analisar a viabilidade de um *Scaffold* capaz de gerar um código de segurança automaticamente sem a necessidade de customização posterior. Para isso, iremos desenvolver um modelo para *Scaffolding* de segurança, utilizar um gerador com templates de granularidade fina e por fim, realizar um experimento para comparar a legibilidade do código de segurança com o template desenvolvido.

1. Introdução

O mercado de software está em grande evolução nos últimos anos, e a forma como se constrói sistemas de informação web também tem evoluído, de modo que os paradigmas de programação também mudam rapidamente paralelo ao surgimento de novas tecnologias. Além disso, existe uma demanda crescente por software no mercado, exigindo dos programadores a entrega cada vez mais rápida de aplicações (Shinde, 2016). Grande parte dos sistemas de informação possui operações de inserção, atualização, remoção e leitura de registros em banco de dados, que são conhecidas como CRUD (do inglês *Create*, *Read*, *Update* e *Delete*). A implementação de cada uma dessas operações requer uma demanda de tempo e recurso em um projeto. Uma das abordagens utilizadas por diversos *frameworks* web para aumentar a produtividade nas implementações é a técnica de *Scaffolding*, que consiste na geração do código fonte inicial de aplicações a partir de meta modelos. Dessa forma, para obter um protótipo funcional do software, o desenvolvedor gasta tempo apenas para construir

[1] Trabalho de Conclusão de Curso (TCC) na modalidade Artigo apresentado como parte dos pré-requisitos para a obtenção do título de Bacharel em Sistemas de Informação pelo curso de Bacharelado em Sistemas de Informação do Centro de Ciências Aplicadas e Educação (CCAE), Campus IV da Universidade Federal da Paraíba, sob a orientação do professor Rodrigo de Almeida Vilar de Miranda.

os meta modelos do projeto, pois todo restante do código referente a CRUDs, como por exemplo, controllers e views do padrão arquitetural MVC, são criados automaticamente.

Quando um desenvolvedor vai implementar algum software do zero, precisa se preocupar com diversos requisitos funcionais e não funcionais. Na maioria dos casos, vários desses requisitos estão relacionados à segurança do sistema, como por exemplo, cadastro de usuários, permissões e papéis; renovação e revogação de chaves de segurança; compartilhamento de recursos de origem cruzada; criptografia; entre outros.

O mecanismo de *Scaffolding* de alguns *frameworks* como, *Ruby on Rails*, *Django*, *JHipster*[1] não geram automaticamente alguns desses aspectos de segurança. Assim sendo, após o processo de geração de código, o desenvolvedor precisa customizar o código fonte gerado para implementar os requisitos de segurança desejados. Apesar dessa limitação no *Scaffolding*, alguns *frameworks* disponibilizam bibliotecas que auxiliam o desenvolvedor a criar componentes de segurança após o código ser gerado, como é o caso do *Cancancan*[2], que é uma biblioteca em Ruby com diversas APIs de segurança. Mas, como é possível observar na avaliação da Seção 5, mesmo com esse auxílio, ainda é custoso e complexo criar tais componentes de segurança, pois o desenvolvedor precisa conhecer e estudar bem a melhor maneira de implementar os requisitos necessários, além de ter que obter conhecimento de boa parte das APIs que a biblioteca do *framework* disponibiliza.

Vilar et, al (2014) propõem que, ao invés de se despender tempo na criação e evolução de código fonte, seja investido tempo na criação de *templates* de geração de código que posteriormente possam servir para transformar meta modelos em código fonte. Dessa forma, existe o esforço inicial apenas para criação dos *templates*, que em seguida podem ser utilizados para gerar código para várias aplicações, sem a necessidade de se realizar customização após a geração.

Por esse motivo, o objetivo deste presente trabalho é desenvolver e avaliar metadados e *templates* de geração de código, com o qual seja possível o desenvolvedor especificar os requisitos de segurança do sistema e em seguida o código seja gerado, sem a necessidade de realizar alguma customização posteriormente. A demanda por geração automática de código para segurança também foi identificada por Magno (2015) em experimento realizado em seu trabalho, onde os participantes responderam um questionário e sugeriram que além do seu *scaffolding* gerar código na arquitetura *MVC (model, view, controller)*, fosse possível gerar também módulo de *login* e cadastro de usuários.

Os componentes de segurança foram desenvolvidos utilizando as linguagens de programação *Java* e *Typescript*[3] com o auxílio dos *frameworks Spring boot* e *Angular*[4] respectivamente. Para criar os templates de código e realizar o processo de geração foi utilizado o *Potter Generator*[5], que possui um diferencial, onde os templates possuem granularidade fina e permitem a implementação de diversos requisitos segurança mais facilmente, pois os templates pequenos são mais independentes entre si e podem ser habilitados ou não de forma declarativa no meta modelo.

