Revista de Sistemas e Software 203 (2023) 111723

Listas de conteúdo disponíveis no ScienceDirect

O Jornal de Sistemas e Software

página inicial do periódico: www.elsevier.com/locate/jss

Melhorando a observabilidade de aplicações web por meio de instrumentação Navegadores automatizadosÿ

Boni García [a,ÿ], Filippo Ricca [b], José M. del Alamo [c], Maurizio Leotta [b]

[a] Universidade Carlos III de Madrid, Madri, Espanha
[b] Universidade de Gênova, Gênova, Itália
[c] ETSI Telecomunicación, Universidade Politécnica de Madrid, Madrid, Espanha

informações do artigo

Histórico do artigo:
Recebido em 18 de junho de 2022
Recebido em formato revisado em 17 de janeiro de 2023
Aceito em 19 de abril de 2023
Disponível online em 5 de maio de 2023

Palavras-chave:
Automação do navegador
Coleta de registros
Estudo empírico

resumo

Na engenharia de software, observabilidade é a capacidade de determinar o estado atual de um sistema de software com base em suas saídas ou sinais externos, como métricas, logs ou rastros. Engenheiros web utilizam o console do navegador como a principal ferramenta para monitorar o lado do cliente de aplicações web durante testes de ponta a ponta. No entanto, essa é uma tarefa manual e demorada devido aos diferentes navegadores disponíveis. Este artigo apresenta o BrowserWatcher, uma extensão de navegador de código aberto que oferece recursos entre navegadores para observar aplicações web e coletar automaticamente logs do console em diferentes navegadores (por exemplo, Chrome, Firefox ou Edge). Utilizamos essa extensão para conduzir um estudo empírico analisando o console dos 50 principais sites públicos de forma manual e automática. Os resultados mostram que o BrowserWatcher coleta todas as categorias de log conhecidas, como console ou rastreamentos de erros. Também revela que cada navegador inclui outros tipos de logs, que diferem entre si, fornecendo, assim, informações distintas para o mesmo site.

1. Introdução

Observabilidade e monitoramento são frequentemente usados de forma intercambiável, embora existam pequenas diferenças entre esses termos. Por um lado, o IEEE define monitoramento como o processo de supervisão, registro, análise ou verificação da operação de um sistema ou componente (IEEE, 1990). Por outro lado, a observabilidade utiliza ferramentas de instrumentação para fornecer insights que auxiliam o monitoramento. Em outras palavras, o monitoramento pode ocorrer quando um sistema é observável (Niedermaier et al., 2019).

O termo observabilidade vem da teoria clássica de controle de Kalman (1970) e refere-se à capacidade de inferir o estado interno de um sistema por meio da coleta e análise de seus resultados externos (Niedermaier et al., 2019). Ela permite a compreensão do estado interno do sistema por meio de seus indicadores externos.

Os três pilares da observabilidade são:

- Métricas: medidas de desempenho do sistema ao longo do tempo, como tempo de resposta, transações por segundo ou uso de memória, para citar algumas.

- Logs: linhas de texto (geralmente com registro de data e hora) que um sistema produz ao executar um trecho de código.

- Rastros: representação de eventos distribuídos causalmente relacionados (como logs selecionados) que caracterizam o fluxo de solicitações de uma determinada operação em um sistema de software.

A observabilidade pode ser essencial para manter sistemas de software complexos e determinar a causa raiz de qualquer problema. Por exemplo, em testes web, quando um teste automatizado de ponta a ponta falha (por exemplo, com Selenium WebDriver[1]), e como todo o sistema é testado, geralmente é desafiador descobrir a causa subjacente do erro (García et al., 2020). Uma fonte relevante de informações para um teste de ponta a ponta com falha pode ser o log do navegador, pois ele pode conter rastros de erros do lado do cliente que permitem determinar a causa da falha. No entanto, a coleta de logs de navegadores web automatizados pode ser complexa devido à falta de ferramentas adequadas. Portanto, esse processo normalmente é manual, caro e sujeito a erros.

Para mitigar esse problema, este artigo apresenta o BrowserWatcher, uma nova extensão de navegador de código aberto projetada para observar aplicações web por meio da instrumentação de navegadores como Chrome, Firefox ou Edge. Seus recursos incluem coleta de logs do console entre navegadores, exibição de logs, gravação de abas (ou seja, captura da janela de visualização do navegador como um arquivo de mídia), desativação da Política de Segurança de Conteúdo (CSP) e injeção de JavaScript/CSS. O BrowserWatcher pode ser usado como uma extensão em qualquer navegador que implemente a Interface de Programação de Aplicativos (API) WebExtensions (Mozilla MDN, 2022a;

ÿ Editor: Earl Barr.

* Autor correspondente.
Endereços de e-mail: boni.garcia@uc3m.es (B. García), filippo.ricca@unige.it (F. Ricca), jm.delalamo@upm.es (JM del Alamo), maurizio.leotta@unige.it (M. Leotta).

[1] https://www.selenium.dev/documentation/webdriver/

https://doi.org/10.1016/j.jss.2023.111723 0164-1212/©

Chrome Team, 2022b) (Chrome, Firefox, Edge, etc.) ou integrado com ferramentas de teste automatizadas.

Para demonstrar sua aplicabilidade e avaliar sua eficácia, realizamos um estudo experimental no qual os logs dos 50 sites mais populares são coletados automaticamente usando os principais navegadores. Para comparar esses resultados com uma realidade, o console de cada navegador também é coletado manualmente. A comparação revela que cada navegador exibe vários tipos de logs, às vezes diferentes (ou seja, erros e avisos de diferentes categorias) e também mostra diferenças relevantes nos logs produzidos pelo mesmo site em diferentes navegadores.

O restante deste artigo está estruturado da seguinte forma. A Seção 2 explica o contexto e a motivação deste trabalho. Em seguida, a Seção 3 apresenta o conjunto de ferramentas proposto para aprimorar a observabilidade em testes automatizados de ponta a ponta em aplicações web. A Seção 4 fornece prova experimental da abordagem apresentada, analisando os logs do navegador dos 50 sites mais populares coletados automaticamente com nossa solução e também por meio de inspeção manual. À luz dos resultados obtidos neste experimento, a Seção 5 analisa o impacto da coleta de logs do navegador em testes de ponta a ponta e suas implicações em testes entre navegadores. Em seguida, a Seção 6 apresenta trabalhos semelhantes disponíveis na literatura atual. Por fim, a Seção 7 resume as descobertas e as etapas futuras deste trabalho.

2. Contexto e motivação

Esta seção apresenta a arquitetura do Selenium WebDriver, a biblioteca de testes automatizados considerada neste artigo. Em seguida, fornece um exemplo para explicar o problema enfrentado que motiva nosso trabalho.

2.1. Driver Web Selenium

Testes de software consistem na avaliação dinâmica de um software, denominado Sistema em Teste (SUT). Testes automatizados de software implicam o uso de ferramentas e frameworks específicos para implementar e executar casos de teste (ou simplesmente testes) em relação ao SUT (Bertolino, 2007). Dependendo do tamanho do SUT, existem diferentes níveis de teste. Alguns dos níveis mais relevantes são os testes unitários (ou seja, avaliar elementos individuais, como classes ou métodos), os testes de integração (ou seja, testar diferentes unidades como uma entidade combinada) e os testes ponta a ponta (ou seja, testar todo o sistema por meio de sua interface de usuário) (Quadri e Farooq, 2010).

Selenium WebDriver (frequentemente conhecido simplesmente como Selenium, como seu projeto principal) é uma biblioteca de código aberto que permite controlar navegadores da web programaticamente usando diferentes linguagens (como Java, JavaScript, Python, Ruby e C#) (García, 2022). Em outras palavras, o Selenium WebDriver é uma biblioteca que permite o desenvolvimento de testes ponta a ponta para aplicações web (Leotta et al., 2016a). Uma pesquisa recente sobre testes de software reconheceu o Selenium como o framework de testes mais valioso, seguido pelo JUnit e pelo Cucumber (Cerioli et al., 2020). Em suma, o Selenium é frequentemente considerado o framework de fato para automação de navegadores (García et al., 2020).

O Selenium WebDriver utiliza os recursos nativos de cada navegador para oferecer suporte à automação. Por esse motivo, um teste usando a API do Selenium WebDriver requer um componente intermediário chamado *driver* no jargão do Selenium. Cada fornecedor de navegador fornece um driver específico. Por exemplo, o driver necessário para controlar o Chrome com o Selenium WebDriver é chamado chromedriver,[2]

[2] https://chromedriver.chromium.org/

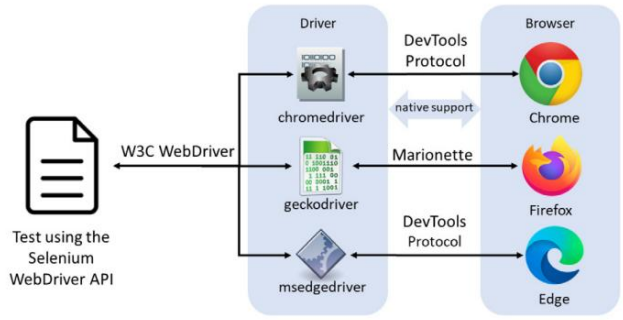


Fig. 1. Arquitetura do Selenium WebDriver.

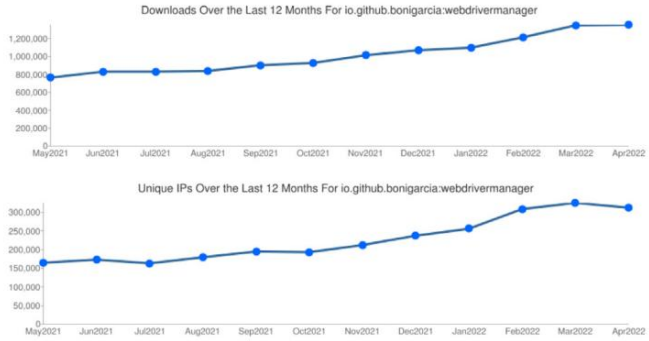


Fig. 2. Estatísticas de uso do WebDriverManager.

geckodriver3 para Firefox ou msedgedriver4 para Edge. Um driver é um componente que recebe mensagens de testes do Selenium WebDriver usando um protocolo padrão chamado W3C Web-Driver (Stewart e Burns, 2022), baseado em mensagens JSON via HTTP. O driver traduz cada mensagem do W3C WebDriver para um comando nativo que o navegador pode entender, como o protocolo DevTools em navegadores baseados em Chromium (como Chrome e Edge) ou o Marionette no Firefox. Portanto, como ilustrado na Figura 1, a arquitetura do Selenium WebDriver possui três camadas compostas pelo teste usando a API do Selenium WebDriver, o driver e o navegador.

WebDriverManager5 é uma biblioteca auxiliar Java de código aberto para o Selenium WebDriver. Seu principal recurso é o gerenciamento automatizado de drivers (por exemplo, chromedriver ou geckodriver) exigidos pelo Selenium WebDriver (García et al., 2021; Leotta et al., 2022). Ele também descobre navegadores instalados no sistema local, cria objetos WebDriver (como ChromeDriver, Firefox-Driver, etc.) ou executa navegadores em contêineres Docker perfeitamente.
O WebDriverManager foi lançado em 2015 e, desde então, tornou-se um utilitário auxiliar bastante conhecido para desenvolvedores do Selenium WebDriver. A Figura 2 mostra a evolução dos downloads mensais e dos IPs únicos do WebDriverManager de maio de 2021 a abril de 2022, de acordo com as Estatísticas Sonatype do Maven Central.6 Por exemplo, o WebDriverManager foi baixado 1.354.172 vezes, de 312.621 IPs únicos, em abril de 2022.

2.2. Exemplo motivacional

Conforme apresentado anteriormente, os logs são um dos pilares da observabilidade. Na área de testes ponta a ponta, o console do navegador

[3] https://github.com/mozilla/geckodriver https://
[4] developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/ https://
[5] bonigarcia.dev/webdrivermanager/ https://
[6] oss.sonatype.org/

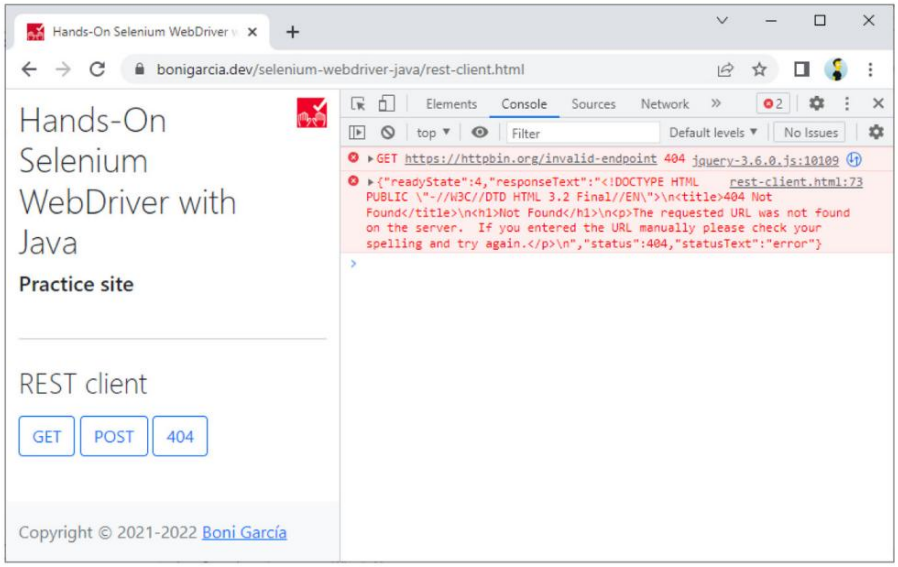


Fig. 3. Exemplo de rastreamento de erro de log no console do navegador.

