Segurança na comunicação entre Aplicações Web e REST

Pedro L. P. Vieira[1], Daves M. S. Martins[1]

[1]Centro de Ensino Superior de Juiz de Fora (CESJF) - Juiz de Fora – MG - Brasil

peluprvi@gmail.com, davesmartins@gmail.com

Abstract. *This article presents the preocupations and actions related to safety on web systems. A case study was developed envolving the integration between a Single Page Application and one RESTful Webservice, developed respectively using the frameworks Angular JS and Phalcon. Through this last study we can approach several problems of safety which envolve communication and information exchange between web systems.*

Resumo. *Este artigo apresenta as preocupações e as medidas relativas a segurança em sistemas Web. Um estudo de caso foi desenvolvido envolvendo a integração entre uma Aplicação de Página Única e um Webservice RESTful, desenvolvidos, respectivamente, utilizando os frameworks AngularJS e Phalcon. Através deste estudo podemos abordar os diversos problemas de segurança que envolve comunicação e troca de informações entre sistemas Web.*

1. Introdução

O Projeto Aberto de Segurança de Aplicação para Internet (OWASP, do inglês *The Open Web Application Security Project*), cuja missão organizacional é informar os reais riscos de segurança de software a indivíduos e organizações, trata o desenvolvedor como a peça chave de qualquer aplicação, sendo necessário adotar diversas técnicas de codificação segura tendo em mente todos os níveis da aplicação, interface do usuário, lógica de negócio, etc.

No contexto de aplicação, é necessário garantir a segurança das operações e da comunicação, assegurando ao usuário o acesso restrito às áreas e ações que lhe são de direito. Uma das soluções mais empregadas na atualidade, pode ser realizada por meio de troca de chaves de acesso que identificam ou limitam o acesso conforme o nível, grupo ou tipo do usuário, também conhecida como *token* de acesso.

Os *tokens*, segundo OWASP (2015) visam fornecer ao usuário a segurança da informação, baseado principalmente na confidencialidade, o acesso à informação será concedido a quem de direito, e na integridade caracterizada por manter a informação com todas as suas características originais, ou seja, ninguém alterar sua mensagem durante uma comunicação.

O presente trabalho apresenta uma discussão sobre a segurança de aplicações *Web*, com o foco em Aplicações de Página Única (SPA, do inglês *Single Page Application*). Trata-se de uma base para projetos *Web*, desenvolvida como estudo de caso de integração entre uma SPA e um *Webservice* no padrão de Transferência de Estado Representacional (REST, do inglês *Representational State Transfer*), que tem como objetivo principal apresentar alguns problemas de segurança na utilização de uma SPA e como implementar medidas para melhoria da segurança da aplicação. Para

abranger grande parte dos processos de segurança, será apresentado um estudo de caso, no qual uma arquitetura para autenticação e controle de acesso foi desenvolvida realizando a identificação do usuário via chave única de acesso (ou no inglês, *token*).

A arquitetura de autenticação e controle de acesso desenvolvida utiliza o AngularJS, *framework frontend*[1] preparado para uma SPA, e o Phalcon, *framework backend*[2] implementado na linguagem de programação C. Esta integração servirá de base para validar questões de segurança durante requisições de conteúdo e chamadas de ações pelo usuário.

2. Referencial Teórico

Segundo Hales (2012), fazia sentido a criação de aplicativos fortemente vinculados ao servidor, dando a liberdade aos desenvolvedores *backend* de não se preocupar com a manipulação do Modelo de Objeto de Documento (DOM, do inglês *Document Object Model*) e Folhas de Estilo (CSS, do inglês *Cascading Style Sheets*). Hoje, o desenvolvimento do lado do cliente exige claramente um maior investimento em aplicações de Linguagem de Marcação de Hipertexto (HTML, do inglês *HyperText Markup Language*) para Internet. Com a evolução dessas aplicações, estamos passando por uma séria mudança entre os tradicionais *frameworks* no lado do servidor, com rigorosas regras de lógica de modelagem e pesado processamento, para clientes JavaScript fracamente acoplados que podem ir de *online* a *offline* a qualquer momento.

DOM define um modelo de plataforma neutra para eventos de forma hierárquica (Figura 1), permitindo programas a acessar e manipular dinamicamente o conteúdo, a estrutura ou o estilo dos documentos (em particular, documentos HTML e XML). O termo "documento" é usado em recursos à base de marcação, que vão desde curtos documentos estáticos a longos ensaios ou relatórios com multimídia e aplicações interativas plenamente desenvolvidas. (W3C, 2015). Neste sentido a W3C é a entidade que padroniza sua implementação e forma de leitura.

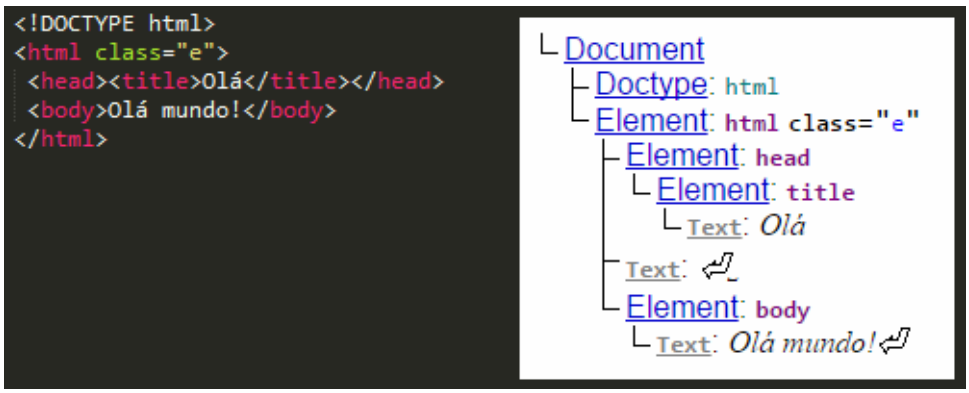

Figura 1. Estrutura HTML e sua representação em forma de DOM

Fonte: Autor

[1] O termo *frontend*, abordado neste artigo, se refere a tecnologias empregadas, visíveis ou processadas no cliente, como: HTML, CSS e JavaScript.
[2] O termo *backend*, abordado neste artigo, se refere a tecnologias empregadas, visíveis ou processadas no servidor, como: PHP e Banco de Dados.