---

[1] https://www.jhipster.tech/
[2] https://github.com/CanCanCommunity/cancancan
[3] https://www.typescriptlang.org/
[4] https://angular.io/
[5] https://github.com/potterjs/potter-project

Este trabalho está organizado da seguinte forma. A Seção 2 apresenta a fundamentação teórica sobre segurança. A Seção 3 apresenta os passos metodológicos do trabalho. A Seção 4 apresenta a solução proposta. A Seção 5 o experimento para validar a solução e por fim, a Seção 6 apresenta a conclusão e trabalhos futuros.

2. Fundamentação teórica

Nesta Seção serão explanados os modelos e características de segurança que serão considerados para a implementação do *Scaffolding* proposto, tais como, cadastro de usuários, permissões e papéis; renovação e revogação de chaves de segurança; compartilhamento de recursos de origem cruzada; além dos modelos *access control list* e *role-based access control*. Por último, será descrito o conceito de templates de geração de código e como deve ser todo o processo desde sua criação até o código gerado.

2.1 Access control list (ACL)

Este modelo de segurança é um dos mais triviais que existem e garante autorização em todo o sistema. De forma geral, o ACL consistem em indicar a quais recursos de um sistema um determinado indivíduo pode ter acesso (Fernandez-Buglioni, 2013). Essas permissões a recursos normalmente são associadas a usuários de um sistema, onde o indivíduo pode possuir *N* permissões. Dependendo do caso de uso planejado, essas permissões podem ser por exemplo, *add a customer*, *update a customer*, *delete a customer*, *get a customer*. Este modelo define que as permissões possuem granularidade fina, e isso quer dizer que cada indivíduo tem permissões para fazer apenas operações individuais dentro do sistema.

Um problema que este modelo possui é que, toda vez que um indivíduo é cadastrado em um sistema, é necessário associar todas as permissões que ele deve possuir. Em sistemas de informação de grande escala, um indivíduo pode possuir um número grande de permissões dependendo da sua função dentro da organização. Um modelo que não possui este problema é o Role-based access control, que está descrito mais adiante no texto.

2.2 Role-based access control (RBAC)

Um modelo de segurança que é bastante utilizado por diversas organizações é o RBAC (*role-based access control*). Esse modelo consiste em um controle de acesso onde é possível restringir um recurso do sistema baseado nas permissões que o papel do indivíduo possui. Esse papel pode ser por exemplo, "Administrador", "Professor" ou outra função existente na organização (Sandhu, 1996). Cada papel possui um conjunto de permissões com granularidade fina igual a do modelo ACL descrito anteriormente. A partir desses aspectos, cada indivíduo da organização só consegue acessar determinados recursos do sistema se seu papel possuir a permissão para o recurso. Uma diferença entre o modelo ACL e RBAC é que, no modelo RBAC as permissão não são associadas ao indivíduo, mas sim ao papel que o indivíduo possui.

Uma vantagem que o RBAC possui é que, ao cadastrar um indivíduo no sistema, é necessário apenas realizar a associação seu papel. Dessa forma, todo o conjunto de permissões associadas a este papel é automaticamente associada ao indivíduo. Outra vantagem é que caso uma nova permissão seja adicionada em um papel da organização, todos os indivíduos já irão possuí-la automaticamente. Já no modelo ACL, esta associação necessitaria ser feita em todos os usuários.

2.3 Tokens de segurança

Na segurança de sistemas de informação, normalmente o servidor e o cliente têm que manipular constantemente *Tokens de Acesso* e *Refresh Tokens*. Token de Acesso é um token que contém todas as credenciais de um indivíduo no sistema (OAuth, 2018). Estas credenciais podem ser por exemplo, o conjunto de permissões que o papel do indivíduo possui no modelo RBAC ou ACL. Já o Refresh Token é um token especial que serve apenas para obter um novo token de acesso do servidor com as credenciais atualizadas do indivíduo (OAuth, 2018). Na maioria dos casos, o *Refresh Token* é utilizado quando a validade do *Token de Acesso* acaba ou quando novas permissões são adicionadas ao usuário que é o portador do token.

Utilizando o padrão JWT (*json web token*) é possível o servidor encapsular as permissões e papéis do usuário em um token, que posteriormente é enviado ao usuário após a operação de login no sistema. O JWT é um padrão definido na RFC 7519 em que, a partir de um JSON (*javascript object notation*) é possível gerar um código *Hash* que é assinado pelo servidor. Também é possível decifrar o código *Hash* utilizando a assinatura do servidor. Nesse caso, após o processo de decifrar o *Hash* é possível obter o valor original do JSON e recuperar as permissões e papéis que foram inseridos inicialmente no token. Na Figura 1, pode-se ver um exemplo de token cifrado e decifrado. Esse processo é importante, pois toda requisição feita ao servidor existe a necessidade de decifrar o token para verificar se realmente este token foi emitido pelo próprio servidor e para verificar se o portador do token possui permissão para acessar o sistema.

O token de acesso possui operações que são importantes para a segurança do sistema e do portador do token. Essas operações são de Revogação e Invalidação do Token. A revogação consiste em forçar o cliente a atualizar seu token de acesso caso alguma permissão seja removida ou atribuída enquanto o indivíduo estiver autenticado no sistema. Caso não houvesse a operação de revogação, mesmo que suas permissões fossem alteradas, o token de acesso do indivíduo iria permanecer o mesmo e a atualização em suas permissões só iriam ser refletidas no sistema caso fosse realizado logout ou seu token de acesso expirasse.

