

Uma abordagem prática de testes automatizados utilizando Selenium WebDriver

Wallace Creton da Costa

Centro Universitário Augusto Motta (UNISUAM)

Rua Paris, 72 – Bonsucesso – 21041-020 - Rio de Janeiro – RJ – Brasil

wallacedacosta@gmail.com

Abstract. *This article aims to clearly and objectively address the benefits of software testing using the Selenium WebDriver tool that provides agility, security, and resource depletion for systems with their implementation. With the increased use of web systems, coupled with a continuous search for more quality, fewer risks and better results, using Selenium WebDriver to facilitate the inclusion of Software Testing into the software development lifecycle becomes more and more important. We will demonstrate your application throughout the article highlighting its architecture, features and limitations.*

Resumo. *Este artigo pretende abordar de forma clara e objetiva os benefícios do teste de software utilizando a ferramenta Selenium WebDriver que proporciona agilidade, segurança e diminuição de recursos para os sistemas com a sua implementação. Com o aumento da utilização de sistemas web, associado a uma busca contínua por mais qualidade, menos riscos e melhores resultados, utilizar o Selenium WebDriver para facilitar a inclusão do Teste de Software ao ciclo de vida de desenvolvimento do software torna-se cada vez mais importante. Será apresentada sua aplicação ao longo do artigo destacando sua arquitetura, características e limitações.*

1. Introdução

Embora algumas empresas e equipes ainda desenvolvam softwares, sem a participação de um testador ou analista de testes, esse comportamento tende a ser cada vez mais raro, pois ter um profissional focado na qualidade do software é indispensável e pode comprometer o produto e a imagem da empresa.

Nenhuma empresa gostaria de ser taxada de ruim seja por vender um produto que não foi bem aceito no mercado pelos consumidores, como por ter vindo com defeito. Se já é difícil fazer algo que seja recebido com bons olhos então imagina algo que vem com defeito? Com certeza é ruim e com software não é diferente, segundo Bartié (2002), administrar um projeto de desenvolvimento de software para o sucesso significa eliminar ou minimizar os riscos e conflitos existentes. Existem diversos fatores que podem contribuir para a qualidade final do produto – profissionais experientes e bem treinados, metodologias e ferramentas adequadas. E mesmo assim podem existir contratempos que acabam por tornar bugs ou defeitos. Devido ao exposto, o teste tem importância primordial, para ser evitado que coloque em produção um software mal feito que cause descrédito por parte dos usuários custando caro, por isso a importância dos testes deve ser levada à sério.

2. Automação de Testes

Os testes de software podem chegar a ter tantos componentes, que sem auxílio de ferramentas de automação de testes seria muito difícil garantir a qualidade do mesmo. Além disso, o processo de teste como um todo e o seu gerenciamento ficaram mais complexos de serem executados.

Assim a grande saída são as ferramentas de testes, porém automação de testes é muito mais que apenas adquirir uma ferramenta. Implementar com clareza e funcionalidade exige conhecimento não só da ferramenta como também da regra de negócio envolvida no software.

Existem muitas ferramentas para tal tarefa, algumas pagas e outras alternativas gratuitas como Selenium WebDriver que iremos demonstrar ao longo do trabalho.

3. História do Selenium

Ele foi criado em 2004, nos laboratórios da ThoughtWorks em Chicago, por Jason Huggins, que na época trabalhava em um projeto interno chamado Time and Expense (T&E), que fazia uso extensivo da linguagem JavaScript.

O Selenium é uma ferramenta para testes de aplicações web que suporta diversos navegadores e uma das ferramentas mais conhecidas da atualidade para realizar a

automação de testes em aplicações web por meio da utilização do browser. Ele tem duas ferramentas, uma é o Selenium IDE, que é uma extensão do navegador Firefox e cria testes com gravações de passos e podem ser separados em planos de testes, e o Selenium WebDriver que iremos abordar neste artigo.



Figura 1 – Logo do Selenium IDE

A diferença entre os dois segundo o próprio site oficial se dá que o Selenium IDE serve para quando se quer criar scripts rápidos de reprodução de erros e scripts para auxiliar em testes exploratórios auxiliados por automação, já o Selenium WebDriver serve para quem quer criar robustos conjuntos de automação de regressão baseados em navegador e escalar e distribuir scripts em vários ambientes.

4. Selenium WebDriver



Figura 2 – Logo do Selenium WebDriver

Surgiu como “*Selenium Remote Control*” assim evoluindo com melhorias de código e se tornou o WebDriver. Sua implementação serve literalmente para dirigir um navegador, assim como nós fazemos ao navegar pelo browser o Selenium o faz. Essa API tem todos os recursos necessários para criarmos scripts em diferentes linguagens e assim fazer inúmeros tipos de testes e assim podendo abordar maiores cenários e até prepararmos testes de regressão de sistemas quando temos aplicações grandes e que estão em constante evolução, visto que não queremos que as novas versões não afetem o funcionamento do que já está em produção.