Mikowski (2014) diz que o custo de produtividade de sites tradicionais são espantosos e para o negócio pode ser devastador. Uma das razões de sites tradicionais serem lentos é que os *frameworks* populares que utilizam o padrão de projeto Modelo, Visão e Controlador (MVC, do inglês *Model-View-Controller*) do lado do servidor são focados em carregamento de páginas de conteúdo estático. A cada requisição de página é necessário então carregar todo o conteúdo novamente (Figura 2), o que pode gerar uma experiência desagradável ao usuário.

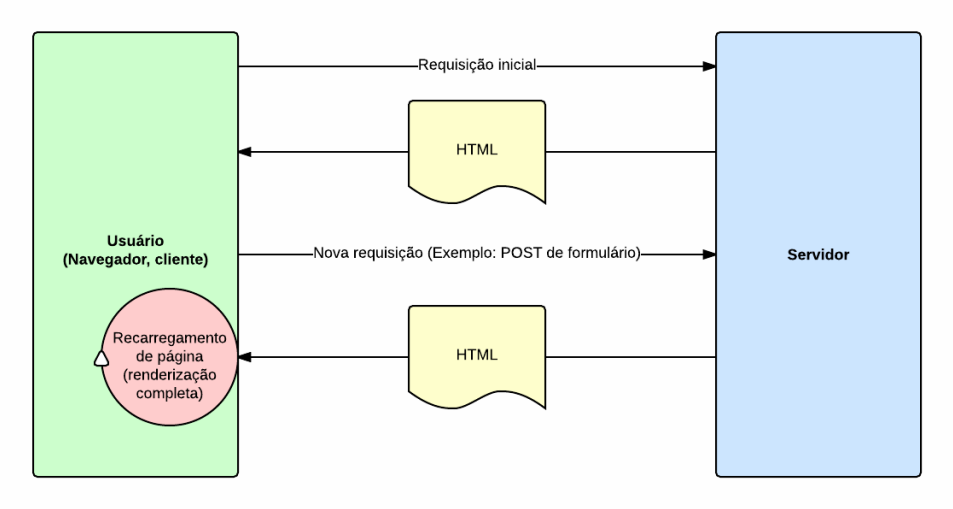

Figura 2. Ciclo de vida de sites tradicionais

Fonte: Autor

Atualmente, o conceito trabalhado no lado do cliente é chamado de SPA. Nesse conceito o navegador se torna uma aplicação *desktop* e, como resultado, gera uma experiência altamente responsiva, tornando a melhor escolha para otimizar a experiência do usuário. Desta forma Mikowski (2014) justifica a escolha de SPA em aplicações onde o fator decisivo de sucesso é o crescente *design* de produto, onde o foco é concentrado na experiência do usuário.

Para que tudo dito anteriormente funcione, é necessário a utilização de recursos avançados, que incluem o modelo de Notação de Objeto JavaScript (JSON, do inglês *JavaScript Object Notation*) e AJAX (do inglês *Asynchronous JavaScript and XML*) para a padronização na troca de mensagens com o servidor. Neste sentido, apenas o conteúdo solicitado é carregado, removendo a carga de processamento da Visão por parte do servidor e reduzindo drasticamente o tamanho do conteúdo da resposta.

Douglas Crockford (2015) descreve originalmente o JSON, no memorando RFC 4627 da IETF (do inglês *Internet Engineering Task Force*), como um formato leve, baseado em texto, independente de linguagem de troca de dados. Derivado dos objetos literais de JavaScript, define um pequeno conjunto de regras de formatação para representação portátil de dados estruturados. Pode representar quatro tipos primitivos (*strings*, números, booleanos e nulo) e dois tipos de estruturas (objetos e *arrays*). A última revisão desta especificação foi realizada por Tim Bray em Março de 2014.

O termo AJAX foi criado em 2005 por Jesse James Garrett. AJAX não é uma tecnologia única licenciada, mas uma coleção de tecnologias, principalmente JavaScript,

CSS, DOM e recuperação de dados do servidor de forma assíncrona. É usado no HTML dinâmico (DHTML, do inglês *Dynamic* HTML) para que o cliente possa recuperar informações sem a necessidade de recarregar a página (Figura 3), transformando a experiência do usuário de "visualizar páginas" para "interagir com uma aplicação". (SOUDERS, 2007).