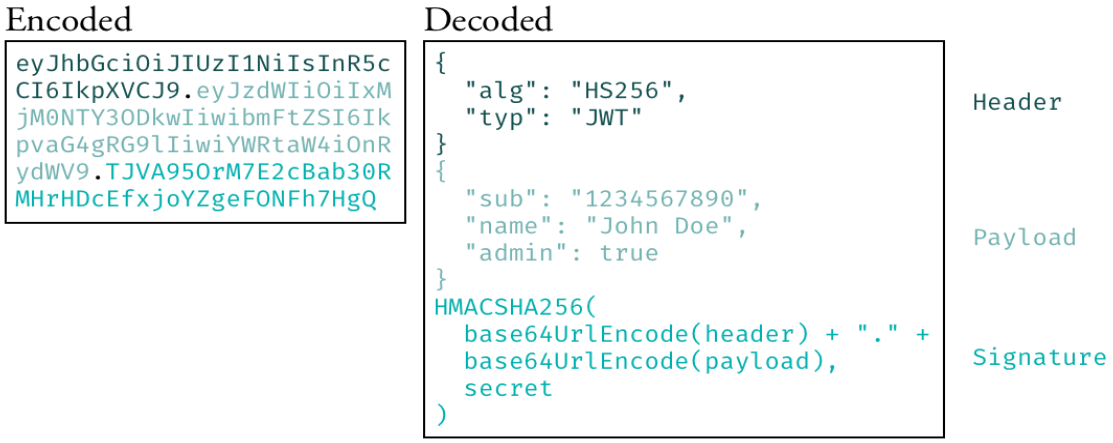

Figura 1. JWT cifrado e descifrado.

Fonte: Página do blog Artsy[6]

[6] http://artsy.github.io/blog/2016/10/26/jwt-artsy-journey/

A invalidação de token é um passo a mais na segurança do sistema. Caso algum indivíduo tenha sua conta comprometida devido ao vazamento das suas credenciais ou tenha esquecido de realizar logout em algum dispositivo, a invalidação de token serve para forçar o cliente autenticado a inserir novamente as credenciais login. Essas operações de Revogação e Invalidação de token, normalmente devem ser realizadas pelo administrador do sistema.

2.4 Outros aspectos de segurança

Os modelos de segurança Role-based control e Access Control List consideram apenas as entidades necessárias para o controle de acesso a um recurso, como por exemplo usuários, papéis e permissões. Mas quando se fala em segurança em sistemas de informação web é necessário considerar outros componentes importantes como *cors* (Compartilhamento de Recursos de Origem Cruzada) e *https*.

cors é uma regra implementada pelos navegadores que garante que um recurso do sistema não vai ser acessado por um domínio diferente no browser que não seja o próprio domínio do servidor (Huang, 2010). Por exemplo, se o domínio do servidor é www.exemplo1.com.br e o domínio que realiza a requisição ao servidor pelo browser é www.exemplo2.com.br, a requisição será bloqueada automaticamente. Essa técnica impede ataques do tipo XSS (Cross-Site Scripting), que exploram vulnerabilidades dos browsers e injetam script na página de um usuário com o intuito de sequestrar sua sessão.

O protocolo *https* garante que a comunicação de ponta a ponta entre o cliente e servidor seja criptografada. Dessa forma, caso algum indivíduo mal intencionado capture a informação trafegada na rede com um farejador de pacotes de rede, a informação estará cifrada e não poderá ser aberta. Em aplicações que seguem o padrão REST, existe grande necessidade em cifrar essa comunicação devido o trânsito constante de tokens de acesso entre o cliente e servidor.

2.5 Template de geração de código

Geradores que são construídos para se basear em templates de código, basicamente conseguem transformar dado em código-fonte. O template nada mais é que um arquivo de texto que contém a especificação do código fonte que se deseja gerar. Todos os códigos genéricos do template, tais como, nome da classe, pacote, métodos ou construtor, devem ser substituídos por parâmetros, que em seguida são interpretados e processados pelo gerador Magno (2015). Os parâmetros que estão no template após serem interpretados, são processados e logo em seguida são substituídos pelo seu valor correspondente no meta modelo. Além de atributos, o template também pode possuir controle de fluxo e de laços de repetições, possibilitando maior flexibilidade em seu desenvolvimento Magno et, al (2015). A Figura 2 demonstra a construção e funcionamento de um template simples.

Antes de algum código-fonte se transformar em template, é necessário que tal código tenha sido usado ou testado antes. Dessa forma, após o processo de geração, o código gerado pode ser considerado confiável e estável.