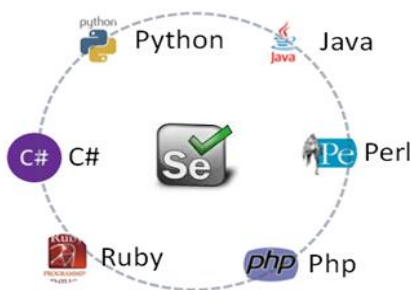


Figura 3 - Diversidade de Linguagens para trabalhar com Selenium WebDriver

É possível utilizar linguagens diversificadas para programar, são elas as mais comuns: Java, C#, Python, Rubi, Perl, PHP e Javascript, tudo depende da linguagem escolhida por conta da empresa, entidade, ou mesmo por parte da equipe de testes e seus programadores visto que a linguagem independe da utilizada para escrever a aplicação, pois se trata de um projeto em separado graças à biblioteca do Selenium

A API Selenium WebDriver tem como principal objetivo automatizar ações do navegador, tais como submits de formulários, seleções em menus dropdown, digitação em campos texto, varredura de dados em elementos, HTML etc. Neste artigo foi utilizado a linguagem C# para demonstrar sua utilização junto com o framework de testes **NUnit** e assim demonstrar a abordagem de como implementar os testes automatizados em sistemas web com o Selenium WebDriver.

5. Preparando o Ambiente

Visto que a linguagem escolhida neste estudo de caso foi C# então foi utilizado a IDE da Microsoft Visual Studio 2015, pode ser utilizada qualquer versão da IDE como, por exemplo, as versões express. Para o estudo, criou-se uma aplicação chamada Loja TCC <https://lojadotcc.000webhostapp.com> onde foi implementada um cadastro e consulta de clientes.

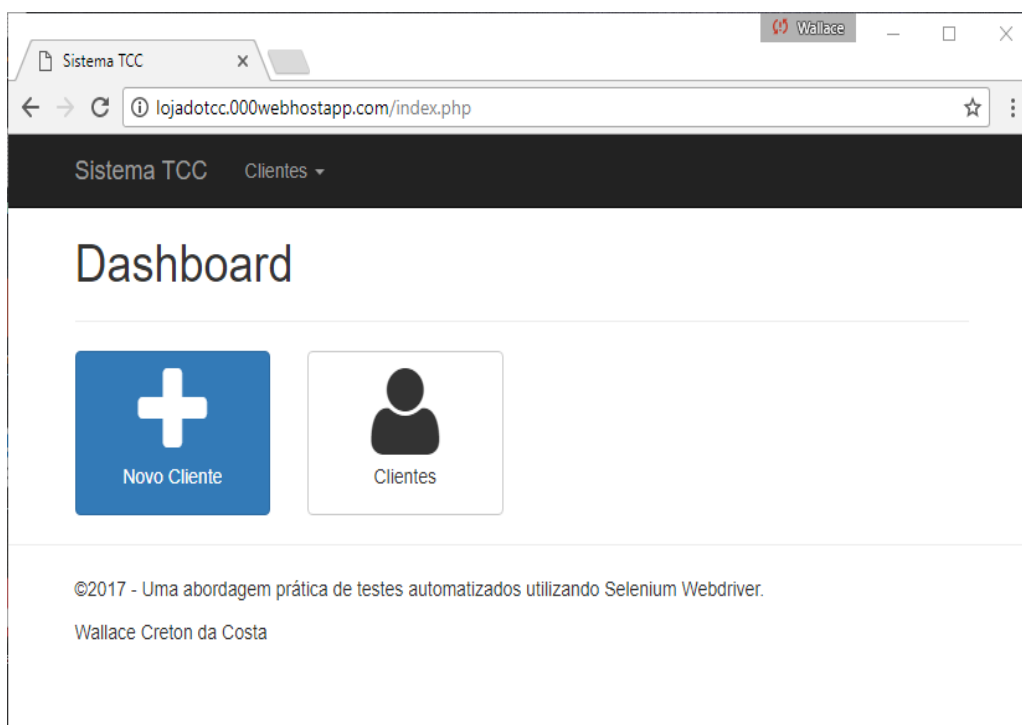


Figura 4 – Tela inicial da aplicação

Antes de começar o nosso projeto, é necessário instalar uma extensão para o Visual Studio. Esta extensão se chama NUnit Test Adapter e é responsável por detectar e rodar os testes da nossa Solution. Em Tools > Extensions And Updates.