Tabela 1

Categorias de logs *rastreáveis* (ou seja, tipos de logs que o BrowserWatcher é capaz de monitorar e coletar).

Categoria	Descrição	Exemplo
console-log	Mensagens regulares (ou seja, chamadas para console.log)	This a call to 'console.log' (index):10
console-aviso	Mensagens de aviso (ou seja, chamadas para console.warn)	▸This a call to 'console.warn' (index):12
erro de console	Mensagens de erro (ou seja, chamadas para console.error)	▸This a call to 'console.error' (index):13
informações do console	Mensagens informativas (ou seja, chamadas para console.info)	This a call to 'console.info' (index):11
console-outro	Outros métodos de console, como console.dir, console.tempo, console.timeEnd, console.table, ou console.count)	(index):17 (index) | Value a | 1 b | 2 c | 3
erro js	Rastros de erros de JavaScript (por exemplo, exceções não controladas)	▸Uncaught Error: This is a (forced) unhandled error unhandled_error.html:10 at unhandled_error.html:10:11
rejeição não tratada	Erro quando um JavaScript *Promessa* sem rejeição manipulador é rejeitado	▸Uncaught (in promise) unhandled_rejection.html:5 Unhandled rejection at 3:47:35 PM
violação de csp	Erro quando um conteúdo Política de Segurança (CSP) restrição é violada	Refused to load the stylesheet 'http://cdn.jsdelivr.net/npm/bootstrap@5.1.0/dist/css/bootstrap.min.css' because it violates the following Content Security Policy directive: "default-src 'self'". Note that 'style-src-elem' was not explicitly set, so 'default-src' is used as a fallback. csp_error.html:6
erro xhr	Respostas de erro de Solicitação XMLHttp	▸GET https://httpbin.org/invalid-endpoint 404 jquery-3.6.0.js:10109

pode conter dados relevantes para depurar e descobrir o subjacente causa de um teste reprovado. Considere o exemplo a seguir. Um exemplo webpage[7] desempenhando o papel de cliente REST faz solicitações a um serviço REST externo (García, 2022). Após cada solicitação, a resposta é exibida no corpo do documento. Conforme ilustrado na Figura 3, uma das solicitações (aquela acionada pelo botão 404) usa um ponto final inexistente. Como resultado, nenhuma resposta é exibida em a página e um rastreamento de erro aparece no console do navegador.

Um teste ponta a ponta típico automatizado com Selenium WebDriver interagiria com os botões da página para enviar solicitações, aguardando os resultados a serem exibidos no corpo. Em caso de erro, o teste normalmente falha por um tempo limite ou exceção de elemento inexistente (por exemplo, acionado por uma expressão localizadora incapaz de encontrar o elemento de interesse na página da web (Leotta et al., 2021, 2016b; Nass et al., 2023)). Ter acesso ao console do navegador seria fundamental para entendendo a causa, neste caso, uma solicitação ruim (404 Não Encontrado) para um ponto de extremidade remoto.

Este cenário mostra um exemplo de um tipo de erro específico (rotulado como *xhr-error* na Tabela 1) que pode ser depurado usando o console do navegador. Além disso, conforme explicado no próximo seções, outras categorias de log podem conter informações relevantes para análise de falhas. Outro exemplo típico é o não capturado exceções devido a violações de JavaScript ou CSP, para citar algumas. Em Além disso, pode ser interessante coletar avisos. Essas mensagens

[7] https://bonigarcia.dev/selenium-webdriver-java/rest-client.html

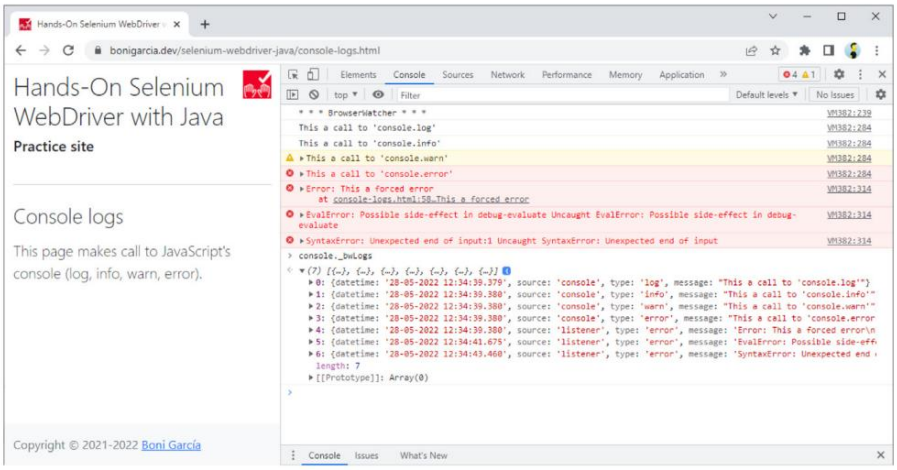


Fig. 4. Exemplo de coleta de logs do console através do BrowserWatcher.

são sintomas que não provocam um problema imediato no SUT, mas alertam para uma situação que pode levar a problemas futuros. Alguns exemplos desses avisos são descontinuações no código JavaScript ou CSS.

Portanto, coletar logs de testes automatizados com o Selenium WebDriver pode ser fundamental para a análise de falhas. Além disso, resolver avisos com antecedência é uma boa prática para evitar problemas futuros. Infelizmente, o protocolo W3C WebDriver não fornece nenhum mecanismo padrão para coleta de logs do console do navegador. Como alternativa, existem alguns mecanismos específicos do navegador para coleta de logs, como o módulo de log do Chrome DevTools Protocol (CDP) (Chrome Team, 2023) ou a implementação personalizada para coleta de logs (por meio do recurso do navegador chamado goog:loggingPrefs) implementado pelo driver para Chrome (ou seja, chromedriver) (Solntsev, 2019). No entanto, esses recursos não padrão não estão disponíveis em navegadores não baseados em Chromium, como o Firefox. A evolução do W3C WebDriver é chamada de protocolo WebDriver BiDirectional (BiDi) (Burns e Smith, 2022), que oferecerá suporte a funcionalidades e eventos relacionados ao registro. Infelizmente, o BiDi ainda está em fase de rascunho e sua adoção é escassa em navegadores e drivers. Como resultado, a coleta automatizada de logs é uma questão em aberto para melhorar a observabilidade de aplicações web nos principais navegadores.

3. Conjunto de ferramentas proposto

Este artigo contribui para o espaço de automação de navegadores ao propor o BrowserWatcher, uma ferramenta de código aberto que oferece diferentes recursos para observar aplicações web. Engenheiros web podem usar o BrowserWatcher como uma extensão comum instalada em um navegador ou por meio de navegadores instrumentados, controlados com o Se-lenium WebDriver e gerenciados pelo WebDriverManager.

3.1. Observador do Navegador

BrowserWatcher[8] é uma extensão de navegador de código aberto projetada explicitamente para melhorar a observabilidade em aplicativos da web. O BrowserWatcher foi implementado sobre a API WebExtensions, um conjunto de APIs JavaScript para vários navegadores para modificar e aprimorar os recursos do navegador (Mozilla MDN, 2022a; Chrome Team, 2022b). Esses recursos fornecidos pelo BrowserWatcher podem ser usados manualmente, por meio de sua Interface Gráfica do Usuário (GUI), e programaticamente por meio de uma API JavaScript. Esses recursos são os seguintes:

- *Coleta de logs do console.* O BrowserWatcher permite coletar diferentes tipos de logs do navegador. Os dados coletados do console e dos ouvintes são armazenados em uma propriedade personalizada do objeto do console chamada _bwLogs. A Figura 4 mostra uma captura de tela de uma página da web de exemplo[9] que grava vários logs no console e como o BrowserWatcher coleta essas informações.
- *Exibição de logs do console.* Quando este recurso está habilitado, o Browser-Watcher permite a exibição do log coletado na página web monitorada como notificações de diálogo em tempo real. A Figura 5 mostra uma captura de tela ilustrando isso. Este recurso pode ser útil para a inspeção manual dos logs gerados. Além disso, pode ser usado em conjunto com a gravação de guias (explicada a seguir) para monitorar os logs durante a navegação automatizada na web.
- *Gravação de abas do console.* O BrowserWatcher permite gravar o que está acontecendo em uma aba do navegador (por exemplo, quando um usuário está navegando em um site) e exportar a gravação como um arquivo de mídia usando o formato de vídeo WebM. Este recurso é baseado na API *tabCapture* (Chrome Team, 2022c).
- *Injeção de JavaScript e CSS.* O BrowserWatcher permite a injeção de código JavaScript personalizado, bibliotecas e folhas de estilo CSS. Esse recurso permite personalizar páginas da web com lógica e estilos externos do lado do cliente (por exemplo, para rastrear os movimentos do mouse, entre muitos outros recursos de ajuste).
- *Desabilitando o CSP.* A Política de Segurança de Conteúdo[10] (CSP) é o nome de um cabeçalho de resposta HTTP que visa melhorar a segurança de um site, instruindo o navegador sobre os recursos que ele pode carregar para aquela página, por exemplo, permitindo o carregamento de imagens de qualquer lugar, mas restringindo uma ação de formulário a um determinado endpoint. No entanto, os desenvolvedores podem querer ignorar os cabeçalhos CSP recebidos do servidor para fins de teste (por exemplo, para injetar código JavaScript personalizado, como explicado acima, mesmo quando o CSP estiver habilitado).

Em relação à coleta de logs, o BrowserWatcher foi projetado para coletar categorias de logs conhecidas, como logs de console ou rastros de erros de JavaScript. A implementação desse recurso é baseada em JavaScript, utilizando a técnica de *monkey patching* (também conhecida como *runtime hooking).* Essa técnica permite estender ou modificar o comportamento padrão de um software em tempo de execução sem alterar sua finalidade principal (Hunt, 2019). Em particular, o BrowserWatcher substitui o protótipo de console JavaScript para monitorar as chamadas de seus métodos (por exemplo, console.log). Além disso, o BrowserWatcher implementa diferentes

8 https://bonigarcia.dev/browserwatcher/

9 https://bonigarcia.dev/selenium-webdriver-java/console-logs.html

10 https://content-security-policy.com/

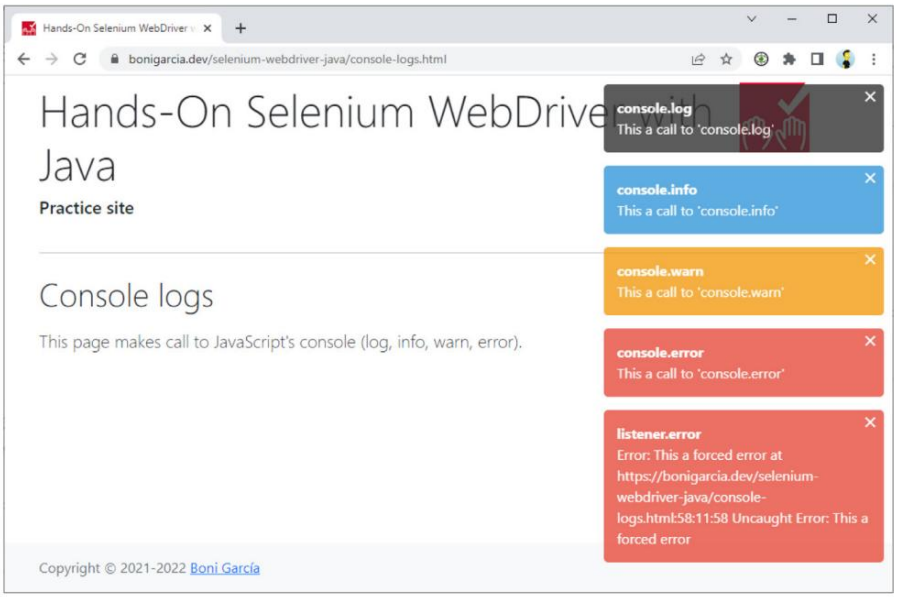


Fig. 5. Exemplo de exibição de log com o BrowserWatcher.

Tabela 2
Métodos da API WebDriverManager para monitoramento do navegador por meio do Browser-Watcher.

Método WebDriverManager	Descrição
assistir()	Habilitar coleta de logs
assistirEexibir()	Habilitar coleta e exibição de logs
obterLogs()	Leia os logs capturados
iniciarGravação()	Iniciar gravação de abas
pararGravação()	Parar gravação da aba
desabilitarCsp()	Desabilitar cabeçalhos CSP

Ouvintes de eventos JavaScript que gravam informações no navegador
console (por exemplo, rastreamentos de erros). Para implementar esse recurso, o
O manifesto da extensão BrowserWatcher usa permissões para permitir
a execução de scripts de conteúdo em linha do lado do cliente. Esses scripts
são configurados para serem executados antes de qualquer Document Object Model
O elemento (DOM) é construído ou qualquer outro script é executado. Isso
o comportamento garante que o protótipo do console corrigido esteja sempre
usado por toda a lógica JavaScript manipulada pelo SUT, mesmo para
chamadas antecipadas (por exemplo, para estratégias de carregamento de página ansioso,
como no Sele-nium WebDriver (García, 2022)) ou quando ocorre uma mudança de página
(ou seja, durante a navegação manual ou automatizada na web). Este início
O mecanismo de carregamento pode ser alcançado graças às WebExtensions
ciclo de vida (Chrome Team, 2021).

Em suma, e dadas as possibilidades técnicas do JavaScript,
O BrowserWatcher é capaz de reunir um conjunto de categorias de log.
A Tabela 1 resume essas categorias de log, chamadas de logs *rastreáveis*
de agora em diante neste artigo.