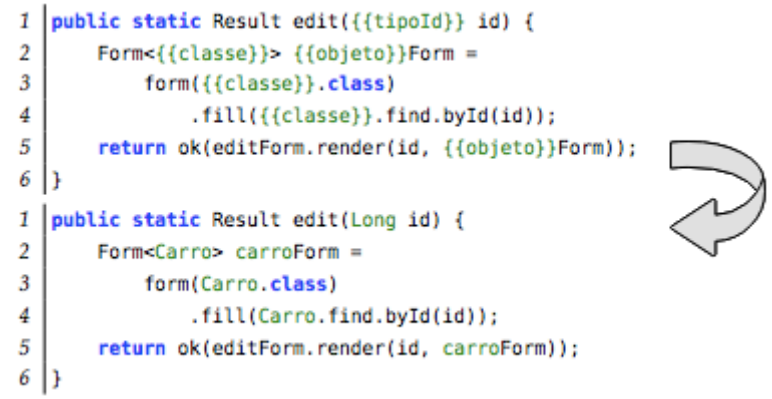

Figura 2. Exemplo de como o template é transformado em código fonte

Fonte: adaptada Magno (2015).

O processo de substituição e sintaxe dos parâmetros no template pode variar de gerador para gerador. No exemplo da Figura 2, o parâmetro *{{classe}}* será substituído pelo valor do atributo *classe* no meta modelo, sendo realizada a mesma operação para todos os parâmetros restantes até um novo código fonte ser criado. Dessa forma, este template pode ser reutilizado diversas vezes, alterando apenas o valor do parâmetro.

3 Metodologia

Nesta Seção será apresentado os passos metodológicos do trabalho, bem como definição de um modelo genérico de segurança para sistemas de informação web, desenvolvimento do código fonte do modelo descrito e por fim extração de templates do código para em seguida ser possível gerá-lo a partir dos metamodelos.

Para validar a solução proposta, elaboramos um experimento no qual selecionamos desenvolvedores com experiências distintas e medimos o tempo de sua percepção para algumas características de segurança presentes no metadado. Posteriormente foi realizada a mesma atividade, mudando apenas o metadado por um código estático. O design do experimento e os resultados estão descritos adiante na Seção 5.

3.1 Modelo de segurança

Antes de iniciar o desenvolvimento da solução proposta, se fez necessário desenvolver um modelo genérico de segurança para sistemas web, onde fosse possível visualizar de maneira trivial quais as características principais que um sistema precisa possuir para prover segurança e qual a dependência entre elas.

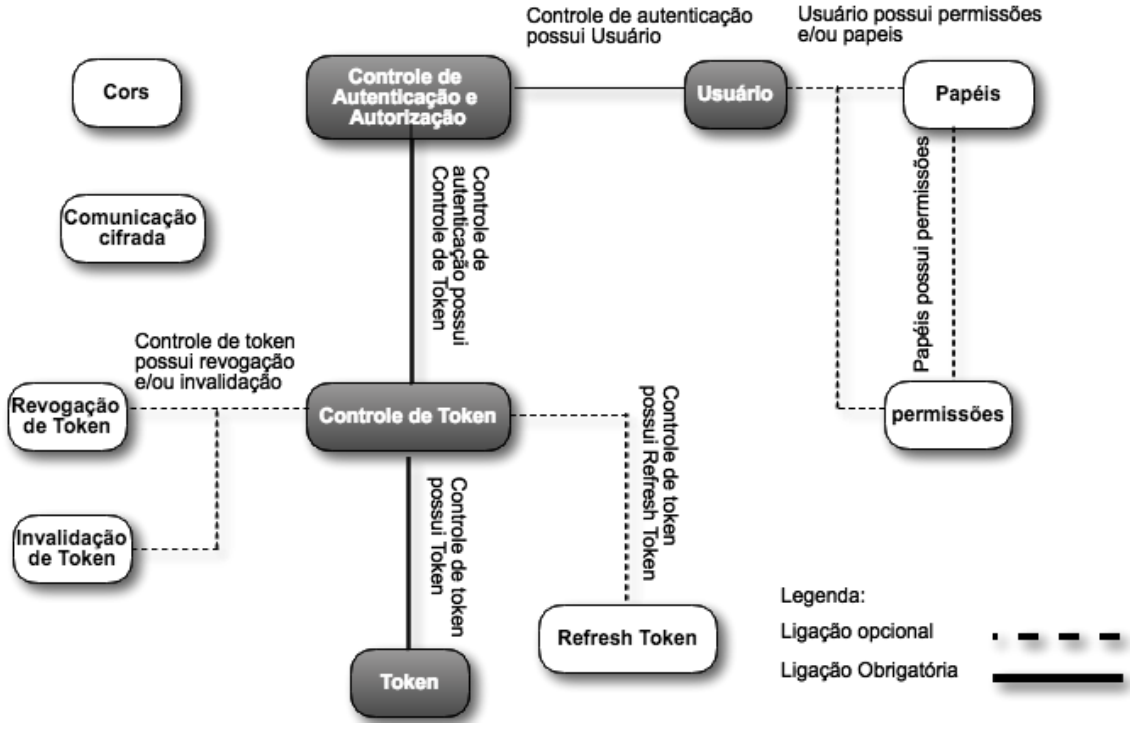

Figura 3. Modelo de segurança genérico

A partir das características mencionadas no referencial teórico foi possível chegar no modelo exposto na Figura 3. Todos os itens em cor escura são itens obrigatórios, e nesse caso, a única característica obrigatória é a autenticação, pois sem ela não faz sentido implementar as características restantes. Já os itens em cor mais clara representam características que são opcionais, como controle de permissões, papéis, etc. Com essas informações estruturada desta forma, se torna mais fácil definir quais atributos de configuração o meta modelo deve possuir para gerar o código desejado.