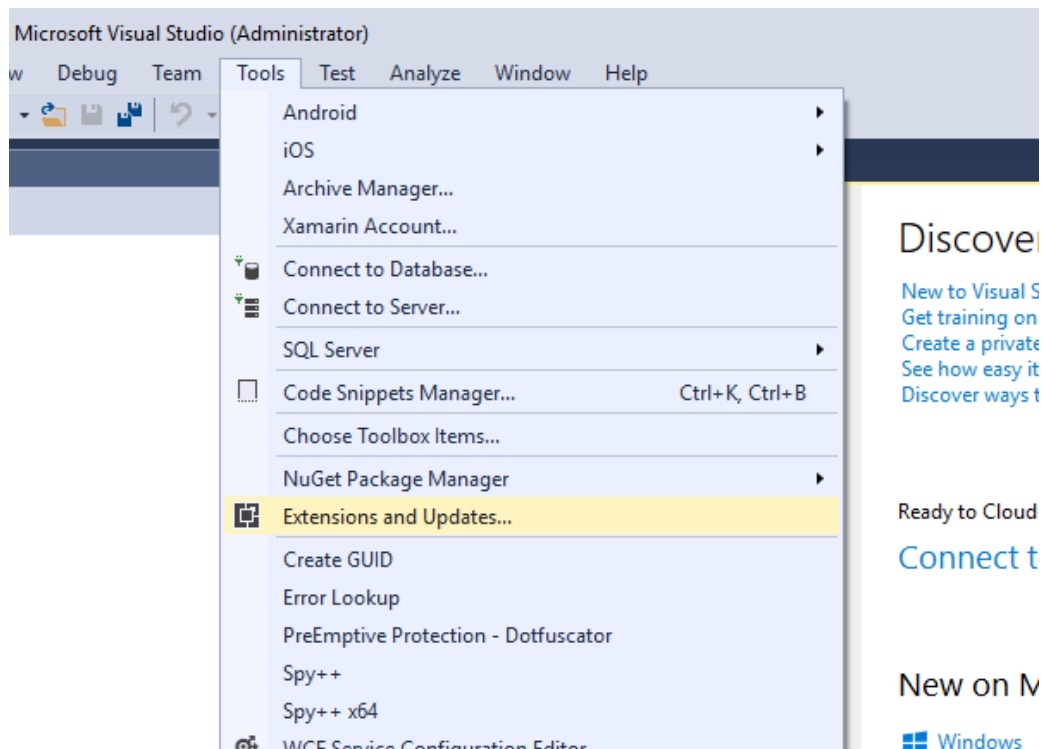


Figura 5

Clique na opção “Online” e procure por “NUnit”

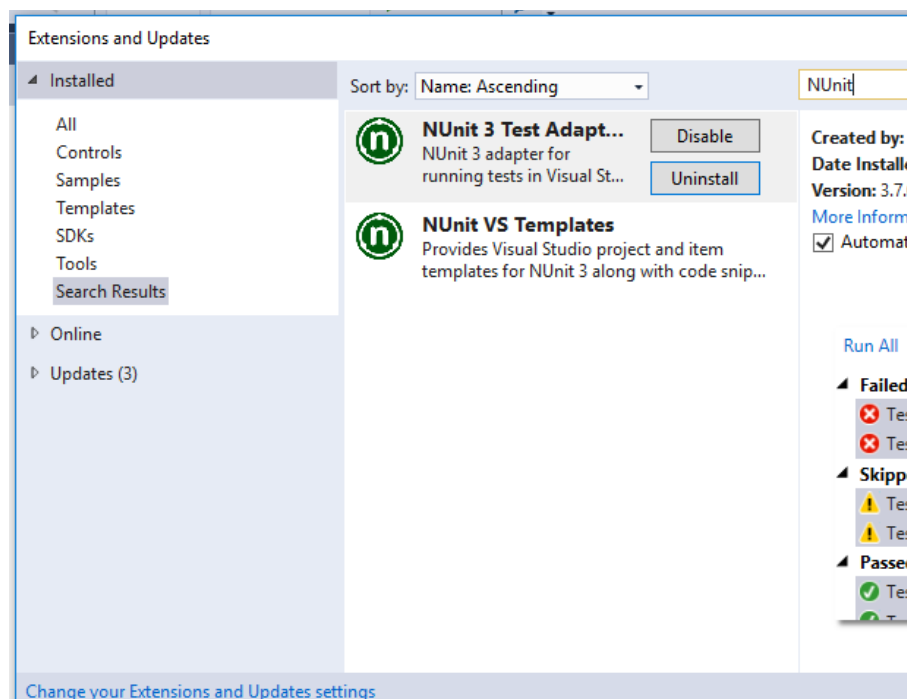


Figura 6 - Instalando NUnit

Após a instalação da extensão, dar-se continuidade ao projeto. Cria-se um novo projeto do tipo *Class Library*. Nomeado como Projeto TCC.