3.2. Integração com o WebDriverManager

A partir da versão 5.2.0, o WebDriverManager é capaz de monitorar
aplicações web através do BrowserWatcher. Para esse fim, o Web-DriverManager
instrumenta os navegadores a serem controlados com
Selenium WebDriver instalando o BrowserWatcher neles em
o início da sessão automatizada e antes do SUT ser
carregado. Então, o WebDriverManager fornece vários métodos de API
que permitem invocar os recursos fornecidos pelo BrowserWatcher.
A Tabela 2 resume esses métodos. Em seguida, a Fig. 6 mostra um método básico
Teste do Selenium WebDriver usando o recurso de coleta de log fornecido
pelo BrowserWatcher através do WebDriverManager.

4. Experimentação

Nós nos concentramos nos logs do console do navegador para o experimento
validação deste trabalho. Os logs são considerados um dos pilares
de observabilidade. Portanto, as informações nos logs do navegador são
essencial para desenvolvedores na análise de falhas (também conhecida como
solução de problemas) de testes automatizados de ponta a ponta. Portanto, este
estudo empírico visa avaliar a eficácia do BrowserWatcher
na coleta de logs gerados por diferentes navegadores. A partir disso
objetivo geral, as seguintes Perguntas de Pesquisa (PQs) são derivadas:

RQ1. Quais são as diferenças entre os logs coletados
automaticamente com BrowserWatcher e WebDriverManager e
os logs reais no console do navegador?

RQ2. Existem diferenças entre os logs coletados em
os principais navegadores da web (tanto automáticos quanto manuais)?

Para investigar essas RQs quantitativamente, temos que implementar
um conjunto de testes automatizados de ponta a ponta baseado no Selenium Web-
Driver e WebDriverManager mais BrowserWatcher para avaliar
diferentes SUTs usando diferentes navegadores. Além disso, precisamos
coletar os logs do navegador manualmente. Consideraremos manualmente
registros coletados como nossa verdade fundamental.

4.1. Design e configurações

O primeiro aspecto que precisamos decidir para projetar o experimento é quais
navegadores são os mais utilizados atualmente.
foco em navegadores de desktop, considerando os navegadores móveis uma
possível linha futura. Assim, podemos encontrar diferentes estatísticas online sobre
uso do navegador na área de trabalho. A Figura 7 mostra um gráfico de barras da área de trabalho
participação de mercado de navegadores em todo o mundo de abril de 2021 a abril de
2022, de acordo com StatCounters.11 Os 4 principais navegadores de desktop, de acordo com
a essas estatísticas, estão Chrome, Safari, Edge e Firefox. No entanto, não
podemos usar o Safari para a análise automatizada, pois
a API do Selenium WebDriver não permite a instalação do navegador
extensões programadas no Safari (portanto, o WebDriverManager não oferece
suporte). Concluindo, focamos no seguinte
navegadores de desktop no experimento: Chrome, Edge e Firefox.

11 https://gs.statcounter.com/browser-market-share/desktop/worldwide/#mensal-202104-202204-bar

```java
class GatherLogsChromeTest {

    WebDriverManager wdm = WebDriverManager.chromedriver().watch();
    WebDriver driver;

    @BeforeEach
    void setup() {
        driver = wdm.create();
    }

    @AfterEach
    void teardown() {
        driver.quit();
    }

    @Test
    void test() {
        driver.get(
                "https://bonigarcia.dev/selenium-webdriver-java/console-logs.html");
        List<Map<String, Object>> logMessages = wdm.getLogs();
        assertThat(logMessages).hasSize(5);
    }

}
```

Fig. 6. Exemplo de teste do Selenium WebDriver usando os recursos de monitoramento fornecidos pelo WebDriverManager mais BrowserWatcher.

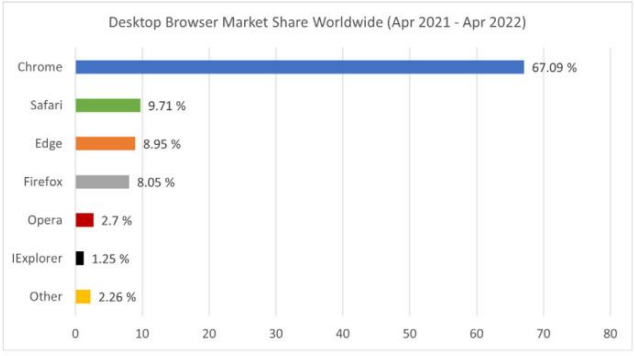


Fig. 7. Participação de mercado de navegadores de desktop em todo o mundo (abril de 2021–abril de 2022).
Fonte: StatCounters.

Outro aspecto que precisamos selecionar é o SUT (ou seja, o site de destino) para a experimentação. Nosso objetivo é coletar logs heterogêneos. Assim, usamos sites populares existentes em todo o mundo como SUT. Para isso, contamos com as informações publicadas pelo Alexa Top Sites, um serviço web que fornece listas de sites ordenados pelo Alexa Traffic Rank (Amazon, 2022). A lista dos 50 principais sites, de acordo com essa fonte em 19 de abril de 2022, é apresentada na Tabela 3. Usamos um identificador exclusivo para cada site (S01 a S50) para nos referirmos a cada site nas seções subsequentes.

Por fim, implementamos um caso de teste ponta a ponta automatizado do Selenium WebDriver usando WebDriverManager e BrowserWatcher. Este teste é parametrizado usando o produto cartesiano dos navegadores de destino (Chrome, Edge e Firefox) e os SUTs (top 50 sites) como argumentos. Em outras palavras, o teste é executado 150 vezes, uma vez por navegador e site. Para cada execução, o teste realiza as seguintes etapas:

1. **Injeção do BrowserWatcher.** O teste invoca os métodos watch() e create() fornecidos pelo WebDriverManager.
   Esses métodos permitem a criação de uma sessão do Selenium WebDriver na qual o BrowserWatcher é injetado antes do carregamento do DOM. Dessa forma, a coleta de logs do navegador começa no início da sessão automatizada.
2. **Carregamento do site.** O teste invoca o método get() do Selenium WebDriver para carregar a página raiz dos diferentes SUTs.
   A estratégia de carregamento de página padrão usada pelo Selenium Web-Driver (chamada *normal)* espera até que a página inteira seja carregada. Isso acontece quando a propriedade *readyState* do DOM está *completa,* o que significa que o documento e todos os sub-recursos (como arquivos JavaScript ou folhas de estilo CSS) terminaram de carregar (García, 2022).
3. **Coleta de logs.** O teste invoca o método getLogs() do WebDriverManager para coletar o log do navegador por execução.
4. **Gravação de arquivo.** O teste cria um arquivo de texto a cada execução contendo os resultados (por exemplo, 01_google.com_CHROME.txt).

Os 150 arquivos de texto gerados a partir dos testes automatizados são o primeiro conjunto de evidências usado para análise de dados e resposta às RQs. Além dessas informações, precisamos coletar manualmente o console do navegador para as mesmas fontes de entrada. Em outras palavras, precisamos visitar os 50 sites de destino usando Chrome, Edge e Firefox e coletar os logs do console manualmente. Como resultado, geramos manualmente outros 150 arquivos de texto.

Para replicabilidade, disponibilizamos o código-fonte e os recursos gerados para este experimento no GitHub.13 Este repositório contém o teste do Selenium WebDriver para a coleta automática de logs. Além disso, inclui alguns scripts de shell (para Windows e Bash) projetados para economizar tempo na etapa de coleta manual, carregando os 50 sites de destino nos diferentes navegadores.

O experimento foi realizado em 3 de maio de 2022. Os processos de coleta de logs, automatizados e manuais, foram realizados em Madri (Espanha) em um PC Windows com CPU AMD Ryzen 7 de 64 bits e 16 GB de RAM. Os navegadores utilizados no experimento foram Chrome 101.0.4951.67, Edge 101.0.1210.53 e Firefox 100.0.1.
Os arquivos de texto resultantes contendo os logs coletados (automatizados e manuais) são publicados no repositório GitHub mencionado acima e são analisados na subseção a seguir.

4.2. Resultados

Para responder às RQs, comparamos cada arquivo de texto gerado nos testes automatizados (por exemplo, navegador Chrome e site S01) com sua saída correspondente do processo manual.

Esta comparação foi um processo manual, uma vez que os registros coletados manual e automaticamente apresentam formatos diferentes, mesmo para as mesmas entradas de registro. A Figura 8 ilustra essas diferenças com

12 https://web.archive.org/web/20220319013139/https://www.alexa.com/topsites

13 https://github.com/bonigarcia/webdrivermanager-log-gathering

Tabela 3

Os 50 melhores sites de acordo com o Alexa Top Sites em 19 de abril de 2022.

Eu ia	Site Site	Eu ia		Eu ia	Site
S01	google.com	S02	youtube.com	S03	baidu.com
S04	facebook.com	S05	instagram.com	S06	qq.com
S07	bilibili.com	S08	yahoo.com	S09	wikipedia.org
S10	amazon.com	S11	twitter.com	S12	zhihu.com
S13	linkedin.com	S14	reddit.com	S15	whatsapp.com
S16	taobao.com	S17	live.com	S18	chaturbate.com
S19	bing.com	S20	sina.com.cn	S21	sohu.com
S22	csdn.net	S23	tmall.com	S24	weibo.com
S25	zoom.us	S26	microsoft.com	S27	jd.com
S28	github.com	S29	google.com.hk	S30	netflix.com
S31	163.com	S32	office.com	S33	vk.com
S34	canva.com	S35	microsoftonline.com	S36	1688.com
S37	aparat.com	S38	naver.com	S39	twitch.tv
S40	pornhub.com	S41	stackoverflow.com	S42	amazon.in
S43	yahoo.co.jp	S44	xvideos.com	S45	myshopify.com
S46	paypal.com	S47	tiktok.com	S48	aliexpress.com
S49	douban.com	S50	apple.com		

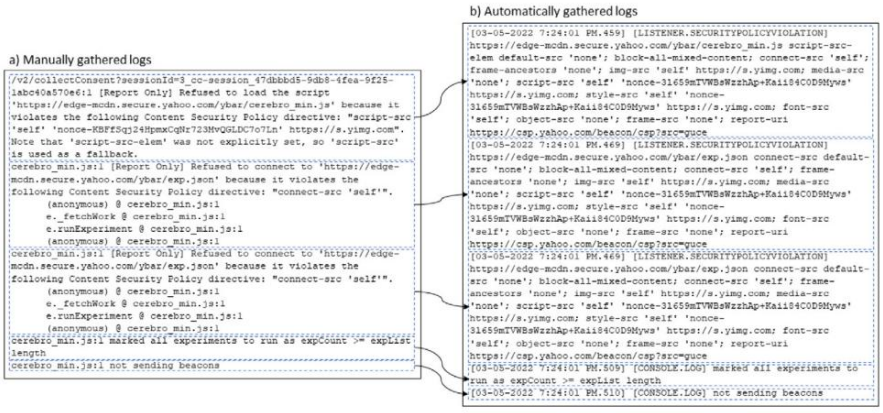


Fig. 8. Exemplo de comparação de logs (manual vs. automático) para https://yahoo.com.

um exemplo concreto: os logs coletados para S08, tanto manualmente e automaticamente. Como mostrado na imagem, o conteúdo parece completamente diferente. No entanto, analisando seu conteúdo, pode-se verificado se corresponde às mesmas entradas de log: três CSP violações e duas chamadas console.log . Como essa comparação pode ser uma tarefa propensa a erros, os resultados foram verificados duas vezes pelo quatro autores deste artigo antes de uma análise mais aprofundada.

Contamos as entradas de log correspondentes em cada arquivo, agrupando cada entrada por categoria. Ao final deste processo, encontramos a primeira constatação relevante desta investigação. Além do registro categorias coletadas pelo BrowserWatcher, apresentadas na Tabela 1 e chamado *rastreável* neste artigo, encontramos outros tipos de logs. De a partir de agora, nos referimos a essas categorias usando o termo *não rastreável* em contraste com os registros rastreáveis introduzidos na Seção 3.1. Em por um lado, logs rastreáveis podem ser coletados usando JavaScript ouvintes e objetos com remendo de macaco. Por outro lado, para o até onde sabemos, registros não rastreáveis não podem ser coletados usando essas técnicas e, portanto, estão fora do escopo de uma extensão de navegador como o BrowserWatcher.

Além disso, descobrimos que alguns registros não rastreáveis apareceram em todos os navegadores estudados (Chrome, Edge e Firefox), e alguns deles eram específicos do navegador. Dessa forma, agrupe essas categorias de log não rastreáveis da seguinte forma:

- *Geral:* categorias de log encontradas no Chrome, Edge e Firefox (Tabela 4).
- *Específico do Chromium:* categorias de log encontradas no Chrome e Borda (Tabela 5).
- *Específico do Chrome:* categorias de log encontradas apenas no Chrome (Tabela 6).
- *Específico do Edge:* categorias de log encontradas apenas no Edge (Tabela 7).
- *Específico do Firefox:* categorias de log encontradas apenas no Firefox (Tabela 8).

A Tabela 9 apresenta os resultados numéricos do log coletado manual e automaticamente, fornecendo detalhes sobre as categorias de log não rastreáveis. Usamos esses resultados para continuar com o análise de dados e responda às perguntas frequentes nas subseções a seguir.

4.2.1. RQ1: Logs coletados automaticamente vs. manualmente

A categoria de logs rastreáveis é o único tipo de logs presente em os processos manuais e automatizados. Portanto, podemos diretamente compare os resultados em cada navegador (Chrome, Edge e Firefox) nos processos automatizados e manuais. As imagens a seguir (Fig. 9 para Chrome, Fig. 10 para Edge e Fig. 11 para Firefox) mostrar uma comparação das entradas de log correspondentes no rastreável categoria, tanto na análise automatizada quanto na manual.