3.2 Implementação dos requisitos de segurança por frameworks

Após o desenvolvimento do modelo, foi necessário verificar se *frameworks* web que possuem *scaffolding* implementavam as características citadas. Três *frameworks* foram comparados: *Django*, *Rails* e *JHipster*, que conseguem gerar código fonte em *Python, Ruby* e *Java*, respectivamente.

Todos os frameworks descritos geram código estático, ou seja, após o processo de geração de código o desenvolvedor consegue alterar características ou comportamentos direto no código fonte. Existem *scaffoldings* que conseguem gerar código em tempo de execução de forma dinâmica, como é o caso do *framework Grails* Magno (2015). Nesse cenário, o framework faz uso de comandos *static* nas próprias classes dos modelos para criar métodos, construtor e atributos. Como o gerador proposto irá gerar código estático, resolvemos não avaliar *scaffoldings* dinâmicos.

Tabela 1 - Implementação dos requisitos de segurança pelos frameworks

Requisito Segurança	Django	Rails	JHipster
ACL	Sim (obrigatório)	Não	Não
RBAC	Sim (obrigatório)	Não	Sim (obrigatório)
Refresh Token	Não	Não	Não
Revogação Token	Sim (obrigatório)	Não	Não
Invalidação Token	Não	Não	Não
https	Não	Não	Não
cors	Não	Não	Não

Como resultado comum entre os *frameworks Django* e *JHipster*, é possível perceber que nenhum dá flexibilidade para o desenvolvedor escolher qual característica deve ser implementada pelo seu *Scaffolding*. Dessa forma, o desenvolvedor não possui autonomia para ativar ou desativar tal característica do modelo. Outro ponto perceptível na análise é que todos os *frameworks* não cobrem nem 50% dos requisitos de segurança elencados neste trabalho.

4. Solução

Magno (2015) define o ciclo de vida de um gerador baseado em template em 4 fases:

1. Criar ou utilizar o código fonte de projetos bem-sucedidos que implementam uma determinada funcionalidade desejada;
2. Parametrizar todo o texto genérico no código, como nome de classes, métodos e etc;
3. Implementar o gerador para atribuir os valores do meta modelos nos templates parametrizados da fase 2;
4. Gerar o código fonte.

Seguindo esse ciclo de vida, foram implementadas todas as características do modelo da Figura 3. A implementação foi feita utilizando os *frameworks* web *Spring Boot* e *Angular.* Após o término da implementação, foi necessário parametrizar todo o código fonte desenvolvido para transformá-los em template de geração. Na Figura 4 é possível visualizar a forma como é parametrizado os templates que serão interpretados pelo gerador.

```
{
    path: '<%= entity.name.toLowerCase() %>s',
        component: List<%= entity.name %>sComponent,
            canActivate: [Loggedinguard],
                data: { permissions: ['list:<%= entity.name.toLowerCase() %>'],
                roles: [<%- $.invoke('make-access-control-module', entity)%> ''] }
},
{
    path: '<%= entity.name.toLowerCase() %>s/:id',
        component: Edit<%= entity.name %>Component,
            canActivate: [Loggedinguard],
            data: { permissions: ['edit:<%= entity.name.toLowerCase() %>'],
            roles: [<%- $.invoke('make-access-control-module', entity)%> ''] }
},
```

Figura 4. Exemplo de template de URLs do *Angular*

Ainda segundo o ciclo de vida descrito por Magno (2015), a fase 3 necessita de um gerador para interpretar e processar os templates criados anteriormente. Neste trabalho, para que fosse possível interpretar e processar os templates utilizamos o *Potter*, que é um gerador de código baseado em templates de granularidade fina. Além do *Potter* existem também outros geradores já implementados, como o *Yeoman.* Ambos conseguem interpretar e gerar código a partir de templates, mas resolvemos utilizar o *Potter* por possuirmos conhecimento prévio e familiaridade com suas diversas APIs. Contudo, apesar do *Potter* possuir grande riqueza de APIs para realizar operações com os templates, foi necessário realizar pequenas alterações em seu código fonte para que fosse possível a interpretação correta das configurações de segurança que eram necessárias serem inseridas no metamodelo. Essa alteração se resumiu a adicionar o atributo "access-control" no metadado de "entity" para configurar o controle de acesso das entidades que fossem ser geradas a partir do metadado. Com isso, o *Potter* consegue interpretar tal atributo e consequentemente gera as configurações necessárias de controle de acesso.