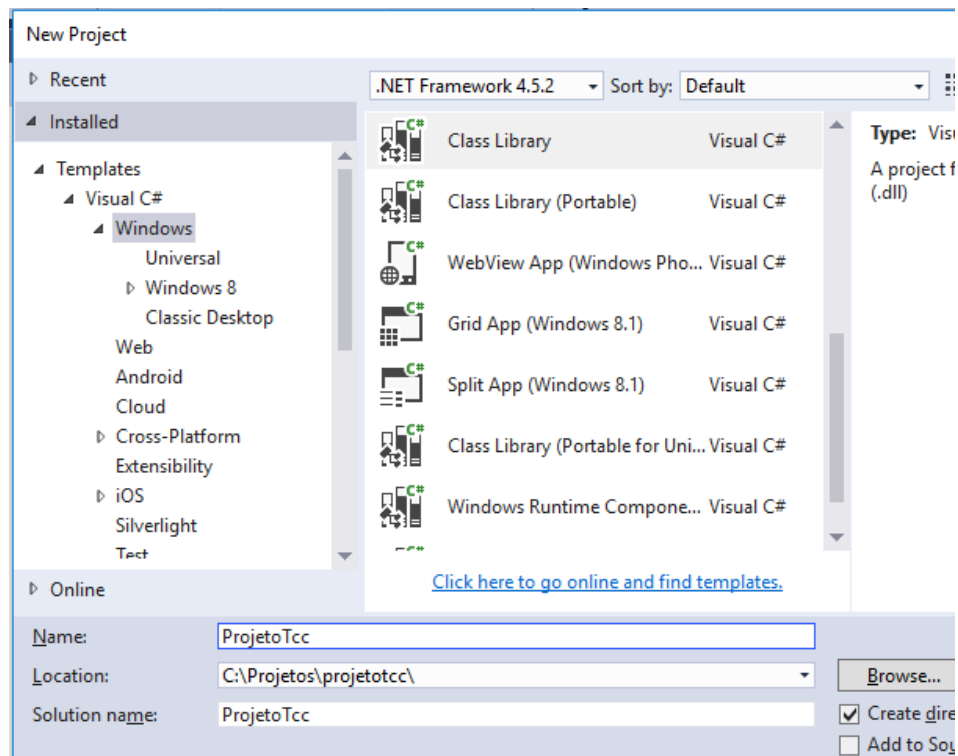


Figura 7 - Criando o Projeto

Com o projeto criado agora deve adicionar as referências ao projeto indo no Solution do projeto e com o botão direito selecionando “Manage NuGet Packages”.

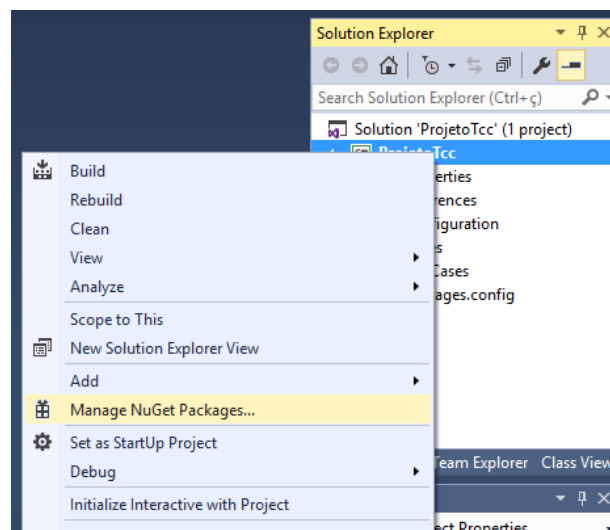


Figura 8

Agora deve procurar as bibliotecas do Selenium Webdriver, NUnit e Chrome Driver: Instale a primeira opção do NUnit como está em destaque:

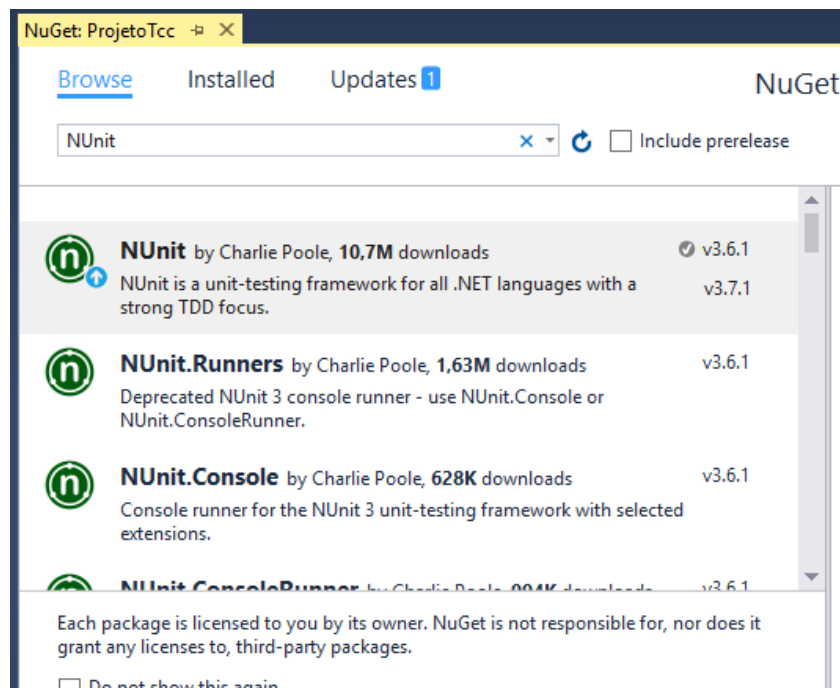


Figura 9

Instale em seguida o Selenium WebDriver e o WebDriver do Chrome para que o Selenium possa acessá-lo: como segue:

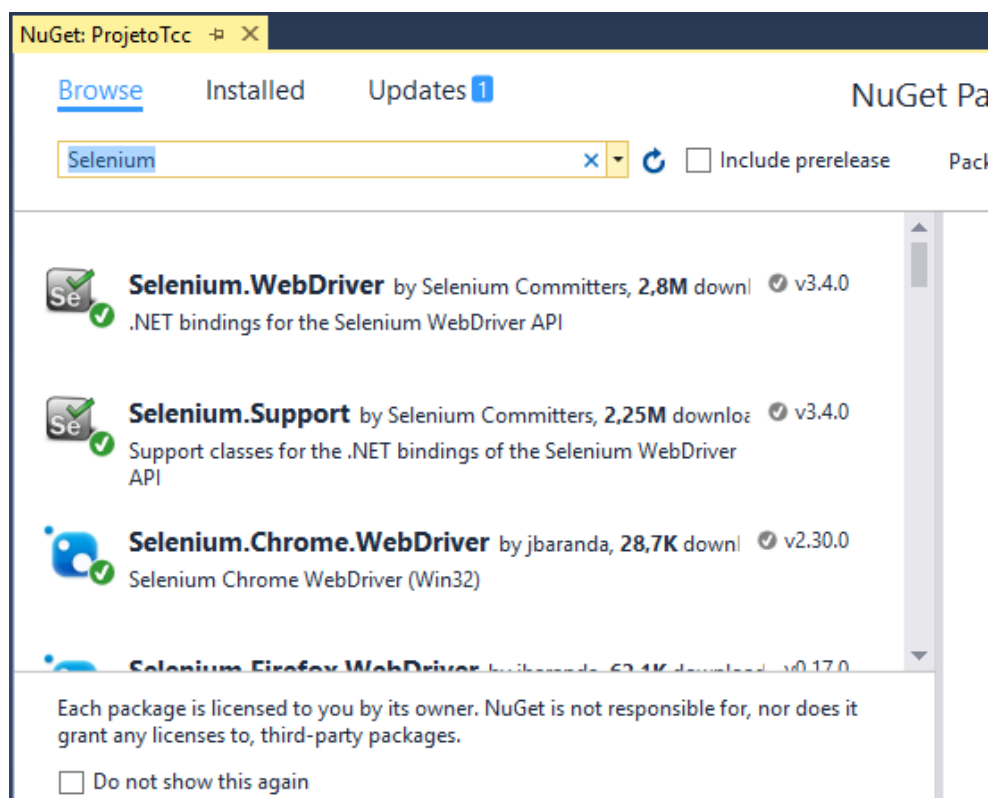


Figura 10

Agora que o ambiente preparado está apto para desenvolver o projeto. O código inicial e solution estão da seguinte forma:

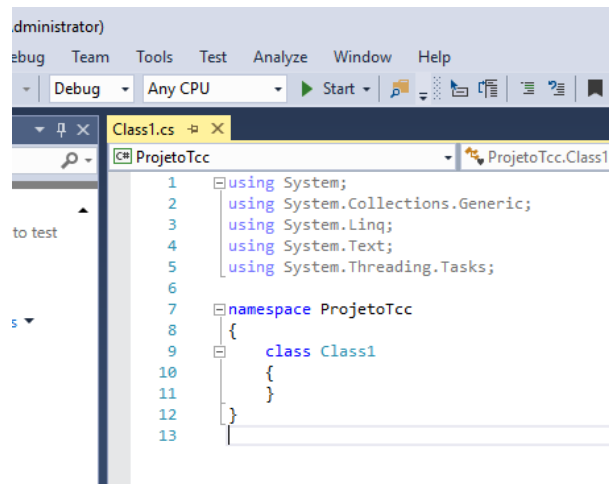


Figura 11

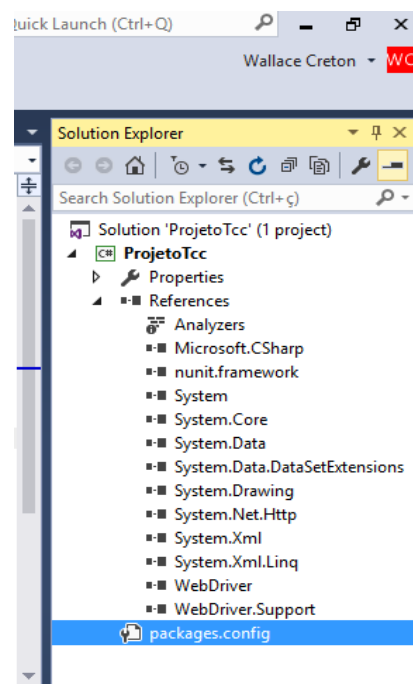


Figura 12

Para finaliza, deve criar 3 novas pastas:

- Configuration;
- Pages;
- TestCases.

Configuration ficará com os arquivos responsáveis por configurar o navegador e timeout. Pages será responsável pelo mapeamento das páginas da aplicação web e por fim TestCases será a pasta onde colocaremos os casos de testes.