Esses resultados revelam que o BrowserWatcher reúne todos os rastreáveis logs. No entanto, encontramos diferenças relevantes quando consideramos os logs que não podem ser coletados automaticamente com o Browser-Watcher (ou seja, os logs não rastreáveis). Portanto, considerando o número total de logs (ou seja, rastreáveis e não rastreáveis, conforme mostrado na Tabela 9), 76 logs de entrada foram capturados no Chrome de 220 (ou seja, 36% de eficácia), 72 registros de entrada foram capturados no Edge de 239 (ou seja, 30,12%) e 145 de 993 no Firefox (ou seja, 14,60%).

As imagens a seguir mostram esses resultados usando gráficos de barras. Em primeiro lugar, a Fig. 12 mostra os registros não rastreáveis que estão presentes em todos os navegadores alvo deste estudo (ou seja, Chrome, Edge e Firefox). Em segundo lugar, a Fig. 13 mostra os logs não rastreáveis nos navegadores baseados em Chromium (ou seja, Chrome e Edge). Por fim, a Fig. 14 mostra

Tabela 4

Categorias *gerais* de logs (ou seja, tipos de logs não rastreáveis que foram encontrados no Chrome, Edge e Firefox).

Categoria	Descrição	Exemplo
erro de carregamento	Erros HTTP (por exemplo, 451 Indisponível) quando carregando recursos	▸GET https://id.rlcdn.com/472486.gif?gtmcb=1347499948 451 472486.gif:1
conteúdo misto	Carga de recursos HTTP através de uma sessão HTTPS	Mixed Content: The page at 'https://www.sohu.com/' was loaded over HTTPS, but requested an insecure element 'http://statics.itc.cn/sohu-homepage/oldentrancetwo.png'. This request was automatically upgraded to HTTPS, For more information see https://blog.chromium.org/2019/10/no-more-mixed-messages-about-https.html :1
js-obsoleto	Uso de JavaScript obsoleto elementos	▸Carousel is deprecated, please use either MultiSlideCarousel or SingleSlideCarousel instead. social?apiVersion=1.0:1
xhr-aviso	Avisos ao invocar Solicitação XMLHttp Objetos ResponseType	▸The provided value 'html' is not a valid enum value of type XMLHttpRequestResponseType.
CSS	Aviso ao aplicar CSS estilos	The keyword 'slider-vertical' is not specified for the 'appearance' property. It will be removed in the future. www.microsoft.com/:215

Tabela 5

Categorias de log *do Chromium* (ou seja, tipos de logs não rastreáveis que foram encontrados no Chrome e no Edge).

Categoria	Descrição	Exemplo
ferramentas de desenvolvimento	Problemas relacionados a DevTools (Equipe Chrome, 2022a)	DevTools failed to load source map: Could not load content for https://www.youtube.com/s/desktop/8498231a/jsbin/custom-elements-es5-adapte…t/k8-opt/bin/third_party/javascript/custom_elements/fast-shim.js.sourcemap: HTTP error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
estrutura em X	Problemas relacionados com o Opções de quadro X HTTP cabeçalho de resposta	Refused to display 'https://accounts.google.com/' in a frame because it set 'X-Frame-Options' to 'deny'. chromewebdata/:1
corb	Bloqueio de leitura entre origens (CORB) respostas bloqueadas	▸Cross-Origin Read Blocking (CORB) blocked cross-origin response https://web.vortex.data.microsoft.com/collect/v1 with MIME type application/json. See https://www.chromestatus.com/feature/5629709824032768 for more details. meCore.min.js:1
erro de cabeçalho	Erro com Política de Permissões cabeçalhos	Error with Permissions-Policy header: Origin trial controlled feature not enabled: 'interest-cohort'.
manifesto	Problemas com o webapp manifesto (Cáceres et al., (2022)	6 Manifest: property 'url' ignored, should be within scope of the manifest.
origem cruzada	Problemas com Cross-Origin Compartilhamento de Recursos (CORS) (Hossain, 2014)	▸A parser-blocking, cross site (i.e. different eTLD+1) script, https://atanx.alicdn.com/t/tanxssp.js, is invoked via document.write. The network request for this script MAY be blocked by the browser in this or a future page load due to poor network connectivity. If blocked in this page load, it will be confirmed in a subsequent console message. See https://www.chromestatus.com/feature/5718547946799104 for more details. index.js:13

Tabela 6

Categorias de log *do Chrome* (ou seja, tipos de logs não rastreáveis que foram encontrados apenas no Chrome).

Categoria	Descrição	Exemplo
notificações-permissão	Parar temporariamente uma origem de solicitar um permissão	▸Notifications permission has been blocked as the user has ignored the permission prompt several times. This can be reset in Page Info which can be accessed by clicking the lock icon next to the URL. See https://www.chromestatus.com/feature/6443143280984064 for more information.
armazenamento local de sessão	Aviso devido à sessão ou o armazenamento local está desabilitado	▸Ada Embed - SessionStorage is disabled. This is likely because your browser is blocking third-party cookies. index.js:1

os logs não rastreáveis presentes apenas em um determinado navegador (Chrome, Edge ou Firefox).

À luz destes resultados, a resposta à RQ1 é dupla, dependendo do tipo de log. Por um lado, a diferença entre os logs rastreáveis coletados manualmente e automaticamente é apenas sintático. Este fato significa que o conteúdo de rastreáveis os logs coletados automaticamente são sintaticamente diferentes dos verdade básica (ou seja, os registros coletados manualmente), mas seu conteúdo é semanticamente equivalente (como ilustrado, por exemplo, na Fig. 8). Em outras palavras, ambos os conjuntos de logs referem-se a entradas de log idênticas.

Tabela 7

Categorias de logs *do Edge* (ou seja, tipos de logs não rastreáveis que foram encontrados somente no Edge).

Categoria	Descrição	Exemplo
prevenção de rastreamento	Avisos devido ao rastreamento prevenção, que é da Edge recurso que restringe rastreadores para acessar armazenamento baseado em navegador e rede	⚠ Tracking Prevention blocked access to storage for https://id.rlcdn.com/472486.gif?gtmcb=1163105743. www.reddit.com/:1
intervenção	Avisos devido à preguiça imagens carregando elas são provavelmente será visto	[Intervention] Images loaded lazily and replaced with placeholders. Load events are deferred. See https://go.microsoft.com/fwlink/?linkid=2048113 nmain.f2c903dd.js?o=www:1

Tabela 8

Categorias de log *do Firefox* (ou seja, tipos de logs não rastreáveis que foram encontrados apenas no Firefox).

Categoria	Descrição	Exemplo
biscoitos	Avisos devido a cookies	⚠ Cookie "AEC" has been rejected because it is already expired. www.google.com
csp-aviso	Avisos devido ao CSP	⚠ Content Security Policy: Ignoring "'unsafe-inline'" within script-src or style-src: nonce-source or hash-source specified
política de referência	Uso de políticas menos restritas	ⓘ Referrer Policy: Ignoring the less restricted referrer policy "unsafe-url" for the cross-site request: *https://dss0.bdstatic.com /5aV1bjqh_Q23odCf/static/superman /img/topnav/newwenku-d8c9b7b0fb.png* newwenku-d8c9b7b0fb.png
ignorado-sem suporte	Problemas relacionados ao observável características de desempenho (Peña Moreno, 2022)	⚠ Ignoring unsupported entryTypes: longtask. …s893eu15f8lrcy12rq:1:112685
desinfetante	Fonte para download rejeitada por desinfetante	ⓔ downloadable font: rejected by sanitizer (font-family: "redesignFont2020" style:normal weight:400 stretch:100 src index:0) source: *https://www.redditstatic.com/desktop2x/fonts /redesignIcon2020 /redesignFont2020.04df26efdb1943ec5c61ee9c3e3ea134.eot*
mídia	Tipos MIME não suportados	⚠ Specified "type" attribute of "video/mp4; codecs=hevc,mp4a.40.2" is not supported. Load of media resource *https://github.githubassets.com/images /modules/site/home/globe-900.hevc.mp4* failed. Trying to load from next <source> element. github.com
modo peculiaridades	Página no Modo Quirks (ou seja, uma compatibilidade com versões anteriores mecanismo usado em antigos navegadores)	⚠ This page is in Quirks Mode. Page layout may be impacted. For Standards Mode use "<!DOCTYPE html>". [Learn More] authorize
tipo de conteúdo x	Problemas relacionados com o X-Content-Type-Opções Cabeçalho de resposta HTTP	⚠ The script from "*https://interface.sina.cn /dfz/outside/ipdx/langshou_feed.d.json*" was loaded even though its MIME type ("text/html") is not a valid JavaScript MIME type. [Learn More] www.sina.com.cn
http2	Problemas relacionados ao HTTP/2	▶ GET https://www.yahoo.co.jp/ [HTTP/2 403 Forbidden 899ms]
conjunto de caracteres	Problemas relacionados com o codificação de caracteres do documento	⚠ The character encoding of a framed document was not declared. The document may appear different if viewed without the document framing it. conditional

Tabela 9

Número de logs coletados manualmente e automaticamente no Chrome, Edge e Firefox.

	Automatizado	Manual						
	Rastreável	Rastreável	Em geral	Específico para cromo	Específico do Chrome	Específico de borda	Específico do Firefox	Total
Cromo	76	76	35	103	6	N/A	N / D	220
Borda	72	72	30	59	N / D	76	N/A	239
Raposa de fogo	145	145	68	N / D	N / D	N / D	780	993

Por outro lado, há diferenças relevantes quando se consideram todas as categorias de logs possíveis (ou seja, os logs não rastreáveis). No geral, concluímos que o conjunto de ferramentas composto por WebDriver-Manager e BrowserWatcher se comporta corretamente para o ambiente gerenciado (ou seja, categorias de log rastreáveis). No entanto, apresenta limitações em outras categorias de log (chamadas de não rastreáveis neste trabalho) como eles não podem ser coletados por meio de JavaScript.

4.2.2. RQ2: Diferenças nos logs de diferentes navegadores

O segundo RQ investiga as diferenças entre os logs gerados em diferentes navegadores. Dividimos a análise em duas partes para responder a esta pergunta. Primeiro, focamos em logs rastreáveis e então analisamos aqueles não rastreáveis.

Encontramos diferenças relevantes entre os logs rastreáveis gerados em diferentes navegadores. A Figura 15 ilustra essas diferenças. mostrando que o número de entradas de log é diferente para todas as categorias. Curiosamente, há uma distinção significativa: o número de violações do CSP. Para entender essa discrepância, verificamos o capturas de log individuais. Por exemplo, a Figura 16 mostra os logs disponíveis ao carregar o S25 no Chrome, Edge e Firefox. O navegador o console no Chrome contém uma exceção JavaScript não controlada e avisos relacionados ao armazenamento local e de sessão. Então, o Edge relata apenas a mesma exceção do JavaScript. Finalmente, o Firefox O console é totalmente diferente. Em relação ao CSP, ele contém vários advertências e uma violação. Embora a implementação do CSP em os navegadores em estudo são equivalentes (Deveria e Schoors,

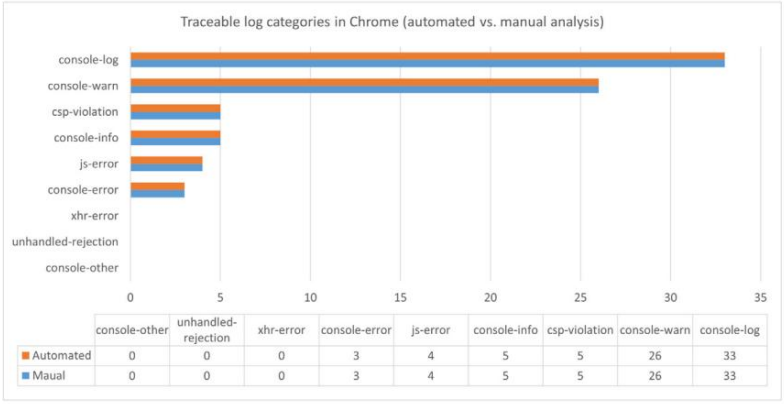


Fig. 9. Categorias de log rastreáveis no Chrome (análise automatizada vs. manual).

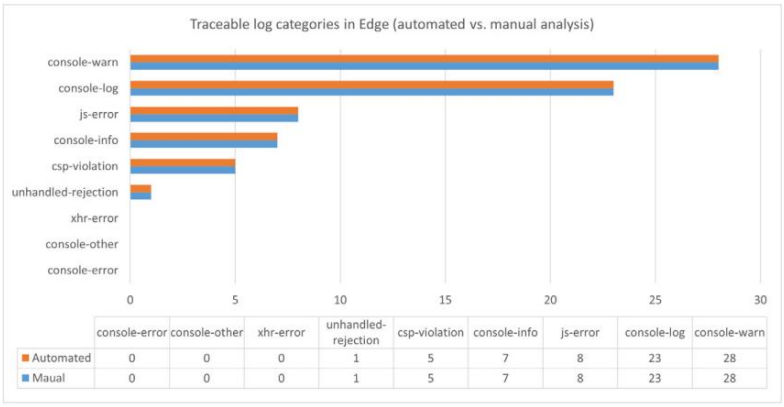


Fig. 10. Categorias de log rastreáveis no Edge (análise automatizada vs. manual).