Após seguir o passo a passo do ciclo de vida descrito anteriormente, foi necessário criar o metadado que será lido pelo *Potter* para que seja possível realizar todo o processo de geração. O metadado foi definido como um arquivo JSON (*javascript object notation*) e na Figura 4, podemos observar como é o formato do metadado para uma implementação das características da Figura 3.

```
{
    "project": {
        "name": "tcc",
        "group": "br.ufpb.dcx",
        "version": "0.0.1"
    },
    "entities": [
        {
            "name": "User",
            "to-string": "email", "labels": { "singular": "User", "plural": "Users" },
            "properties": [
              { "name": "roles", "label": "Roles", "type": "rel", "target": "Role", "to-string": "label"},
              { "name": "permissions", "label": "Permissions", "type": "rel", "target": "Permission", "to-string": "label"}
            ]
        },
        {
            "name": "Role",
            "to-string": "label", "labels": { "singular": "Role", "plural": "Roles" },
            "access-control": ["Admin", "User"],
            "tags": ["acl"],
            "properties": [
              { "name": "label", "label": "Label", "type": "string"},
              { "name": "description", "label": "Description", "type": "string"},
              { "name": "permissions", "label": "Permissions", "type": "rel", "target": "Permission", "to-string": "label"}
            ]
        },
        {
            "name": "Permission","to-string": "label", "labels": { "singular": "Permission", "plural": "Permissions" },
            "properties": [
              { "name": "label", "label": "Label", "type": "string"},
              { "name": "codename", "label": "Codename", "type": "string"}
            ]
        }
    ],
    "resources": {
        "tags": [
            "cors",
            "refresh-token", "invalidate-token", "revoke-token",
            "permission", "role", "role-permission"
        ],
        "cors": ["localhost", "www.example.com.br"]
    }
}
```

Figura 5. Exemplo de metadado

O metadado descrito na Figura 5 implementa todas as características que estão contidas no modelo desenvolvido anteriormente, exceto a comunicação cifrada, pois durante o processo de desenvolvido, percebemos que para obter essa característica, seria necessário criar templates fora do escopo do *spring boot* e *angular*. Caso o desenvolvedor deseje não implementar alguma característica opcional, basta removê-la da chave *"tags"*, dessa forma, todas as configurações se tornam declarativas. A partir disso, basta executar o comando *"pot angular:generate"* ou *"pot spring-boot:generate"* para o *Potter* inicializar todo o processo de geração de código. Caso seja necessário gerar apenas o código do *Spring Boot* por exemplo, basta executar apenas o comando do pertencente ao *Spring Boot*, pois apesar de compartilharem o mesmo metadado, o *Potter* interpreta cada um de uma forma, dependendo da tecnologia (*Spring boot* ou *Angular*).

Após o término das implementações, realizamos uma contagem de linhas de código para observar melhor a complexidade do metadado. O resultado pode ser visualizado na Tabela 2.

Para realizar a contagem no *spring boot* foi considerado apenas a quantidade de linhas de código dentro dos métodos relacionado a sua característica. Os resultados que estão com sinal de “>” significa que, a quantidade de código não pode ser mensurada de maneira finita, como por exemplo as Permissões, onde a entidade *permission* pode conter N atributos, dependendo da modelagem do sistema. Nesse cenário, a única coisa que se sabe é que por *default* ela deve possuir no mínimo 4 linhas de código.

Tabela 2 - Comparação de linhas de código com o metadado e *Spring Boot*

	Linhas de código *Spring boot*	Linhas de código metadado
cors	5	2
Autenticação	19	1
Refresh Token	19	1
Invalidação Token	25	1
Revogação Token	25	1
Permissões	> 19	> 4
Papéis	> 19	> 4
Permissões -> Papéis	> 38	> 4

5. Validação

Para validar a solução, um experimento foi elaborado contendo 7 aplicações, implementadas do zero, com características de segurança distintas. Uma combinação utilizada por exemplo,

foi um código que possuía a implementação apenas de *cors* e *roles*. Além das 7 aplicações desenvolvidas do zero, também foi feita 7 combinações de metadados do *Potter* com características de segurança iguais às das sete aplicações implementadas. A partir desses 7 metadados e dos templates desenvolvidos para este experimento, é possível gerar exatamente o mesmo código das aplicações. Após isso, foram selecionadas 6 pessoas, 4 alunos de graduação da área de TI e dois desenvolvedores de software experientes.

O experimento consistiu em medir o tempo em que cada desenvolver conseguia descobrir quais características estavam implementadas no código disponibilizado e quais características eram implementadas também pelo metadado. Os códigos utilizados para realizar o comparativo foram desenvolvidos utilizando o *framework Spring Boot*, mesmo *framework* utilizado para criação dos templates deste trabalho. Antes de iniciar o experimento, foi realizado um treinamento com cada indivíduo para nivelar o conhecimento nas tecnologias e torna o experimento o mais justo possível. Além disso, a distribuição dos códigos para cada pessoa foi feita de forma aleatória. No total, foram sorteados para cada pessoa 4 combinações, 2 códigos em *Spring Boot* e 2 metadados.