A estrutura do projeto deve ficar conforme a imagem:

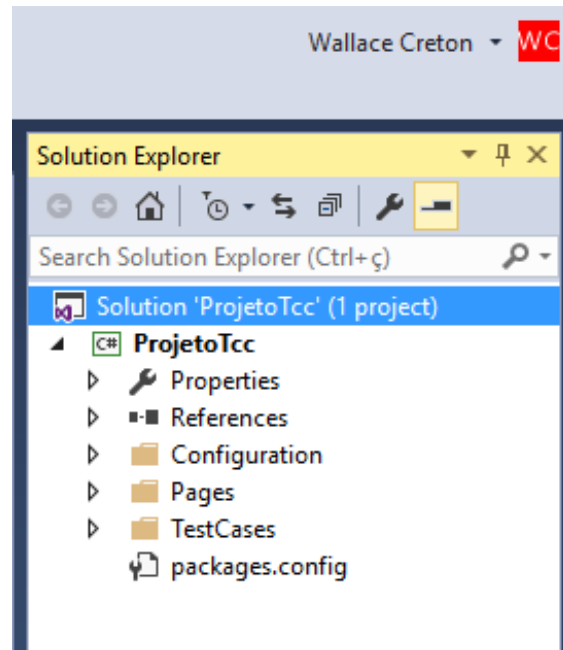


Figura 13

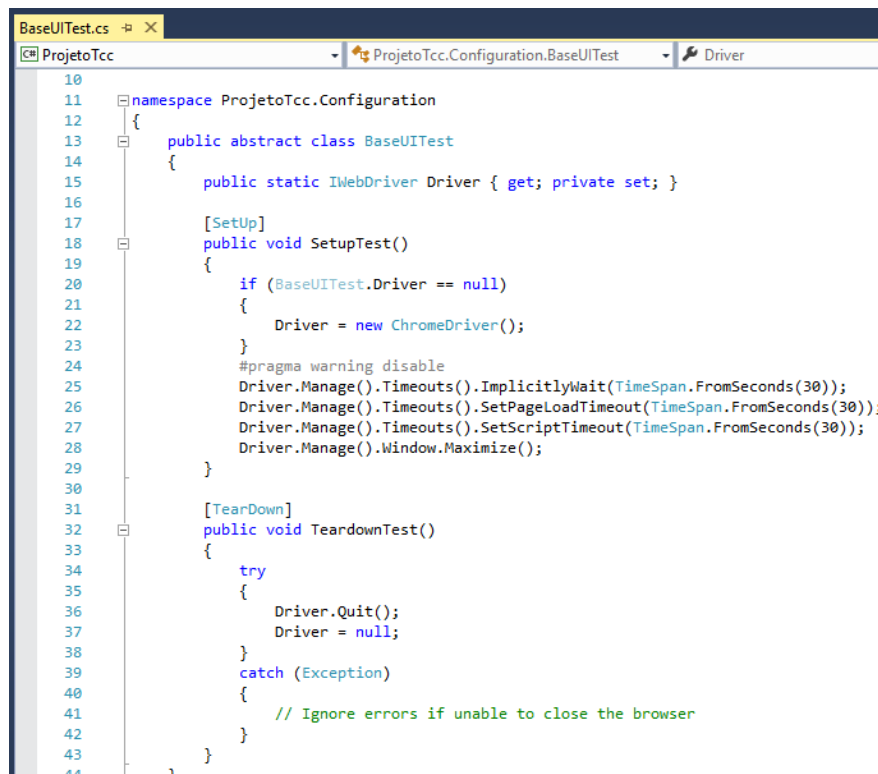
6. Configurando Navegador Chrome

Agora que a estrutura do projeto foi criada e todas referências necessárias adicionadas, é hora de começar a codificação. O NUnit disponibiliza dois eventos referentes aos testes que são:

- Setup: Executado toda vez que uma classe de testes for instanciada;
- TearDown: Executado toda vez que uma classe de testes for finalizada.

Pois bem, com estes dois eventos, deve implementar uma classe *Base* para os testes. Crie uma nova classe dentro da pasta *Configuration* e nomeie esta classe como *BaseUITest*.

O código da implementação segue abaixo:



```
10
11 namespace ProjetoTcc.Configuration
12 {
13     public abstract class BaseUITest
14     {
15         public static IWebDriver Driver { get; private set; }
16
17         [SetUp]
18         public void SetupTest()
19         {
20             if (BaseUITest.Driver == null)
21             {
22                 Driver = new ChromeDriver();
23             }
24             #pragma warning disable
25             Driver.Manage().Timeouts().ImplicitlyWait(TimeSpan.FromSeconds(30));
26             Driver.Manage().Timeouts().SetPageLoadTimeout(TimeSpan.FromSeconds(30));
27             Driver.Manage().Timeouts().SetScriptTimeout(TimeSpan.FromSeconds(30));
28             Driver.Manage().Window.Maximize();
29         }
30
31         [TearDown]
32         public void TeardownTest()
33         {
34             try
35             {
36                 Driver.Quit();
37                 Driver = null;
38             }
39             catch (Exception)
40             {
41                 // Ignore errors if unable to close the browser
42             }
43         }
44     }
45 }
```

Figura 14

Utilizou-se essa classe mais a frente para implementar o método de testes. O código acima é responsável por criar um driver para o Chrome, realizar configurações de Timeout referente à manipulação dos elementos da página e maximizar a janela do navegador que será criada toda vez que um teste for executado. Por fim finaliza o driver do Chrome.