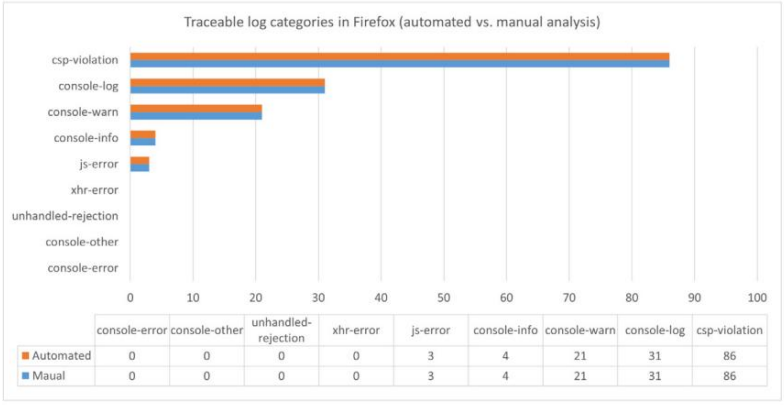


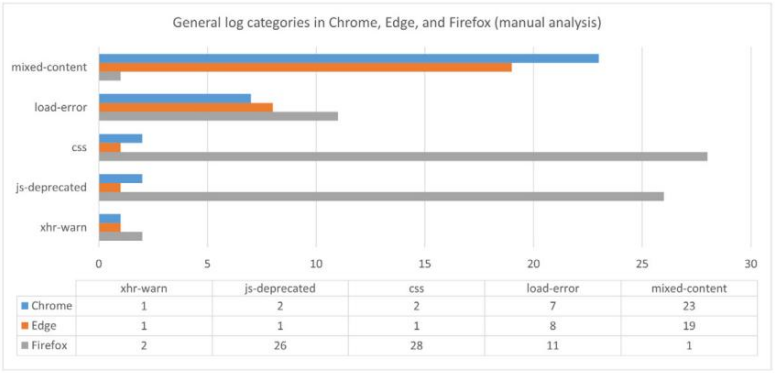
Fig. 11. Categorias de log rastreáveis no Firefox (análise automatizada vs. manual).

2022), o Firefox é mais detalhado (ou seja, ele exibe avisos de CSP que o Chrome e o Edge não exibem).

Em relação aos logs gerais não rastreáveis (Fig. 12), existem várias diferenças relevantes entre os navegadores. Primeiro, o Chrome e o Edge exibem muitas entradas de log relacionadas a *conteúdo misto* em comparação com o Firefox. Por outro lado, o Firefox exibe muito mais logs relacionados à descontinuação de *CSS* e *JS* em comparação com o Chrome e o Edge. Para ilustrar essas diferenças com exemplos, a Fig. 17 mostra um exemplo do *conteúdo misto* exibido ao carregar o S21.

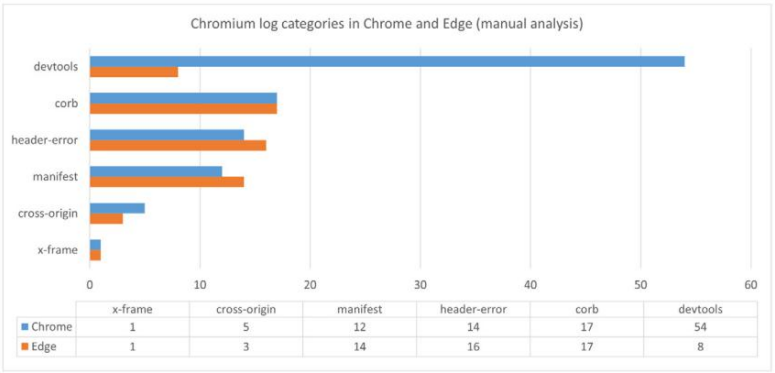
A Figura 18 mostra as diferenças relacionadas ao *CSS* no S17. Por fim, a Figura 19 mostra que apenas o Firefox apresenta problemas relacionados à *depreciação do JS* ao carregar o S48.

Em relação aos logs não rastreáveis do Chromium (Fig. 13), o único desvio significativo aparece na categoria *devtools* . Essa diferença é claramente ilustrada na Fig. 20, na qual o Chrome exibe mais entradas relacionadas a essa categoria ao navegar no S12.



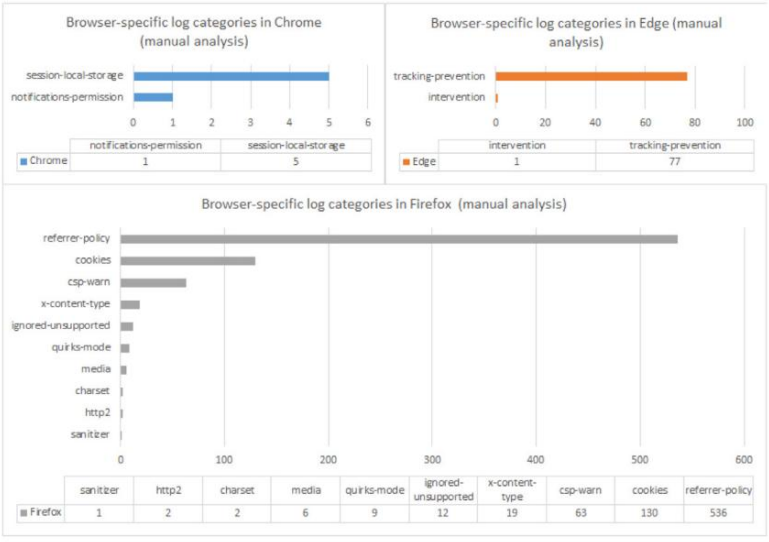
	xhr-warn	js-deprecated	css	load-error	mixed-content
Chrome	1	2	2	7	23
Edge	1	1	1	8	19
Firefox	2	26	28	11	1

Fig. 12. Categorias gerais de log no Chrome, Edge e Firefox (análise manual).



	x-frame	cross-origin	manifest	header-error	corb	devtools
Chrome	1	5	12	14	17	54
Edge	1	3	14	16	17	8

Fig. 13. Categorias de log do Chromium no Chrome e no Edge (análise manual).



	notifications-permission	session-local-storage
Chrome	1	5

	intervention	tracking-prevention
Edge	1	77

	sanitizer	http2	charset	media	quirks-mode	ignored-unsupported	x-content-type	csp-warn	cookies	referrer-policy
Firefox	1	2	2	6	9	12	19	63	130	536

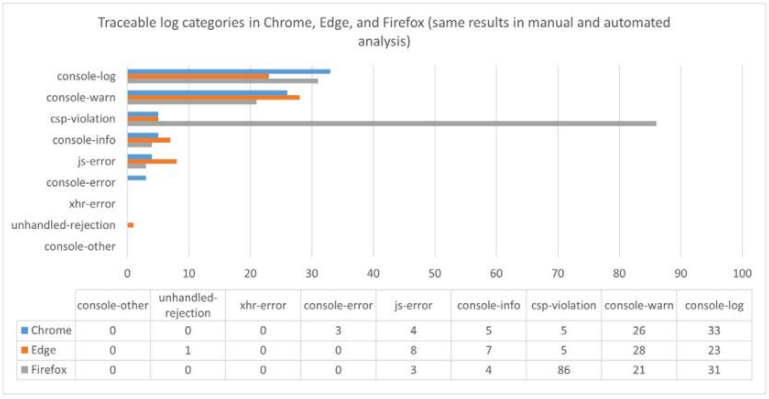
Fig. 14. Categorias de log específicas do navegador no Chrome, Edge e Firefox (análise manual).

Por fim, para logs específicos de navegadores não rastreáveis (Fig. 14), as categorias mais numerosas aparecem no Firefox. Primeiro, *referrer-policy* (um exemplo pode ser visto na Fig. 19). Segundo, *cookies* (por exemplo, como mostrado também na Fig. 19). E, finalmente, *csp-warn* (por exemplo, mostrado na Fig. 16).

Concluindo, e respondendo à Pergunta 2, concluímos que existem diferenças relevantes nos logs exibidos pelos diferentes navegadores. Primeiro, nos logs coletados automaticamente (ou seja, os logs rastreáveis), a principal diferença ocorre nas violações de CSP reportadas pelo Firefox, que são muito mais numerosas do que os erros de CSP reportados pelo Chrome e Edge. Segundo, em relação aos logs coletados manualmente (ou seja, os logs não rastreáveis), há várias categorias de logs com resultados heterogêneos nos navegadores de destino deste estudo, como conteúdos mistos (mais numerosos no Chrome e no Edge do que no Firefox), avisos de descontinuação de CSS e JavaScript (relatados mais vezes no Firefox do que no Chrome e no Edge) ou mensagens do DevTools (relatadas mais no Chrome do que no Edge).

4.3. Ameaças à qualidade da pesquisa

Ao avaliar a qualidade da pesquisa, a confiabilidade e a validade a qualidade dos resultados precisa ser considerada.



	console-other	unhandled-rejection	xhr-error	console-error	js-error	console-info	csp-violation	console-warn	console-log
Chrome	0	0	0	3	4	5	5	26	33
Edge	0	1	0	0	8	7	5	28	23
Firefox	0	0	0	0	3	4	86	21	31

Fig. 15. Categorias de log rastreáveis no Chrome, Edge e Firefox (mesmos resultados na análise manual e automatizada).

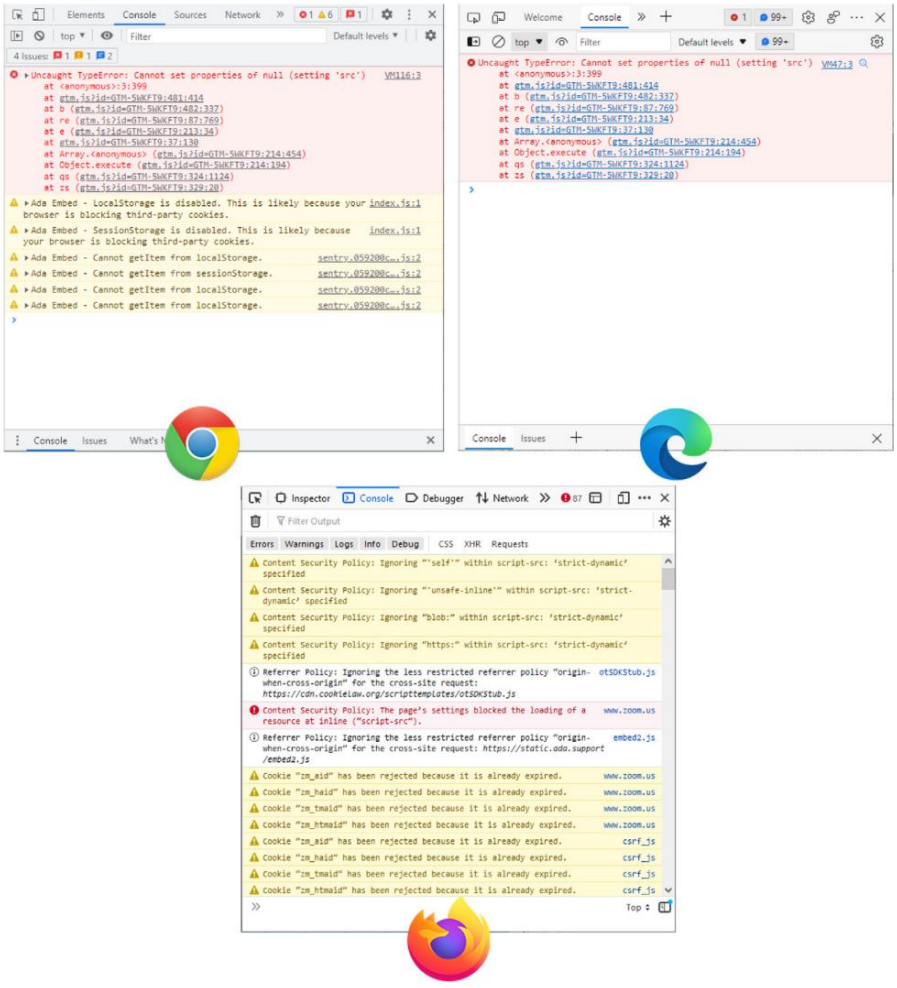


Fig. 16. Logs no Chrome, Edge e Firefox ao carregar https://zoom.us.

Para minimizar a ameaça à confiabilidade, nosso ambiente experimental é consistente e nossos resultados são reproduzíveis. Primeiramente, para garantir a consistência do nosso experimento, reduzimos a influência de fatores externos que poderiam gerar variação nos resultados. Por exemplo, o mesmo teste foi executado para cada site e navegador. No processo automatizado, o teste terminou assim que a página foi carregada. Isso é feito automaticamente com o Selenium WebDriver, ouvindo o estado *completo* de prontidão do Document Object Model (DOM) (García, 2022). Assim, no processo manual, o teste terminou quando o navegador indicou que a página foi carregada, verificando o spinner exibido no ícone do aplicativo durante o carregamento da página. É verdade que não podemos garantir que os resultados serão os mesmos ao longo do tempo, pois os sites podem mudar. Além disso, sites modernos podem usar estratégias de carregamento lento para atualizar componentes da web após o carregamento do DOM. No entanto, para reduzir essa ameaça, realizamos testes manuais e automatizados no mesmo dia, com uma diferença de cerca de uma hora entre os testes automatizados e manuais.

Além disso, para minimizar a ameaça à consistência do observador, os testes manuais e automatizados usaram navegadores *em sandbox* (ou seja, usando um novo perfil e configuração padrão por teste).