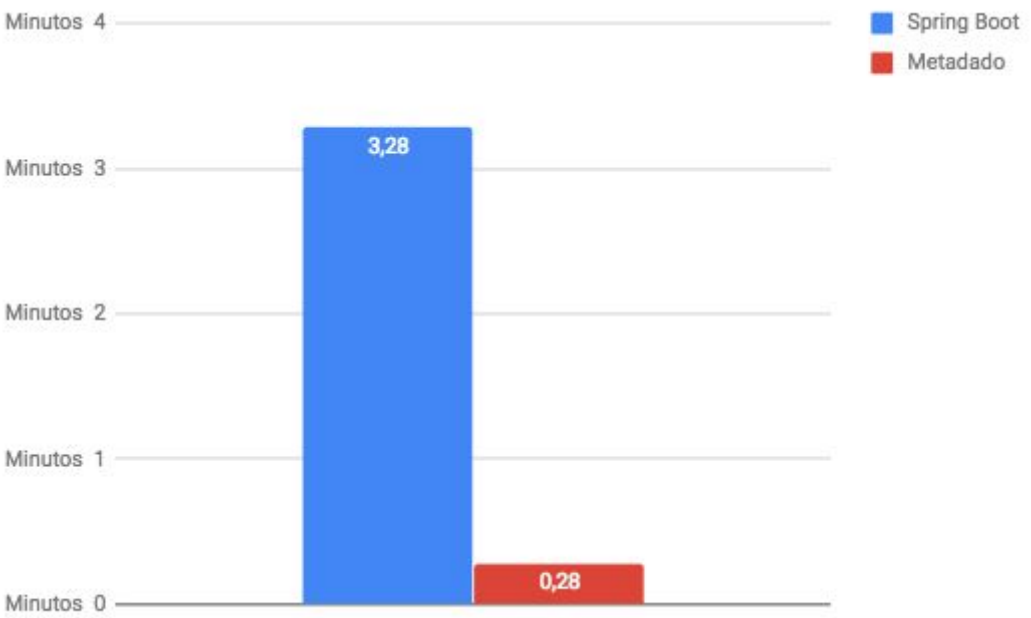

Figura 6. Tempo médio de respostas

No Gráfico 1 é possível perceber que existe uma diferença muito grande no tempo de resposta entre o código e o metadado disponibilizado. Nesse sentido, a forma como o metadado foi modelado se mostrou bastante amigável para os desenvolvedores.

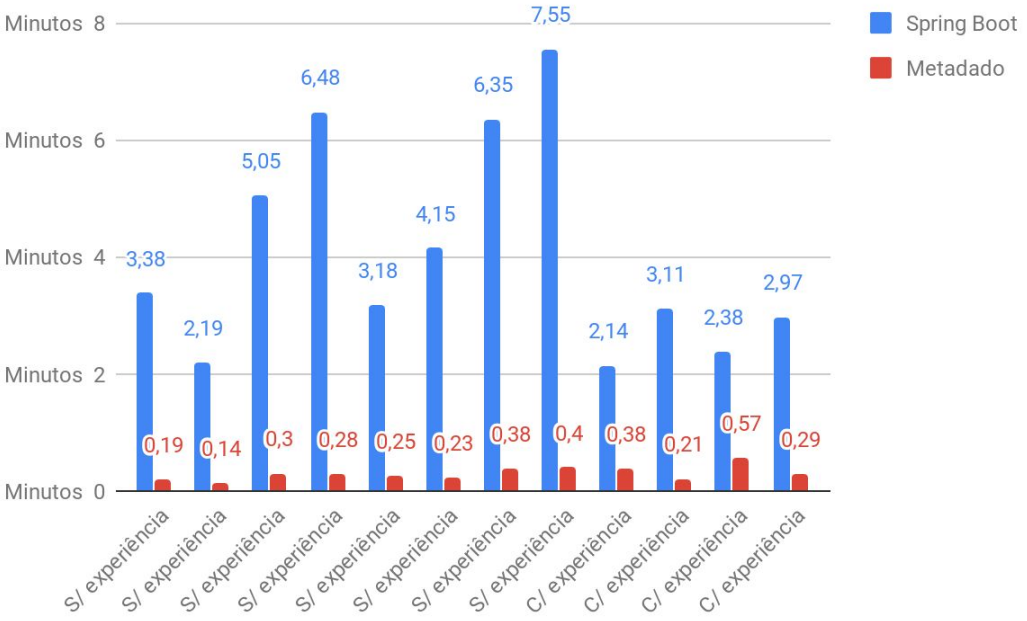

Figura 7. Tempo individual das respostas

Nas Figuras 7 e 8, se pode notar que desenvolvedores sem experiência em desenvolvimento de software conseguiram obter basicamente os mesmos tempos que desenvolvedores que já possuíam experiência, considerando apenas a análise dos metadados.