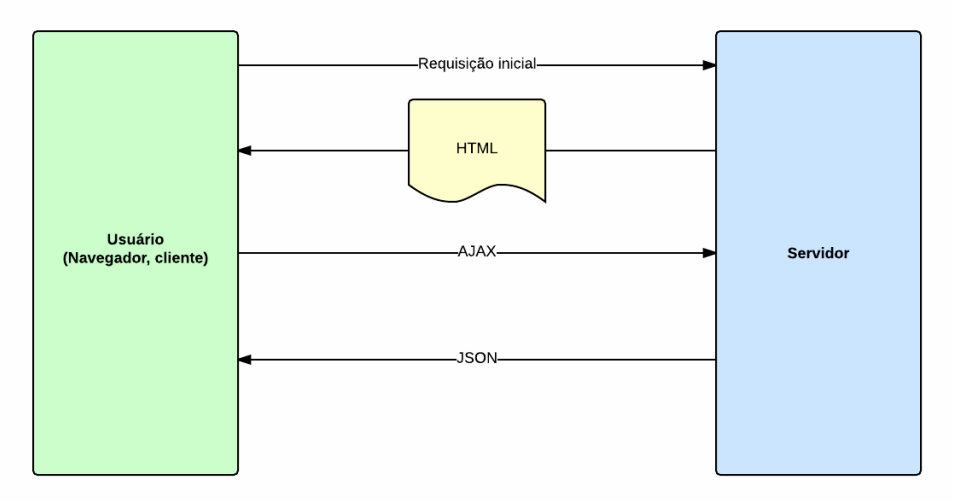

Figura 3. Ciclo de vida de SPA

Fonte: Autor

Segundo Green (2013), o padrão de projeto MVC separa a camada de Modelo para o gerenciamento dos dados, o Controlador para a lógica da aplicação e a Visão para apresentar os dados ao usuário. Em aplicações construídas em AngularJS, a camada de visão é o DOM, os controladores são as classes JavaScript, e os dados do modelo são armazenados em propriedades de objeto.

O *framework* AngularJS auxilia na criação de aplicações dinâmicas para a Internet, que resultam em um ambiente expressivo, legível e de rápido desenvolvimento, pois trata a manipulação do DOM de forma diferente das outras soluções disponíveis, eliminando essa preocupação do desenvolvedor. Adicionalmente, utiliza o padrão de projeto MVC, o que gera uma maior reutilização de componentes e melhor manutenção de código.

A estrutura do AngularJS foi desenvolvida para minimizar a duplicidade e a complexidade no código final da camada de apresentação em HTML, facilitando sua compreensão e estilização por *designers*. Um outro efeito benéfico deste *framework* é a redução do tamanho do código gerado, sendo um ponto muito importante se tratando de aplicações para internet, impactando positivamente e visivelmente no tempo de carregamento do documento.

Para tratar as operações de Criação, Leitura, Atualização e Remoção (CRUD, do inglês *Create, Read, Update, Delete*), Mikowski (2014) indica, como troca de dados entre o cliente SPA e o servidor, o padrão de projeto REST, que utiliza uma estrutura bem definida dos métodos do HTTP/1.1 (GET, POST, PUT, PATCH e DELETE), descritos por Fielding (2015). As aplicações que utilizam o padrão REST são chamadas de aplicações RESTful e deve-se utilizar os códigos padrões de retorno no HTTP/1.1, sendo esses os mais comuns:

- Códigos de retorno no formato 2xx são gerados em caso de sucesso na requisição, garantem que a requisição foi recebida, compreendida e aceita:
  - 200 - *OK*: Requisição bem sucedida
  - 201 - *Created*: A requisição foi realizada e resultou na criação de um ou mais novos recursos. Geralmente se identifica o recurso criado
- Códigos de erros no formato 4xx podem ser gerados na aplicação no lado do servidor e não são gerados automaticamente. Indicam possíveis erros do usuário:
  - 400 - *Bad Request*: Erro no corpo da requisição
  - 401 - *Unauthorized*: O usuário não foi autorizado a realizar a requisição pois não está autenticado, mas pode ganhar acesso posteriormente à requisição caso se autentique
  - 403 - *Forbidden*: O usuário não é autorizado a realizar esta requisição e o processo de autenticação não dará direito de acesso. Geralmente usado quando o usuário já está autenticado e, portanto, não se deve repetir a requisição que gerou este erro
  - 404 - *Not Found*: O conteúdo requisitado não foi encontrado. É possível especificar a razão no corpo da mensagem
  - 405 - *Not Allowed*: O método HTTP solicitado não é um recurso válido
  - 409 - *Conflict*: Conflitos entre a requisição e o atual estado do recurso. Deve-se listar os problemas no corpo do retorno
  - 413 - *Request Entity Too Large*: Corpo da requisição pelo método POST ou PUT extrapola o tamanho limite. Deve-se detalhar o retorno fornecendo alternativas
  - 415 - *Unsuported Media Type*: Formato do corpo da requisição não foi compreendido
- Códigos de erros no formato 5xx podem ser gerados de forma automática em erros de requisição por algum erro no servidor
  - 500 - *Internal Server Error*: Erro interno no servidor
  - 503 - *Service Unavailable*: Erro gerado quando o servidor não consegue responder uma requisição após um intervalo de tempo

Allamaraju (2010) reforça que os números são padrões, mas a mensagem informativa dos códigos de retorno pode ser alterada, assim como é possível gerar um conteúdo personalizado no corpo da resposta, seja em caso de sucesso ou erro na requisição.

OWASP (2015) lista as principais técnicas de segurança que devem ser aplicadas no projeto e no cotidiano do desenvolvimento de um *software*, sendo eles, por ordem de importância:

- C1 - Parametrizar consultas ao banco de dados (do inglês *queries*)
- C2 - Codificar dados

- C3 - Validar todas as entradas
- C4 - Implementar controles de acesso apropriados
- C5 - Estabelecer controles de autenticação e identidade
- C6 - Proteger dados e privacidade
- C7 - Implementar registro de ações (do inglês *logging*), tratamento de erros e detecção de intrusão
- C8 - Identificar o nível de segurança das características dos *frameworks* e bibliotecas de segurança
- C9 - Incluir requisitos específicos de segurança
- C10 - Projetar e arquitetar a segurança interna

OWASP (2015) também informa sobre a necessidade e boas práticas para garantir que apenas pessoas autorizadas realizem ações permitidas em seu nível de privilégio. Não podendo então confiar em dados sobre o nível de acesso enviados pelo usuário, sendo fundamental consultá-los sempre que necessários em uma base de dados.