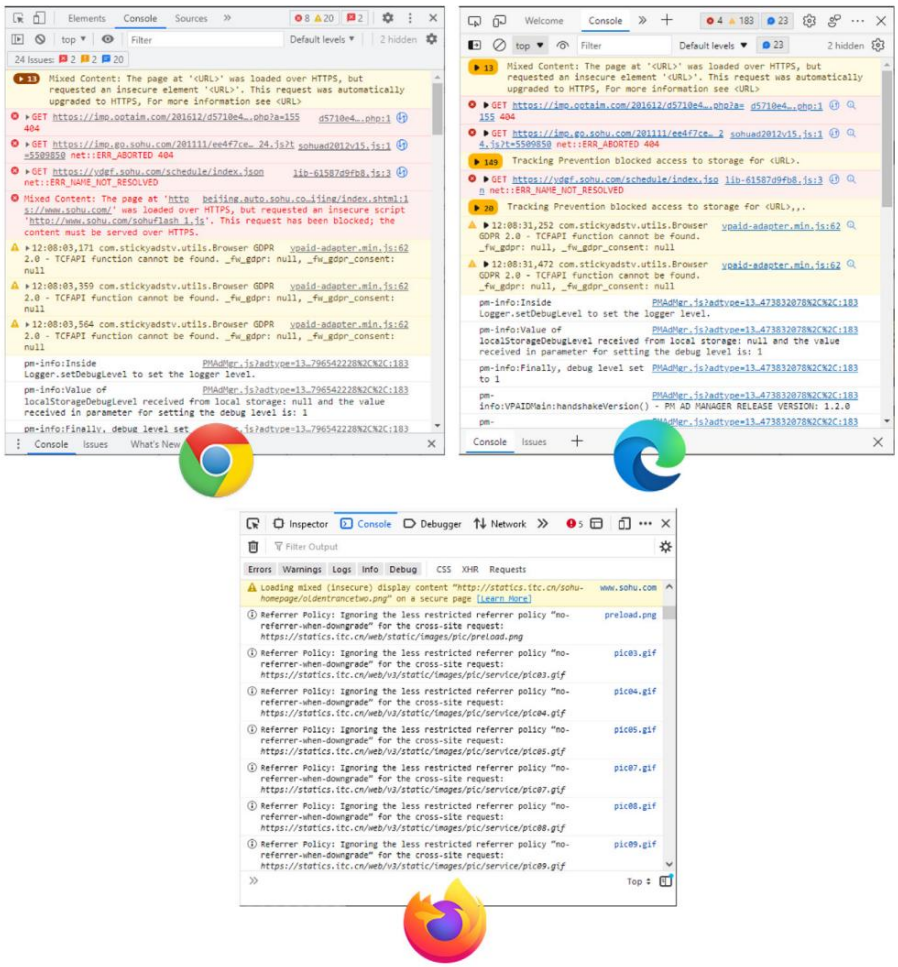


Fig. 17. Logs no Chrome, Edge e Firefox ao carregar https://sohu.com.

Em segundo lugar, para dar suporte à reprodutibilidade do nosso experimento, todos os testes, ativos e resultados estão disponíveis publicamente no GitHub. Além disso, ao ser integrado aos processos do GitHub Actions, o experimento é reproduzido automaticamente em uma compilação de Integração Contínua (CI).14

Tomamos diferentes medidas para lidar com as outras ameaças à validade. Primeiramente, realizamos uma amostragem adequada, trabalhando com os 50 principais sites do mundo e os principais navegadores que permitem testes automatizados, representando mais de 84% de participação de mercado. Reconhecemos que os resultados obtidos não são generalizáveis para outros navegadores (por exemplo, Safari), pois isso requer mais trabalho que planejamos abordar assim que esses navegadores adotarem o futuro protocolo WebDriver BiDi.

Em segundo lugar, para abordar a validade do construto, e devido à falta de estudos comparáveis, coletamos manualmente uma verdade fundamental para comparar nossos resultados automatizados. Esse fato, por si só, é uma contribuição relevante que outros pesquisadores podem aproveitar, visto que esse conjunto de dados também está disponível no GitHub. Além disso, diferentes ações foram tomadas para evitar viés na comparação dos logs coletados manual e automaticamente. Primeiramente, as categorias de logs foram discutidas por todos os membros da equipe até que um entendimento comum de todos os aspectos abordados fosse alcançado. Por fim, a classificação e a comparação dos logs foram verificadas novamente pelos quatro autores.

14 https://github.com/bonigarcia/webdrivermanager-log-gathering/actions

5. Discussão

Este trabalho é um esforço para aprimorar os recursos de observabilidade de aplicações web por meio de uma extensão cross-browser chamada BrowserWatcher. Esta extensão foi desenvolvida com base na API WebExtensions. Dessa forma, qualquer navegador moderno pode se beneficiar de seus recursos de observabilidade, como coleta de logs ou gravação de abas.

Dentre todos os recursos oferecidos pelo BrowserWatcher, e devido à sua importância na análise de falhas, focamos neste artigo na capacidade de coleta de logs do navegador. A coleta de logs não faz parte do mecanismo padrão para automação de navegadores (ou seja, o protocolo W3C WebDriver). Portanto, não há um mecanismo cross-browser amplamente adotado até o momento para realizar essa tarefa em testes automatizados.

Para superar esse problema, a partir da versão 5.2.0, o WebDriver-Manager permite a instrumentação de navegadores web com o BrowserWatcher, fornecendo uma API simples para coletar logs de testes automatizados do Selenium WebDriver. O principal benefício dessa abordagem é a implementação perfeita da coleta de logs entre navegadores (inclusive no Firefox, o que não era possível até o momento). Graças à validação experimental, verificamos que as categorias de log conhecidas (como rastreamentos de console e de erro) são coletadas perfeitamente com este conjunto de ferramentas. No entanto, a abordagem proposta apresenta uma limitação significativa. Como se baseia em APIs JavaScript, existem várias categorias de log (por exemplo, relacionadas a DevTools, conteúdo misto ou CSS).

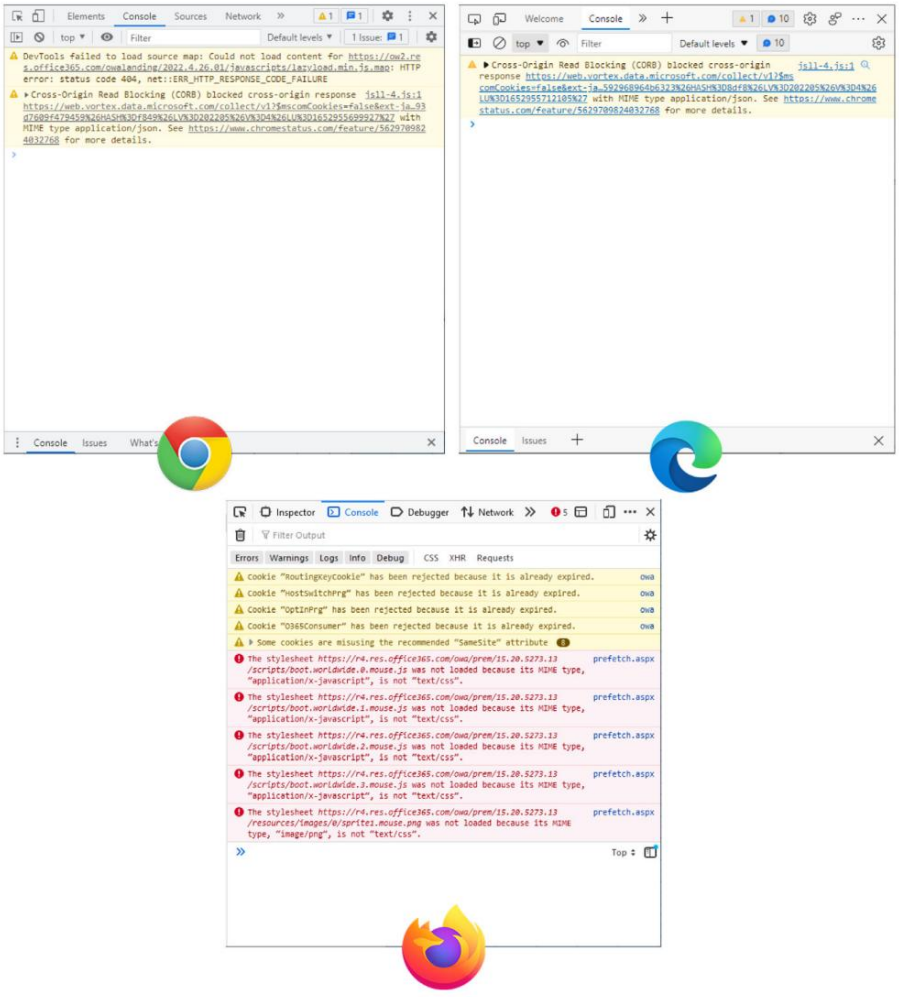


Fig. 18. Logs no Chrome, Edge e Firefox ao carregar https://live.com.

entre outros) que permanecem sem coleta. Acreditamos que esses logs, chamados de não rastreáveis neste artigo, devem ser considerados por profissionais (como pesquisadores e desenvolvedores) em trabalhos futuros relacionados a logs de navegadores do lado do cliente.

Utilizamos o conjunto de ferramentas apresentado (ou seja, WebDriverManager e BrowserWatcher) para navegar automaticamente em sites populares com diferentes navegadores. As diferenças encontradas nos logs coletados revelaram que os aplicativos web se comportam de maneira diferente dependendo do navegador usado para renderizá-los. Portanto, em nossa opinião, testes entre navegadores (ou seja, usar diferentes navegadores para testes de ponta a ponta) são essenciais para garantir um comportamento consistente dos aplicativos web em todos os navegadores.

Este artigo se concentrou no Selenium WebDriver como ferramenta de automação de navegador. No entanto, outras tecnologias de automação de navegador (como Cypress, Playwright ou Puppeteer) também podem usar o BrowserWatcher. Por exemplo, ao usar o Cypress (que utiliza navegadores locais), os desenvolvedores podem instalar o BrowserWatcher no navegador a ser automatizado com o Cypress. Em seguida, um script Cypress pode interagir com o BrowserWatcher usando JavaScript (por exemplo, para coletar os logs do navegador ou registrar a sessão automatizada).
Por outro lado, os scripts Playwright e Puppeteer podem usar o BrowserWatcher de forma semelhante ao Selenium WebDriver, ou seja, instalando o BrowserWatcher usando sua própria API e interagindo com a extensão usando a API JavaScript do BrowserWatcher.

Por fim, vale mencionar que a solução padrão para reunir o console do navegador está em desenvolvimento no momento da redação deste texto. A evolução do W3C O protocolo WebDriver é o protocolo W3C WebDriver BiDi. O BiDi possui um módulo para coleta de logs, implementado graças à comunicação bidirecional entre o driver e o navegador. O primeiro navegador com suporte ao BiDi foi o Firefox. A partir da versão 101 (lançada em 31 de maio de 2022), uma sessão BiDi do WebDriver pode ser solicitada usando o protocolo WebDriver clássico (Mozilla MDN, 2022b). Portanto, planejamos oferecer suporte à coleta de logs no WebDriverManager usando o protocolo W3C BiDi em versões futuras.

A ideia é aprimorar o recurso de coleta de logs no WebDriver-Manager para que seja o mais completo possível. Para isso, planejamos tornar esse recurso configurável entre o BrowserWatcher (cross-browser, baseado em JavaScript), o recurso goog:loggingPrefs e o CDP (disponível apenas para navegadores baseados em Chromium) e, finalmente, via BiDi (a opção padrão no futuro).

No entanto, não está claro se o BiDi fornecerá um mecanismo abrangente para coletar todos os tipos de logs, incluindo as categorias não rastreáveis encontradas nesta pesquisa. Acreditamos que esses logs merecem mais atenção em pesquisas futuras (por exemplo, em observabilidade da web do lado do cliente) e desenvolvimento (por exemplo, para serem considerados no padrão BiDi).

6. Trabalhos relacionados

Embora os logs de erros que ocorrem no console do navegador sejam muito valorizados pelos profissionais durante as sessões de teste e depuração, existem poucos artigos científicos que os abordam. Pesquisando na web, é possível encontrar diversos artigos e blogs.

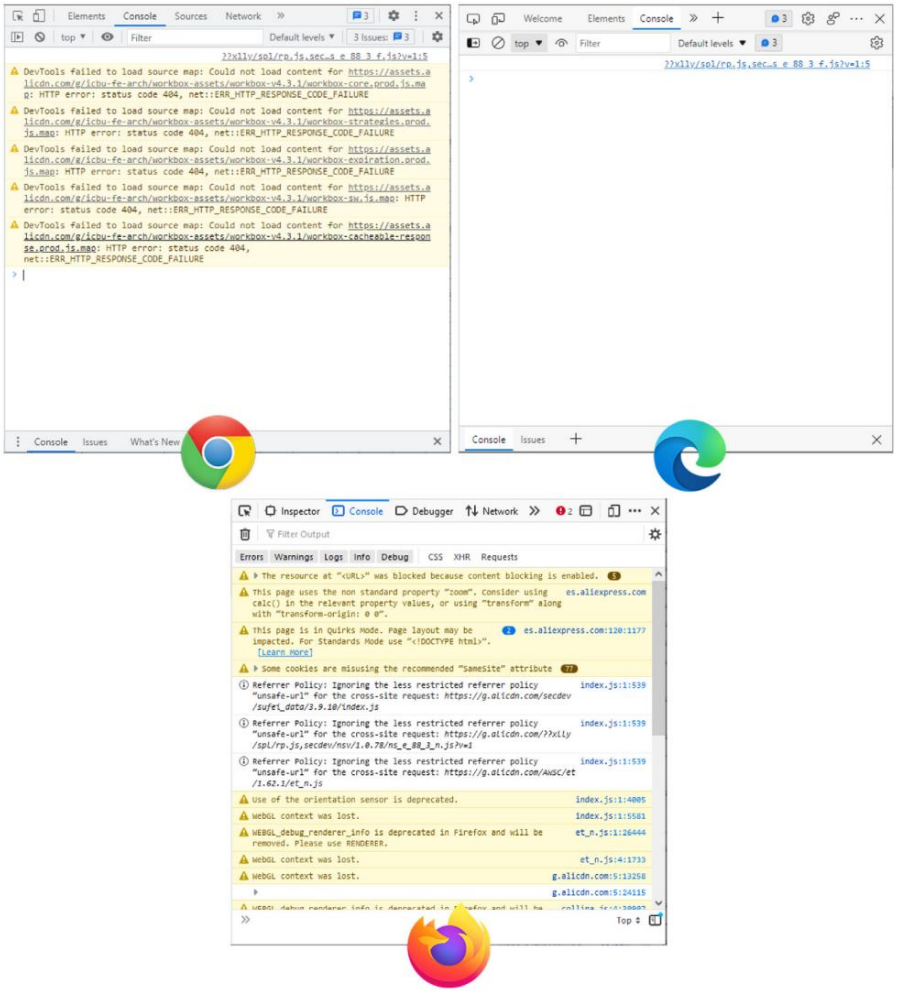


Fig. 19. Logs no Chrome, Edge e Firefox ao carregar https://live.com.