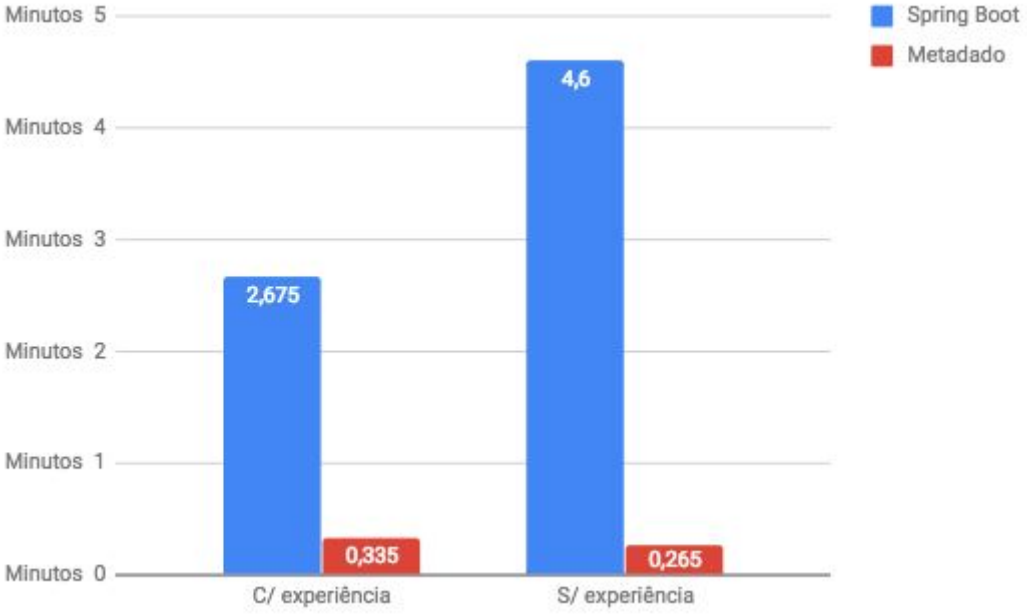

Figura 8. Tempo médio considerando desenvolvedores com e sem experiência

6. Conclusão e trabalhos futuros

Com base na análise dos resultados do experimento na Seção anterior, podemos concluir que, gerar código de componentes de segurança para sistemas de informação web a partir de metamodelos pode ser considerado uma alternativa viável para aumentar a produtividade de desenvolvimento, pois, além de ser possível gerar código de todas as entidades do sistema, também é possível gerar todo o código de controle de acesso das mesmas e várias outras características que foram citadas durante o trabalho.

Outro ponto importante a ser notado é que, a complexidade para compreender e preencher o metadado com as configurações necessárias é muito baixa. Além disso, os nomes de algumas características de segurança são bastante intuitivas, como por exemplo: Roles, Permissions. Dessa forma, mesmo que o desenvolvedor não possua experiência com o assunto, o próprio nome da característica já o induz a preencher o metadado com o requisito desejado. Portanto, o metadado se comporta com a mesma eficiência para desenvolvedores de níveis diferentes.

Como trabalhos futuros, pretendemos criar templates de código para outras linguagens de programação além de *Java*, como *Python* e *Javascript*. Um ponto interessante nisso é que os templates não possuem dependências entre si, então podemos gerar código com algumas combinações de *frameworks*, como por exemplo: *Angular* + *Python*[7], *Angular* + *Node*[8] *ou React*[9] + *Spring boot*.

[7] https://www.python.org/
[8] https://nodejs.org/en/
[9] https://reactjs.org/

7. Referências

Auth0. (2018). Auth0 tokens. Acesso em 21 de 08 de 2018, disponível em: https://auth0.com/docs/tokens/

FERNANDEZ-BUGLIONI, Eduardo. Security patterns in practice: designing secure architectures using software patterns. John Wiley & Sons, 2013.

HUANG, Lin-Shung et al. Protecting browsers from cross-origin CSS attacks. In: Proceedings of the 17th ACM conference on Computer and communications security. ACM, 2010. p. 619-629.

MAGNO, Danillo Goulart. Aplicação da Técnica de Scaffolding para a Criação de Sistemas CRUD. 2015.

R. S. Sandhu, E. J. Coyne, H. L. Feinstein and C. E. Youman, "Role-based access control models," in *Computer*, vol. 29, no. 2, pp. 38-47, Feb. 1996.

SHINDE, Kshitija; SUN, Yu. Template-Based Code Generation Framework for Data-Driven Software Development. In: 2016 4th Intl. Conf. on Applied Computing and Information Technology (ACIT), 3rd Intl. Conf. on Computational Science/Intelligence and Applied Informatics (CSII), and 1st Intl. Conf. on Big Data, Cloud Computing, Data Science & Engineering (BCD). IEEE, 2016. p. 55-60.

Stackoverflow, (2018). Developer Survey Results. Acesso em 21 de 08 de 2018, disponível em: https://insights.stackoverflow.com/survey/2018

VILAR, Rodrigo; OLIVEIRA, Delano; ALMEIDA, Hyggo. Rendering patterns for enterprise applications. In: Proceedings of the 20th European Conference on Pattern Languages of Programs. ACM, 2014. p. 22.