A segurança de acesso, segundo Allamaraju (2010), requer que:

- Somente usuários autenticados possam acessar recursos
- Garantir a confidencialidade e integridade das informações
- Evitar abuso de recursos e dados de forma não autorizada
- Manter a privacidade e leis vigentes

Allamaraju (2010), também descreve uma forma de autorização simples, com a utilização de *token* (chave única gerada por uma chave secreta na rotina de acesso e autenticação do sistema), onde, através deste identificador do usuário é possível validar as credenciais do mesmo, permitindo ou negando o acesso ao recurso ou ação solicitada

Além da identificação do usuário, OWASP (2015), também informa a necessidade de prevenir os ataques XSS (do inglês *Cross-Site Scripting*), CSRF (do inglês *Cross-Site Request Forgery*) e outros. As principais vulnerabilidades, em ordem do maior para o menor risco, segundo OWASP, são:

- A1 – Injeção (do inglês *Injection*)
- A2 - Quebra de autenticação e gerência de sessão
- A3 - *Cross-Site Scripting* (XSS)
- A4 - Referências diretas a objetos sem segurança
- A5 - Configuração incorreta da segurança
- A6 - Exposição de dados sensíveis
- A7 - Falta de função de controle de nível de acesso
- A8 - *Cross-Site Request Forgery* (CSRF)
- A9 - Uso de componentes com vulnerabilidades conhecidas

•A10 - Redirecionamentos e encaminhamentos não validados

Através da injeção, o banco de dados pode ficar aberto ao usuário, que executará comandos através de entrada de dados e quebrará a confidencialidade e integridade do sistema.

Ataques XSS ocorrem quando é permitido ao usuário inserir códigos maliciosos em entradas de dados, no preenchimento de formulários ou parâmetros, não tratadas e que podem ser executadas no carregamento de uma página por um outro usuário. Neste tipo de ataque o código malicioso é considerado como confiável pelo navegador pois é interpretado como sendo um código da própria aplicação e, portanto, pode ter acesso a informações sigilosas da vítima.

Já o ataque por CSRF ocorre em *website*, *email*, *blog*, mensageiro instantânio ou programas que fazem com que o navegador da vítima execute uma ação indesejada em uma aplicação onde o usuário se encontra autenticado, podendo comprometer o usuário final e suas funções. Em casos onde a vítima é administrador da aplicação, o ataque pode comprometer todo o sistema.

Para mitigar os riscos de segurança, cada técnica de segurança previne uma ou mais vulnerabilidades. Este mapeamento pode ser analisado na Tabela 1.

Tabela 1. Mapeamentos das principais técnicas de segurança contra os principais riscos de segurança

	A1	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**A10**
C1	X									
C2	X		X							
C3	X		X							X
C4				X			X			
C5		X								
C6						X				
C7	X	X	X	X	X	X	X	X	X	X
C8	X	X	X	X	X	X	X	X	X	X
C9	X	X	X	X	X	X	X	X	X	X
C10	X	X	X	X	X	X	X	X	X	X

Fonte: Autor

Segundo Google (2015), o AngularJS trata por padrão validações contra os riscos A1, A2, A3 e há uma possível configuração para o A8. O *framework*, suportado pela Google, é um dos mais novos no mercado e surge quebrando paradigmas na manipulação do DOM de forma automática, tornando-o a primeira opção entre os desenvolvedores. No servidor, o Phalcon (2015), previne contra os riscos de segurança A1 e A4 de forma padrão e, por ser uma extensão ao PHP implementado em C, é considerado a melhor escolha para aplicações que dependem de uma resposta rápida sem perder os benefícios do uso da linguagem PHP no desenvolvimento. Em ambos os *frameworks*, os demais riscos devem ser tratados manualmente via código ou configuração, de acordo com a necessidade da aplicação.

3. Problema

Para aprimorar a experiência do usuário, uma SPA traz para o lado do cliente algumas responsabilidades, como: roteamento, carregamento dinâmico de pequenos templates ou partes de código quando necessário, tratamento direto com variáveis, dentre outras funcionalidades. É necessário garantir que cada funcionalidade seja acessada apenas aos usuários que tenham direito de acesso. Antes de finalizar qualquer ação, uma inclusão por exemplo, é necessário validar as credenciais do usuário no servidor. Essa preocupação se torna ainda maior se tratando de uma ação em que apenas pessoas autenticadas e autorizadas possam realizar.

Ao lado do servidor, cada função do *Webservice* deve ser tratada individualmente com a finalidade de garantir integridade, confiabilidade e disponibilidade dos dados.

Segundo OWASP (2015), diversas questões de segurança devem ser tratadas no processo de autenticação e autorização de acesso e podem ser consideradas críticas à aplicação, como a utilização de *token* e a forma de construí-lo, a garantia da legitimidade do usuário nos processos onde apenas usuários conhecidos tenham privilégio e a proteção contra XSS e CSRF. Além desses riscos, o desenvolvedor deve estar ciente sobre as limitações das ferramentas e *frameworks* utilizados na aplicação e garantir que todos os riscos, tratados de forma automática ou manual, sejam mitigados no processo de desenvolvimento do sistema.

4. Estudo de Caso

Desenvolver uma arquitetura de autenticação e controle de acesso por token entre uma SPA, desenvolvida em AngularJS, e um *Webservice* RESTful, desenvolvido em Phalcon, demonstrará possíveis soluções aos principais riscos de segurança de sistemas, além de exemplificar o uso dessas tecnologias em aplicações seguras.

Através da criação desta arquitetura de integração entre uma SPA e um *Webservice* RESTful, para controle e autorização de acesso, as tecnologias envolvidas são compreendidas de forma clara, prevenindo os riscos sobre configuração incorreta de segurança e uso de componentes com vulnerabilidades conhecidas.