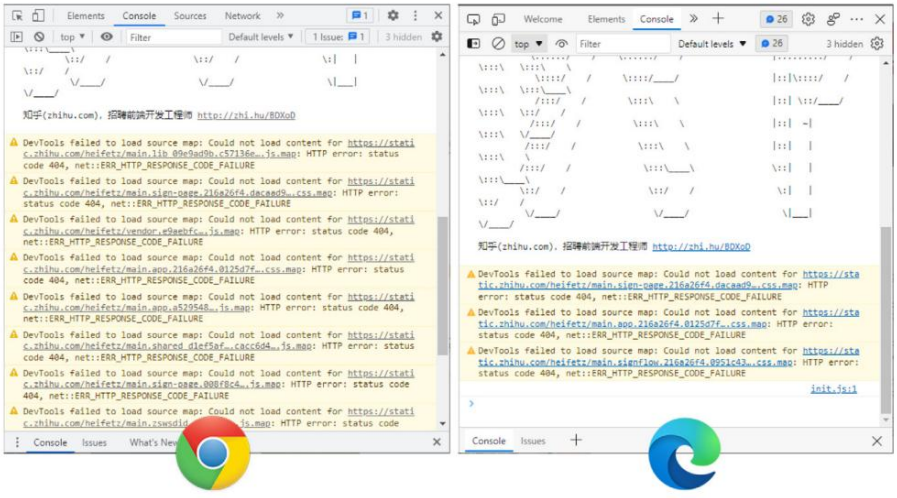


Fig. 20. Logs no Chrome e Edge ao carregar https://zhihu.com.

e posts (por exemplo, 1516) que explicam como usar um console e analisar os erros nele contidos. Há também alguns artigos na web que explicam como recuperar logs de erros usando a classe LogEntries e o método get() fornecidos pela API Selenium WebDriver (por exemplo, 17). Decidimos dividir os trabalhos relacionados em três subseções: análise de erros em código JavaScript, abordagens capazes de identificar erros e anomalias por meio da análise de logs de console e estudos relacionados ao WebDriverManager.

6.1. Erros de JavaScript

Um trabalho que enfatiza a importância do monitoramento de aplicações web do lado do cliente é o de Filipe e Araujo (2016). Para demonstrar as limitações das ferramentas de monitoramento atuais, os autores realizaram um experimento para revelar erros em páginas web em uma amostra de 3.000 sites (selecionados de forma semelhante à nossa usando o Alexa — sites mais bem classificados), incluindo erros de rede e JavaScript. Os resultados que os autores obtiveram de sua análise são impressionantes: até 16% dos 1.000 principais sites apresentam erros em seus próprios recursos; sites menos populares apresentam ainda mais. Os erros considerados nesse artigo são: erros de HTTP (4xx e 5xx), links quebrados e erros relacionados a recursos: fontes, folhas de estilo, imagens e JavaScript.

É interessante notar que uma grande proporção de erros se deve ao JavaScript e, em particular, aos recursos externos dos sites mais populares. Os autores concluem o artigo apresentando três possíveis abordagens diferentes para melhorar a observabilidade do cliente (que eles chamam de transparência). Entre essas abordagens, esboçamos a solução de extensão do navegador, que também consideramos em nossa proposta, para obter as métricas mais importantes da interação com a página web.

O objetivo de outro trabalho empírico apresentado por Ocariza et al. (2013) é compreender as causas-raiz e o impacto das falhas de JavaScript em aplicações web. Eles descobriram que a maioria das falhas de JavaScript está relacionada ao DOM, ou seja, são causadas por interações do código JavaScript com o DOM. Os autores concluem que a maioria das falhas de JavaScript se origina de erros do desenvolvedor cometidos no próprio código JavaScript, e não no código do lado do servidor ou HTML, e, como consequência, os desenvolvedores de JavaScript precisam de ferramentas de desenvolvimento/teste que os ajudem a encontrar erros de JavaScript e a raciocinar sobre o DOM.

Nossa ferramenta BrowserWatcher, que melhora a observabilidade do lado do cliente, é uma das ferramentas desejadas pelos autores.

Outro trabalho com o objetivo de analisar os erros de JavaScript exibidos no console de um navegador (especificamente o Firefox) é o de Ocariza et al. (2011). Eles utilizaram o FireBug, um complemento do navegador Firefox, atualmente sem manutenção,18 para capturar as mensagens de erro. Assim como nós, o conjunto de avaliação consiste em 50 sites da lista dos 100 mais visitados da Alexa. As contribuições deste trabalho são variadas, assim como as implicações para desenvolvedores, testadores e construtores de ferramentas. Entre as contribuições, mencionamos a metodologia sistemática que eles desenvolveram para executar aplicações web em múltiplas velocidades de teste e coletar suas mensagens de erro, das quais fomos parcialmente inspirados a realizar nosso experimento.

6.2. Detecção automatizada de anomalias por meio de logs do console

Os logs do console são usados não apenas em aplicativos da web, mas também para detecção automática de anomalias em sistemas complexos, por exemplo composto por centenas de componentes de software executados em milhares de nós de computação (Bao et al., 2018). Diferentemente da nossa proposta, que se concentra nos problemas de coleta efetiva de logs de diferentes navegadores, no contexto de sistemas complexos o principal problema reside na análise dos logs. De fato, uma enorme quantidade de dados em tempo de execução é continuamente coletada e armazenada em arquivos de log. Analisá-los para identificar causas e locais de problemas em caso de mau funcionamento e falhas do sistema é uma tarefa complexa e útil. Há, portanto, uma grande demanda por abordagens automáticas de detecção de anomalias baseadas em análise de logs (Bao et al., 2018). Como consequência, vários pesquisadores propuseram diferentes soluções. Para tentar resolver este problema, várias abordagens são possíveis, incluindo:

Técnicas baseadas em regras aprendem regras que capturam o comportamento normal de um sistema. Uma instância que não é coberta por nenhuma dessas regras é considerada uma anomalia (Chandola et al., 2009).

- Técnicas baseadas em análise de séries temporais analisam todo o log como uma única sequência de mensagens repetidas e tentam encontrar anomalias com métodos de análise de séries temporais (por exemplo, Yaman-ishi e Maruyama (2005)).

Técnicas baseadas em aprendizado dependem do aprendizado de máquina para detectar anomalias. Dependendo do tipo de dado envolvido e das técnicas de aprendizado de máquina empregadas, as abordagens de detecção de anomalias podem ser classificadas em duas subcategorias (He et al., 2016): detecção supervisionada de anomalias e detecção não supervisionada de anomalias.

Todas essas técnicas podem ser úteis também no contexto de aplicações web: de fato, os 50 principais sites considerados neste estudo podem ser considerados sistemas complexos, em particular da perspectiva do lado do servidor. No entanto, nosso estudo se concentra nos logs do console dos navegadores (portanto, apenas do lado do cliente). Durante a experimentação, descobrimos que, em geral, a quantidade de logs coletados ao visitar uma única página web não é enorme e pode ser razoavelmente analisada manualmente (como fizemos durante nosso estudo empírico). No entanto, o BrowserWatcher permite coletar logs durante a execução de suítes de teste, onde cada script de teste potencialmente visita dezenas de páginas web: nesse cenário de uso, as técnicas apresentadas na literatura podem ajudar a analisar rapidamente os logs e encontrar anomalias. Assim, o BrowserWatcher também pode ser considerado um facilitador fundamental para realizar tais análises avançadas em logs do console do lado do cliente.

6.3. Gerenciador de Drivers da Web

O WebDriverManager, a ferramenta que entre outras coisas possibilita o uso do BrowserWatcher, foi relatado pela primeira vez na pesquisa do ecossistema Selenium por García et al. (2020).
A pesquisa, realizada em 2019 por 72 participantes de 24 países, revelou como os profissionais utilizam o Selenium WebDriver em testes web. Com foco apenas na parte de gerenciamento de drivers, o estudo revelou que cerca de 39% dos entrevistados declararam gerenciar os gerenciadores de drivers manualmente, enquanto cerca de 35% declararam realizar esse processo automaticamente. Os usuários restantes (ou seja, cerca de 26%) afirmaram não saber como os drivers são gerenciados em seus conjuntos de testes. Esse resultado demonstra a disseminação do WebDriverManager.

A metodologia completa para realizar o processo de gerenciamento de drivers (ou seja, download, configuração e manutenção) e a ferramenta WebDriverManager são apresentadas em García et al .
Além de apresentar em detalhes a arquitetura e a API do WebDriverManager, o artigo avaliou, por meio de uma pesquisa, também a usabilidade do WebDriverManager. Os entrevistados consideraram o WebDriverManager útil e utilizável. No entanto, os benefícios de

15 https://wordpress.org/support/article/usando-seu-navegador-para-diagnosticar-erros-de-javascript/ 16 https://yoast.com/help/como-encontrar-erros-de-javascript-com-seus-navegadores-console/ 17 https://thoughtcoders.com/blog/selenium-code-to-test-browser-console-error-logs/ 18 https://getfirebug.com/

O WebDriverManager (redução do esforço de desenvolvimento e melhoria da manutenibilidade) não foi considerado na pesquisa.

Os benefícios do WebDriverManager foram considerados em um trabalho subsequente (Leotta et al., 2022) , no qual foi conduzido um experimento controlado com 25 alunos de mestrado. Os resultados do experimento mostram que a adoção do WebDriverManager, em vez do gerenciamento manual de drivers, reduz significativamente o tempo necessário para configurar/atualizar um conjunto de testes multinavegador (economia média de mais de 33% do tempo).

O WebDriverManager foi mencionado como uma ferramenta útil por muitos profissionais, também no contexto de uma pesquisa que visava entender os desafios (e as possíveis soluções) dos testes de ponta a ponta com o Selenium WebDriver (Leotta et al., 2023).

Finalmente, em outro artigo, García et al. (2022) descrevem Selenium-Jupiter, uma extensão JUnit 5 para Selenium WebDriver.
O Selenium-Jupiter visa facilitar o desenvolvimento de suítes de testes de ponta a ponta usando o WebDriverManager e a integração com o Docker. O Selenium-Jupiter oferece recursos avançados para testes entre navegadores, testes de carga e solução de problemas. Agora que a nova versão do WebDriverManager inclui o Browser Watcher, o Selenium-Jupiter também será enriquecido com recursos relacionados à melhoria da observabilidade em aplicações web.

7. Conclusões

Testes automatizados de ponta a ponta são um dos mecanismos mais populares para garantir a qualidade e reduzir defeitos em aplicações web. No entanto, a análise de falhas nesses tipos de testes pode ser desafiadora, especialmente para sistemas complexos. Este artigo apresentou o BrowserWatcher, uma extensão de navegador que pode ser usada para instrumentar navegadores web e observar aplicações web. O BrowserWatcher é construído sobre a API WebExtensions. Internamente, ele usa objetos e ouvintes com monkey-patches para coletar logs do navegador (por exemplo, rastreamentos de console e erros) enquanto um site é testado.

Acreditamos que esta ferramenta pode ser útil para muitos profissionais em todo o mundo. Portanto, para facilitar seu uso e alcançar uma gama mais ampla de desenvolvedores, a integramos ao WebDriverManager, uma ferramenta auxiliar popular (usada principalmente para gerenciamento automatizado de drivers) do ecossistema Selenium. Dessa forma, os recursos de observabilidade fornecidos pelo BrowserWatcher estão disponíveis por meio de uma API simples, usada para controlar navegadores instrumentados programaticamente com o Selenium WebDriver.

Realizamos a validação experimental do conjunto de ferramentas proposto para avaliar o BrowserWatcher na coleta de logs gerados por diferentes navegadores (Chrome, Edge e Firefox) em 50 sites populares. Verificamos que as categorias de log (denominadas *rastreáveis* neste artigo) gerenciadas com os recursos JavaScript mencionados são sempre coletadas com o BrowserWatcher. No entanto, descobrimos outras categorias de log (denominadas *não rastreáveis* neste artigo) que, até onde sabemos, não podem ser coletadas com JavaScript. Consideramos esta uma questão em aberto que deve atrair a atenção dos profissionais em pesquisas e desenvolvimentos futuros.

Outra descoberta do estudo experimental apresentado nesta pesquisa é que o mesmo site pode produzir logs diferentes em diferentes navegadores. Esse resultado reforça o valor dos testes entre navegadores (ou seja, o uso de diferentes tipos de navegadores para testes web de ponta a ponta) para garantir que os aplicativos web se comportem conforme o esperado em qualquer navegador.

Planejamos manter o BrowserWatcher e o Web-DriverManager para beneficiar a comunidade de automação de testes. Em trabalhos futuros, planejamos oferecer suporte ao mecanismo de coleta de logs fornecido pelo protocolo W3C BiDi no WebDriverManager quando esse recurso estiver disponível em todos os principais navegadores. Outra possível pesquisa futura é estender esse trabalho para navegadores móveis.