7. Padrão de Design Page Object Model

Antes de se prosseguir com o projeto de testes, é importante entender um padrão de design de projetos automatizados de testes de interface. Este padrão é exemplificado no site do Selenium e explicado também no site do Martin Fowler, um dos responsáveis pelo padrão.

O padrão defende que deve criar classes que representem as propriedades e funcionalidades das páginas do sistema web com que deve interagir. Desta forma fica mais fácil reutilizar comportamentos e funcionalidades das páginas, uma vez que estamos programando conforme as funcionalidades que são expostas pelas próprias páginas.

Um resumo sobre este padrão:

- Representa as telas do seu aplicativo web encapsulando as funcionalidades que são disponibilizadas por suas respectivas páginas;
- Permite que modelemos a UI em nossos projetos de testes.

Vantagens:

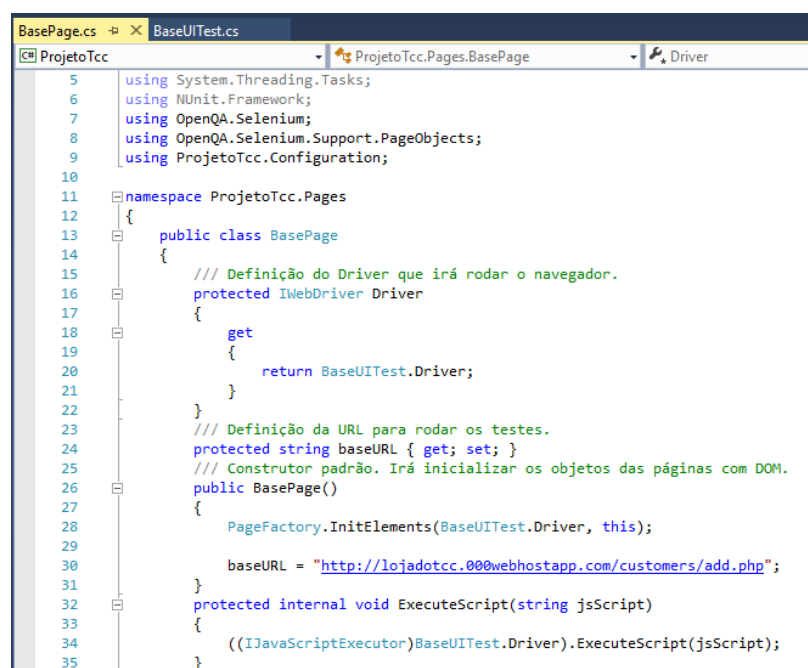
- Reduz os códigos duplicados;
- Facilita a leitura do que foi implementado no projeto de testes.
- Facilita a manutenção do projeto de testes.

A fim de dar continuidade ao nosso exemplo, deve-se criar um teste que realize um cadastro de novo cliente na nossa aplicação web e testar se foi cadastrado como deveria.

8. Caso de teste

Criado um caso de teste para exemplificar a utilização do Selenium WebDriver. O cenário será simples, mas nos dará a noção do que pode ser feito com essa ferramenta poderosa. Implementando uma automatização de cadastro de cliente que mostrará como resultado se o formulário de cadastro está salvando os dados de entrada com sucesso. Este teste tem como objetivo esta simples função, porém pode-se fazer muito mais se enriquecer os casos de testes podendo fazer testes de regressão de aplicações web como exemplo, automatizando processos e provando assim que os comportamentos do sistema não foram alterados devido novas implementações de customizações.

Primeiro precisa definir a classe que ficará responsável por definir a URL base para direcionar o navegador à página que se deseja, para isso deve se implementado a classe chamada BasePage a seguir:



```
5 using System.Threading.Tasks;
6 using NUnit.Framework;
7 using OpenQA.Selenium;
8 using OpenQA.Selenium.Support.PageObjects;
9 using ProjetoTcc.Configuration;
10
11 namespace ProjetoTcc.Pages
12 {
13     public class BasePage
14     {
15         /// Definição do Driver que irá rodar o navegador.
16         protected IWebDriver Driver
17         {
18             get
19             {
20                 return BaseUITest.Driver;
21             }
22         }
23         /// Definição da URL para rodar os testes.
24         protected string baseUrl { get; set; }
25         /// Construtor padrão. Irá inicializar os objetos das páginas com DOM.
26         public BasePage()
27         {
28             PageFactory.InitElements(BaseUITest.Driver, this);
29
30             baseUrl = "http://lojadotcc.000webhostapp.com/customers/add.php";
31         }
32         protected internal void ExecuteScript(string jsScript)
33         {
34             ((IJavaScriptExecutor)BaseUITest.Driver).ExecuteScript(jsScript);
35         }
36     }
37 }
```