A arquitetura em questão permitirá, através do reuso ou refatoração, o aproveitamento do código em pequenas ou grandes aplicações, acelerando o desenvolvimento e garantindo uma base inicial com um mínimo de tratamento de segurança.

Nakstad (2015) demonstra uma solução que considera o acesso restrito a páginas individuais e customização diferenciada por nível de acesso, porém não implementa uma solução para ações, onde uma página poderia ter ações diferenciadas ou restritas por tipo de usuário.

Utilizar o máximo do potencial das tecnologias gratuitas disponíveis para esta integração e englobar uma maior quantidade de soluções no processo de autorização e autenticação oferecerá o ambiente ideal para estudar os riscos de segurança envolvidos e encontrar suas possíveis soluções. Serão tratados os riscos de ataques XSS e CSRF, considerados comuns entre aplicações *Web* e *Webservice* RESTful, que comumente dependem de configurações, e estudados os demais riscos de segurança que são tratados

de forma automática pelos *frameworks* apresentados ou que dependem de uma total implementação.

4.1 Solução

O *Webservice* RESTful foi desenvolvido em Phalcon, *framework* para PHP desenvolvido em linguagem C, que por padrão realiza diversos tratamentos de segurança para aplicação. As informações são recebidas pelo *Webservice* através do corpo das requisições, que são definidas com caminho e método de chamada no padrão HTTP. Toda troca de informações entre o AngularJS e o *Webservice* é feita por AJAX e o conteúdo é em formato JSON. Um erro é retornado em qualquer requisição não esperada. Maiores detalhes sobre uma requisição, e sua resposta, podem ser visualizadas na Figura 4.

Para garantir que o usuário realize ações e visualize somente as informações pertinentes ao seu nível de acesso, o usuário precisa ser identificado via processo de autenticação e autorização, também chamado de *login*. O *token* de identificação é gerado pelo sistema após autenticação do usuário. Neste processo o usuário informa seu nome de usuário e senha e o sistema valida se os dados estão conferem com algum usuário existente. Estando válido, o *token* é gerado através do nome do usuário e outras variáveis de identificação ou distinção, como local e hora de acesso. Desta forma o *token* só será válido em um mesmo dispositivo e mesma rede de acesso. Um processo de criptografia é usado, informando uma variável aleatória e uma chave secreta da aplicação, para que o *token* seja único e ilegível.

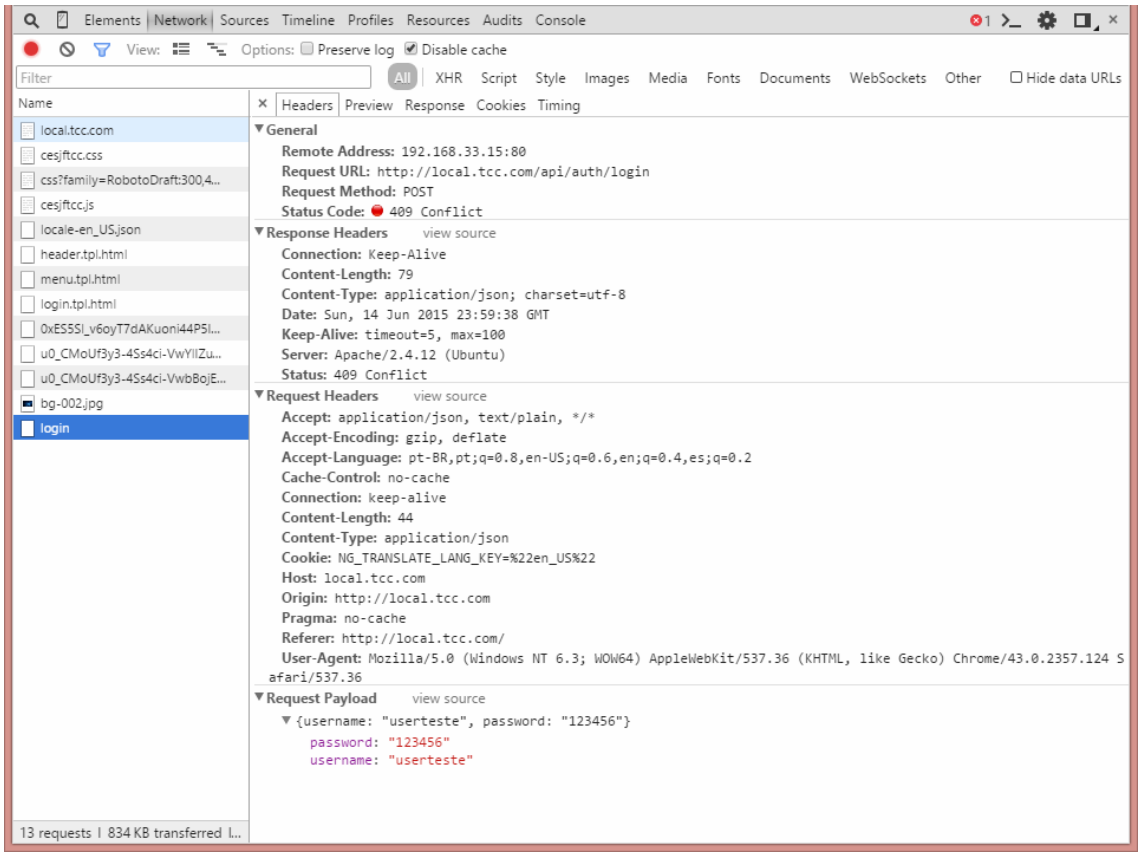

Figura 4. Detalhamento da requisição e da resposta, com erro HTTP/1.1 409, durante uma requição de método POST em AJAX para uma tentativa de acesso inválida

Fonte: Autor

Após identificado, há uma troca de *token* em toda requisição onde o *Webservice* valida a legitimidade do *token* e retorna uma resposta positiva ou negativa à ação solicitada. Em ambos os casos há um conteúdo de resposta que será processado no AngularJS e uma reação é disparada, podendo ser o carregamento de uma lista, conclusão de uma ação, ou exibição do erro retornado.