Declaração de contribuição de autoria do CRediT

Boni García: Conceitualização, Metodologia, Software, Validação, Redação – rascunho original, Redação – revisão e edição.
Filippo Ricca: Metodologia, Validação, Redação – revisão e edição. **Jose M. del Alamo:** Conceitualização, Validação, Redação – revisão e edição. **Maurizio Leotta:** Metodologia, Validação, Redação – revisão e edição.

Declaração de interesses conflitantes

Os autores declaram não ter conhecimento de quaisquer interesses financeiros conflitantes ou relacionamentos pessoais que possam ter influenciado o trabalho relatado neste artigo.

Disponibilidade de dados

Todo o código-fonte e dados da validação experimental estão disponíveis publicamente em repositórios de código aberto do GitHub (linkados no artigo).

Agradecimentos

Este trabalho foi parcialmente apoiado em parte pelo Ministerio de Ciencia e Innovación-Agencia Estatal de Investigación, Espanha (10.13039/501100011033) através do projeto H2O Learn sob Grant PID2020-112584RB-C31, em parte pelo Governo Regional de Madrid através do Projeto e-Madrid-CM, Espanha sob Grant S2018/TCS-4307, e em parte apoiado pela Comunidade de Madrid e Universidade Politécnica de Madrid, Espanha, através do Programa de Pesquisa V-PRICIT Apoyo a la realización de Proyectos de I+D para jovens investigadores UPM-CAM, sob concessão APOYOJOVENES-QINIM8-72-PKGQ0J. Financiamento para Taxa de Processamento de Artigos (APC): Universidade Carlos III de Madrid (Contrato de Leitura e Publicação CRUE-CSIC 2023).

Referências

Amazon, 2022. Principais sites da Alexa. https://www.alexa.com/topsites. (Acessado online em 19 de abril de 2022).

Bao, L., Li, Q., Lu, P., Lu, J., Ruan, T., Zhang, K., 2018. Detecção de anomalias de execução em sistemas de larga escala por meio da análise de logs de console. J. Syst. Softw. 143, 172–186. http://dx.doi.org/10.1016/j.jss.2018.05.016.

Bertolino, A., 2007. Pesquisa em testes de software: Conquistas, desafios, sonhos. Em: O Futuro da Engenharia de Software. FOSE'07, IEEE, pp. 85–103.

Burns, D., Smith, M., 2022. WebDriver bidi. Rascunho do editor. https://w3c.github.io/ webdriver-bidi/. (Acessado online em 27 de maio de 2022).

Cáceres, M., Christiansen, KR, Giuca, M., Gustafson, A., Murphy, D., Kosti-ainen, A., 2022. Manifesto de aplicação web, rascunho de trabalho do W3C. https://https: //www.w3.org/TR/appmanifest/. (Acessado online em 29 de maio de 2022).

Cerioli, M., Leotta, M., Ricca, F., 2020. O que 5 milhões de anúncios de emprego nos dizem sobre testes: uma investigação empírica preliminar. Em: Anais do 35º Simpósio Anual da ACM sobre Computação Aplicada. pp. 1586–1594.

Chandola, V., Banerjee, A., Kumar, V., 2009. Detecção de anomalias: uma pesquisa. ACM Comput. Surv. 41 (3), http://dx.doi.org/10.1145/1541880.1541882.

Equipe Chrome, 2021. Scripts de conteúdo. https://developer.chrome.com/docs/ extensions/mv3/content_scripts/. (Acessado online em 3 de janeiro de 2023).

Equipe Chrome, 2022a. Ferramentas para desenvolvedores do Chrome. https://developer.chrome.com/docs/ferramentas de desenvolvimento/. (Acessado online em 29 de maio de 2022).

Equipe Chrome, 2022b. API de extensão do Chrome. https://developer.chrome.com/docs/extensions/reference/ . (Acessado online em 28 de maio de 2022).

Equipe Chrome, 2022c. API tabCapture. https://developer.chrome.com/docs/extensions/reference/tabCapture/ . (Acessado online em 28 de maio de 2022).

Equipe Chrome, 2023. Protocolo Chrome DevTools, módulo de log. https:// chromedevtools.github.io/devtools-protocol/tot/Log/. (Acessado online em 4 de janeiro de 2023).

Deveria, A., Schoors, L., 2022. Tabelas de suporte do navegador para política de segurança de conteúdo (CSP) em navegadores da web. https://caniuse.com/?search=CSP. (Acessado online em 28 de maio de 2022).

Filipe, R., Araujo, F., 2016. Técnicas de monitoramento do lado do cliente para sites. Em: 15º Simpósio Internacional IEEE de 2016 sobre Computação em Rede e Aplicações. NCA, pp. 363–366. http://dx.doi.org/10.1109/NCA.2016. 7778642.

García, B., 2022. Selenium WebDriver prático com Java. O'Reilly Media.
García, B., Gallego, M., Gortázar, F., Munoz-Organero, M., 2020. Um levantamento do ecossistema de selênio. Eletrônica 9 (7), 1067.
García, B., Kloos, CD, Alario-Hoyos, C., Munoz-Organero, M., 2022. Selenium- jupiter: uma extensão junit 5 para selenium WebDriver. J.Sist. Suave. 111298.

García, B., Munoz-Organero, M., Alario-Hoyos, C., Kloos, CD, 2021. Gerenciamento automatizado de driver para Selenium WebDriver. Império. Suave. Eng. 26 (5), 1–51.
He, S., Zhu, J., He, P., Lyu, MR, 2016. Relato de experiência: Análise de logs de sistemas para detecção de anomalias. Em: 27º Simpósio Internacional de Engenharia de Confiabilidade de Software do IEEE de 2016. ISSRE, pp. 207–218. http://dx.doi.org/10.1109/ISSRE. 2016.21.

Hossain, M., 2014. CORS em ação: criando e consumindo APIs de origem cruzada. Simon e Schuster.
Hunt, J., 2019. Monkey patching e pesquisa de atributos. Em: Um Guia para Iniciantes em Programação em Python 3. Springer, pp. 325–336.
IEEE, 1990. Glossário padrão IEEE de terminologia de engenharia de software. Em: IEEE Std 610.12-1990. IEEE, pp. 1–84. http://dx.doi.org/10.1109/IEEESTD. 1990.101064.

Kalman, RE, 1970. Palestras sobre Controlabilidade e Observabilidade. Representante técnico, Departamento de Pesquisa Operacional, Universidade Stanford, Stanford, CA.
Leotta, M., Clerissi, D., Ricca, F., Tonella, P., 2016a. Abordagens e ferramentas para testes web automatizados de ponta a ponta. Em: Memon, A. (Ed.), Adv. Comput. 101, 193–237. http://dx.doi.org/10.1016/bs.adcom.2015.11.007.
Leotta, M., García, B., Ricca, F., 2022. Um estudo empírico para quantificar os benefícios de configuração e manutenção da adoção do WebDriverManager. Em: Vallecillo, A., Visser, J., Pérez-Castillo, R. (Orgs.), Anais da 15ª Conferência Internacional sobre Qualidade da Tecnologia da Informação e Comunicação. QUATIC 2022, Em: CCIS, vol. 1621, Springer, pp. 31–45. http://dx.doi.org/10.1007/978-3-031-14179-9_3 .

Leotta, M., García, B., Ricca, F., Whitehead, J., 2023. Desafios dos testes ponta a ponta com Selenium WebDriver e como enfrentá-los: Uma pesquisa. Em: Anais da 16ª Conferência Internacional IEEE sobre Teste, Verificação e Validação de Software . ICST 2023, IEEE, p. (no prelo).
Leotta, M., Ricca, F., Tonella, P., 2021. SIDEREAL: Geração adaptativa estatística de localizadores robustos para testes web. Em: Xie, T., Hierons, RM (Eds.), J. Softw. Teste. Verif. Confiável. (STVR) 31, http://dx.doi.org/10.1002/stvr.1767.
Leotta, M., Stocco, A., Ricca, F., Tonella, P., 2016b. ROBULA+: Um algoritmo para gerar localizadores XPath robustos para testes web. In: Canfora, G., Dalcher, D., Raffo, D. (Eds.), J. Softw. Evol. Processo (JSEP) 28 (3), 177–204. http://dx.doi. org/10.1002/smr.1771.

Mozilla MDN, 2022a. Extensões de navegador. https://developer.mozilla.org/en-US/ docs/Mozilla/Add-ons/WebExtensions. (Acessado online em 28 de maio de 2022).
Mozilla MDN, 2022b. Notas de lançamento do Firefox 101. https://developer.mozilla.org/en- US/docs/Mozilla/Firefox/Releases/101. (Acessado online em 31 de maio de 2022).
Nass, M., Alégroth, E., Feldt, R., Leotta, M., Ricca, F., 2023. Localização de elementos web baseada em similaridade para automação robusta de testes. ACM Trans. Software Eng. Methodol. (TOSEM) (no prelo). http://dx.doi.org/10.1145/3571855.
Niedermaier, S., Koetter, F., Freymann, A., Wagner, S., 2019. Sobre observabilidade e monitoramento de sistemas distribuídos – um estudo de entrevista na indústria. Em: Conferência Internacional sobre Computação Orientada a Serviços. Springer, pp. 36–52.

Ocariza, F., Bajaj, K., Pattabiraman, K., Mesbah, A., 2013. Um estudo empírico de bugs de JavaScript do lado do cliente. Em: Simpósio Internacional ACM/IEEE de 2013 sobre Engenharia e Medição de Software Empírica. pp. 55–64. http://dx.doi.org/10.1109/ESEM.2013.18 .

Ocariza, Jr., FS, Pattabiraman, K., Zorn, B., 2011. Erros de JavaScript na prática: Um estudo empírico. Em: 22º Simpósio Internacional de Engenharia de Confiabilidade de Software do IEEE de 2011. pp. 100–109. http://dx.doi.org/10.1109/ISSRE.2011.28.

Peña Moreno, N., 2022. Cronograma de desempenho nível 2, rascunho de trabalho do W3C. https://w3c.github.io/performance-timeline/ . (Acessado online em 29 de maio de 2022).
Quadri, S., Farooq, SU, 2010. Teste de software – objetivos, princípios e limitações. Int. J. Comput. Aplicação. 6 (9), 1.
Solntsev, A., 2019. Como obter logs do navegador. https://selenide.org/2019/12/16/ advent-calendar-browser-logs/. (Acessado online em 28 de maio de 2022).
Stewart, S., Burns, D., 2022. WebDriver, rascunho de trabalho do W3C. https://www.w3.org/TR/webdriver/. (Acessado online em 27 de maio de 2022).
Yamanishi, K., Maruyama, Y., 2005. Mineração dinâmica de syslog para monitoramento de falhas de rede. Em: Anais da Décima Primeira Conferência Internacional ACM SIGKDD sobre Descoberta de Conhecimento em Mineração de Dados. KDD '05, Association for Computing Machinery, Nova York, NY, EUA, pp. 499–508. http://dx.doi.org/10.1145/1081870.1081927 .

Boni Garcia é professor associado (com estabilidade) na Universidade Carlos III de Madrid, na Espanha. Seu principal interesse de pesquisa é engenharia de software, com foco em testes automatizados. Ele é engenheiro de software da Sauce Labs, no Escritório do Programa de Código Aberto. Ele é um dos responsáveis pelo projeto Selenium e criador de diversos projetos pertencentes ao seu ecossistema, como WebDriverManager, Selenium-Jupiter e BrowserWatcher. Ele escreveu os livros Mastering Software Testing with JUnit 5 (Packt Publishing, 2017) e Hands-On Selenium WebDriver with Java (O'Reilly Media, 2022). É autor de mais de 45 artigos de pesquisa em diferentes periódicos e conferências.

Filippo Ricca é professor associado da Universidade de Gênova, na Itália. Obteve seu doutorado em Ciência da Computação pela mesma universidade em 2003, com a tese "Análise, Teste e Reestruturação de Aplicações Web". É autor (ou coautor) de mais de 100 artigos científicos publicados em periódicos e conferências/workshops internacionais. Em 2011, recebeu o prêmio ICSE 2001 MIP (Artigo Mais Influente) pelo artigo "Análise e Teste de Aplicações Web" e, em 2018, recebeu o prêmio ICST MIP. De 1999 a 2006, trabalhou no grupo de Engenharia de Software do ITC-irst (hoje FBK-irst), em Trento, Itália. Durante esse período, fez parte da equipe que trabalhou em Engenharia Reversa, Reengenharia e Teste de Software. Seus atuais interesses de pesquisa incluem testes de aplicações Web e estudos empíricos em Engenharia de Software. A pesquisa é conduzida principalmente por meio de métodos empíricos, como estudos de caso, experimentos controlados e pesquisas.

Jose M. del Alamo é atualmente Professor Associado (com estabilidade) da Universidade Politécnica de Madrid (DIT-UPM). Seus interesses de pesquisa incluem questões relacionadas à gestão de dados pessoais, incluindo divulgação de dados pessoais, identidade, privacidade e gestão de confiança, e a consideração desses aspectos para o avanço de metodologias de engenharia de software e sistemas, com foco na avaliação de sua qualidade. O Dr. Del Alamo é copresidente do Workshop Internacional de Engenharia de Privacidade do IEEE, co-localizado no Simpósio de Segurança e Privacidade do IEEE, desde 2015.

Maurizio Leotta é pesquisador da Universidade de Gênova, Itália. Obteve seu doutorado em Ciência da Computação pela mesma universidade em 2015, com a tese "Testes Web Automatizados: Análise e Redução do Esforço de Manutenção".
É autor ou coautor de mais de 90 artigos de pesquisa publicados em periódicos e conferências/workshops internacionais. Seus atuais interesses de pesquisa estão em engenharia de software, com foco particular nos seguintes temas: testes de aplicações web, móveis e IoT, automação de testes funcionais, engenharia de software empírica, modelagem de processos de negócios e engenharia de software orientada a modelos.