O *token* é validado em toda requisição de página em que apenas usuários identificados podem ter acesso (Figura 5) e, caso a resposta seja negativa, o *token* inválido é limpo, o usuário então é deslogado e é redirecionado para a tela de login. Em caso positivo a página solicitada e seu conteúdo são carregados normalmente, ambos via AJAX.

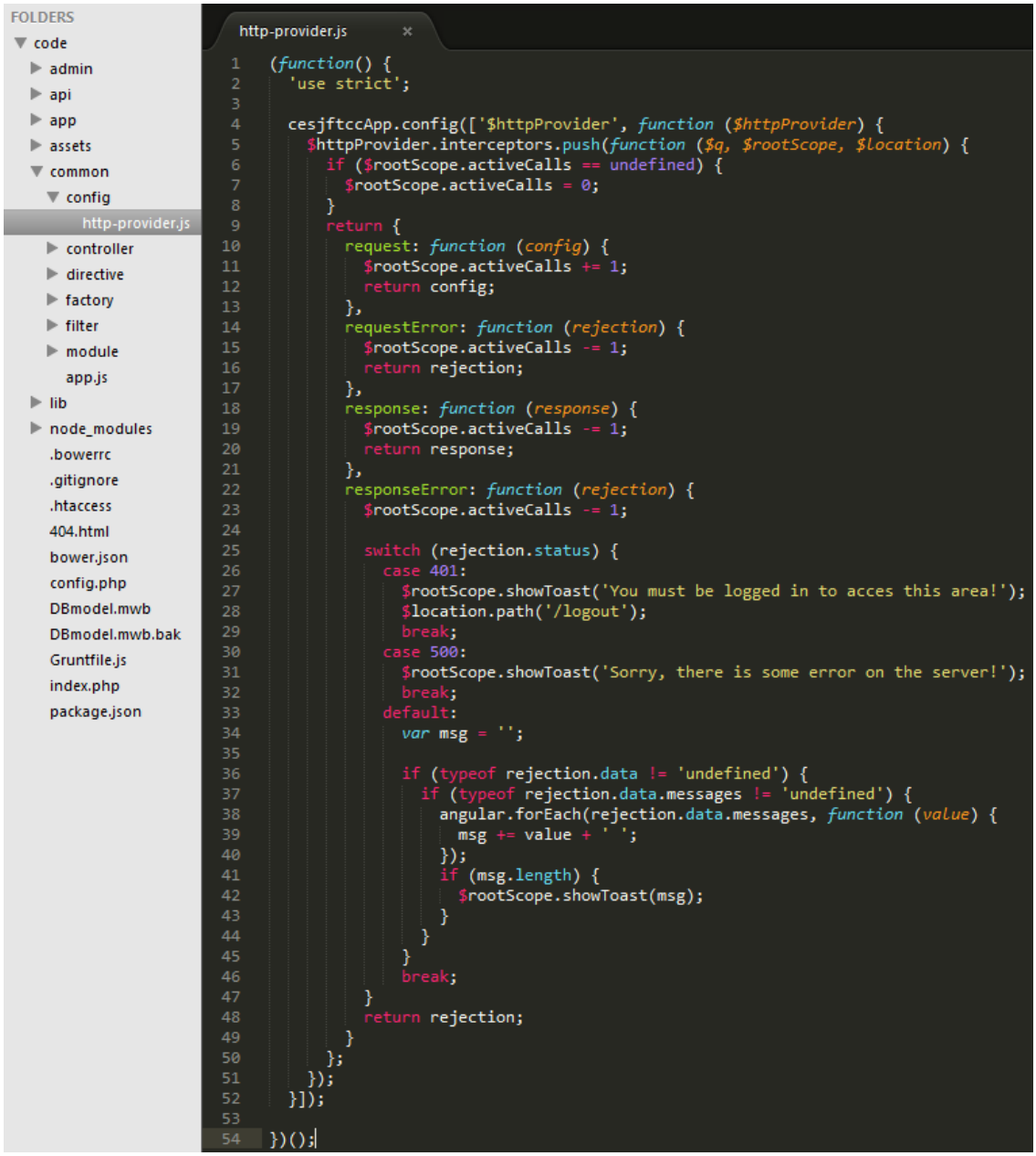

Figura 5. Interceptador de requisições com tratamento de erros de resposta

Fonte: Autor

O processo de validação do *token* é realizado em duas etapas, detalhado da Figura 6 A primeira validação é realizada pelo AngularJS, onde é verificada a existência de um *token* no cliente, independente do conteúdo da chave de acesso. Em caso positivo, um AJAX de validação do *token* é disparado ao servidor. Antes de iniciar a validação necessária, o servidor se encarrega de limpar a lista de *token* expirados, ou que não foram usados a determinado tempo, mantendo assim apenas *tokens* recém criados. O servidor busca o *token* nesta lista final e, em caso positivo, retorna informações necessárias para identificação do usuário. Em caso de erro em qualquer parte deste processo, um erro é retornado, o *token* é limpo no cliente e o usuário é redirecionado para a página de *login*.

Ao realizar uma ação restrita, a requisição AJAX é enviada ao servidor contendo o *token* e a solicitação. Neste caso, o processo de validação do *token* é similar ao apresentado na Figura 6, sendo que o código de resposta HTTP, de sucesso ou erro, será de acordo com a solicitação e a resposta é tratada para cada ação ou requisição. Sendo assim, tratamentos específicos podem ser realizados de acordo com o tipo de erro ou sucesso retornado e mensagens explicativas podem ser trabalhadas para cada ação, gerando uma melhor experiência ao usuário.

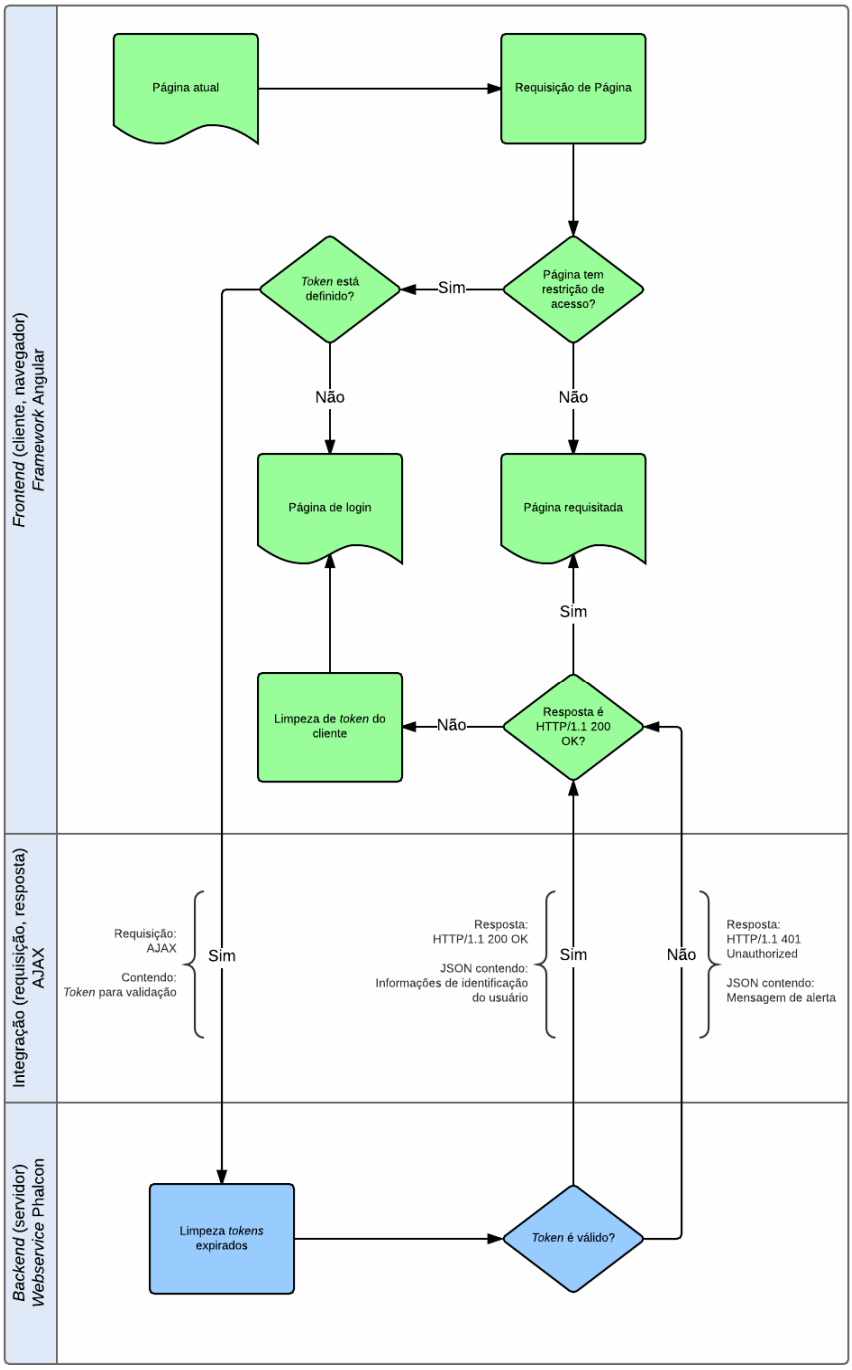

Figura 6. Detalhamento do processo de validação do *token* durante a solicitação de uma nova página

Fonte: Autor

Em ações restritas, a não existência do *token* no cliente gera um redirecionamento para a página de *login* sem que a requisição AJAX seja realizada, reduzindo o uso desnecessários de recursos do servidor. Em ações comuns, a validação do *token* não é necessária e, portanto, não é realizada no servidor.

Por padrão, o AngularJS trata todas as exibições de dados com segurança contra XSS, reduzindo a responsabilidade por parte do desenvolvedor, sendo desativado qualquer tratamento de variável como elementos HTML. Em alguns momentos o desenvolvedor pode precisar desses elementos, sendo necessário utilizar algumas funções, ou outros elementos do *framework* como o módulo ngBindHtml, que permitem esse tipo de tratamento. Nestes casos é vital para a segurança da aplicação a utilização do módulo ngSanitize no AngularJS, que impede que conteúdos inseguros possam ser executados.

O tratamento contra ataques CSRF é realizado em etapas no cliente e no servidor. Primeiramente, no AngularJS, deve-se ativar o módulo ngCookies e configurar um atributo padrão chamado XSRF-TOKEN que será enviado no cabeçalho de todas as requisições em AJAX consideradas restritas a usuários autenticados. Esse atributo pode ser atualizado a cada troca de página ou a cada número de ações através da resposta a alguma requisição. No servidor, Phalcon, durante o processo de criação do *token*, autenticação do usuário, uma chave inicial de CSRF é gerada e retornada ao cliente, que preencherá o atributo XSRF-TOKEN. No processo de validação do *token*, a chave CSRF anterior é enviada e, caso válida, uma nova é gerada e atualizada no atributo XSRF-TOKEN. Quando uma requisição restrita é enviada ao servidor, após o processo de validação de autorização do usuário, deve-se comparar o conteúdo da chave CSRF guardada com o XSRF-TOKEN enviado no cabeçalho. Caso não sejam idênticas, a requisição não pode ser realizada e uma mensagem de erro HTTP/1.1 401 Unauthorized deve ser retornada.

4.2 Resultados Obtidos

O AngularJS mostrou ter uma estrutura de fácil compreensão e baixa complexidade ao trocar requisições com o *Webservice*, agilizando o processo de desenvolvimento e manutenção do código. Como é um *framework frontend*, continua sendo necessário garantir a qualidade das funções no *Webservice* e, consequentemente, a integridade e a segurança dos dados.

Ambos, AngularJS e Phalcon, foram idealizados para tratar diversas questões de segurança, mas a responsabilidade da segurança da aplicação fica a cargo do desenvolvedor. Na documentação do AngularJS é possível encontrar boas práticas e recomendações de segurança, principalmente contra XSS e CSRF, sendo ainda necessário realizar algumas configurações, não devemos ser negligentes no desenvolvimento da aplicação. A ferramenta, ou o *framework*, não são responsáveis por todo tratamento de segurança.

5. Considerações Finais

O artigo abordou uma arquitetura de autenticação baseado em *token,* foram desenvolvidas e apresentadas soluções de segurança e controle de acesso em uma integração entre uma SPA e um *Webservice* RESTful.

Para a construção da solução, foram estudadas tecnologias atuais, abertas e gratuitas, observando o estado da arte das técnicas de desenvolvimento no lado do cliente.

Também foi apresentado um levantamento bibliográfico, mostrando as principais falhas de segurança, e as tecnologias aplicadas no desenvolvimento da arquitetura. O artigo mostrou os passos para a contrução de uma SPA, com o *framework* AngularJS, um *Webservice* RESTful, com o *framework* Phalcon, e a autenticação de usuários com controle de acesso.

Considerando os *frameworks* estudados neste artigo, a maior parte dos tratamentos aos principais riscos de segurança de aplicações *Web* ficam sobre responsabilidade do desenvolvedor e a utilização das técnicas de prevenção dependem diretamente da necessidade do projeto. Sendo assim, realizar a implementação de um algoritmo para autenticação, autorização e as demais técncas de prevenção, exige um alto nível de maturidade da equipe e da organização para que os resultados sejam satisfatórios.

Referências

ALLAMARAJU, Subbu. RESTful Web Services Cookbook. Sebastopol: O'Reilly Media, 2010. p. 217-234.

BRAY, Tim. The JavaScript Object Notation (JSON) Data Interchange Format. Disponível em: <http://tools.ietf.org/html/rfc7159>. Acesso em: 15 abr. 2015.

CROCKFORD, Douglas. The application/json Media Type for JavaScript Object Notation (JSON). Disponível em: <http://tools.ietf.org/html/rfc4627>. Acesso em: 15 abr. 2015.

FIELDING, R. et al. Hypertext Transfer Protocol (HTTP1.1): Semantics and Content. Disponível em: <http://tools.ietf.org/html/rfc7231>. Acesso em: 13 mai. 2015.

GARRET, Jesse James. Ajax: A New Approach to Web Applications. Disponível em: <http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications>. Acesso em: 15 abr. 2015.

GOOGLE. AngularJS – Superheroic JavaScript MVW Framework. AngularJS. Disponível em: <http://www.angularjs.org>. Acesso em: 15 abr. 2015.

GREEN, Brad; SESHADRI, Shyam. AngularJS. Sebastopol: O'Reilly Media, 2013.

HALES, Wesley. HTML5 and Javascript Web Apps. Sebastopol: O'Reilly Media, 2012. p.1-2, 5, 7, 64-65.

MIKOWSKI, Michael S.; POWELL, Josh C.. Single Page Web Applications. Manning, 2014.

NAKSTAD, Frederik. Authentication in Single Page Applications With Angular.js. Disponível em: <http://www.frederiknakstad.com/2013/01/21/authentication-in-single-page-applications-with-angular-js>. Acesso em: 15 abr. 2015.

OWASP. Cross-Site Request Forgery (CSRF). Disponível em: <https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)>. Acesso em: 14 mai. 2015.

OWASP. Cross-site Scripting (XSS). Disponível em: <https://www.owasp.org/index.php/XSS>. Acesso em: 13 mai. 2015.

OWASP. Guide to Authentication. Disponível em: <https://www.owasp.org/index.php/Guide_to_Authentication>. Acesso em: 8 fev. 2015.

OWASP. Guide to Authorization. Disponível em: <https://www.owasp.org/index.php/Guide_to_Authorization>. Acesso em: 8 fev. 2015.

OWASP. OWASP Proactive Controls. Disponível em: < https://www.owasp.org/index.php/OWASP_Proactive_Controls>. Acesso em: 12 jun. 2015.

OWASP. OWASP REST Security Cheat Sheet. Disponível em: <https://www.owasp.org/index.php/REST_Security_Cheat_Sheet>. Acesso em: 6 fev. 2015.

OWASP. OWASP Secure Coding Practices Quick Reference Guide. Disponível em: <https://www.owasp.org/images/0/08/OWASP_SCP_Quick_Reference_Guide_v2.pdf>. Acesso em: 6 fev. 2015.

PHALCON. High performance PHP framework. O mais rápido Framework PHP. Disponível em: <http://phalconphp.com/pt>. Acesso em: 15 abr. 2015.

SOUDERS, Steve. High Performance Web Sites: Essential Knowledge for Frontend Engineers. Sebastopol: O'Reilly Media, 2007. p.1, 97-98.

W3C. Document Object Model (DOM). Disponível em: <http://www.w3.org/TR/dom/>. Acesso em: 14 mai. 2015.