Figura 15

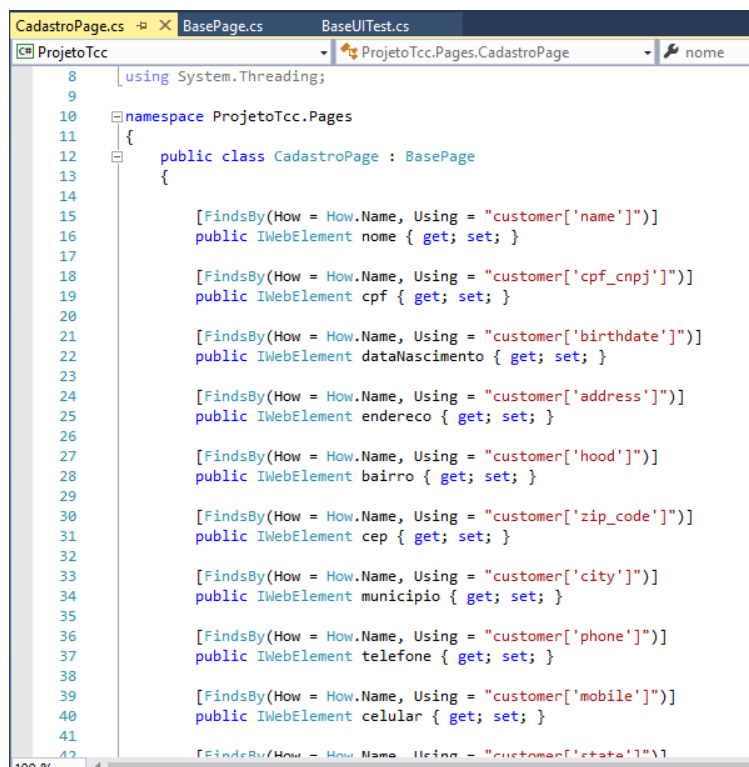
Após concluído este passo deve definir a classe responsável por manipular os elementos da página de cadastro, chamar de CadastroPage. Esse momento é importante para entender como funciona o processo de manipular o DOM e assim acessar o HTML, CSS e Javascript. Segue alguns dos comandos mais utilizados:

- CssSelector
- Id
- LinkText
- Name
- PartialLinkText
- TagName
- XPath

O exemplo a seguir utilizado no projeto e demonstra que irá procurar por um elemento do tipo Name = customer['name']. Assim tem-se mapeado para mais à frente no caso de teste invocar esse elemento mapeado e realizar uma ação.

```
[FindsBy(How = How.Name, Using = "customer['name']")]  
public IWebElement nome { get; set; }
```

Parte do código do CadastroPage implementado onde é mapeado os elementos do cadastro de clientes:



```
CadastroPage.cs | BasePage.cs | BaseUITest.cs  
ProjetoTcc | ProjetoTcc.Pages.CadastroPage | nome  
8 using System.Threading;  
9  
10 namespace ProjetoTcc.Pages  
11 {  
12     public class CadastroPage : BasePage  
13     {  
14  
15         [FindsBy(How = How.Name, Using = "customer['name']")]  
16         public IWebElement nome { get; set; }  
17  
18         [FindsBy(How = How.Name, Using = "customer['cpf_cnpj']")]  
19         public IWebElement cpf { get; set; }  
20  
21         [FindsBy(How = How.Name, Using = "customer['birthdate']")]  
22         public IWebElement dataNascimento { get; set; }  
23  
24         [FindsBy(How = How.Name, Using = "customer['address']")]  
25         public IWebElement endereco { get; set; }  
26  
27         [FindsBy(How = How.Name, Using = "customer['hood']")]  
28         public IWebElement bairro { get; set; }  
29  
30         [FindsBy(How = How.Name, Using = "customer['zip_code']")]  
31         public IWebElement cep { get; set; }  
32  
33         [FindsBy(How = How.Name, Using = "customer['city']")]  
34         public IWebElement municipio { get; set; }  
35  
36         [FindsBy(How = How.Name, Using = "customer['phone']")]  
37         public IWebElement telefone { get; set; }  
38  
39         [FindsBy(How = How.Name, Using = "customer['mobile']")]  
40         public IWebElement celular { get; set; }  
41  
42         [FindsBy(How = How.Name, Using = "customer['state']")]  
43     }  
100 %
```

Figura 16

Neste trecho tem mapeados os campos: Nome, CPF, Data de Nascimento, Endereço, Bairro e Cep, que são alguns dos campos necessários para cadastrar um cliente.

Finalizado o mapeamento foi criado o caso de teste na pasta TestCases, o teste que irá realizar o cadastro do cliente e verificará se foi cadastrado realmente, provando se a persistência dos dados inseridos foi realizada com sucesso. Assim iniciaremos criando a classe CadastroCliente e cadastrarClienteVerificarSeSalvou como o roteiro que definirá o teste.

```
namespace ProjetoTcc.TestCases
{
    [TestFixture]
    public class CadastroCliente : BaseUITest
    {
        [Test]
        public void cadastrarClienteVerificarSeSalvou()
        {
            CadastroPage cliente = new CadastroPage();
            cliente.NavegarParaPaginaPrincipal();

            cliente.nome.SendKeys("Wallace Costa");
            cliente.cpf.SendKeys("123.456.789-10");
            cliente.dataNascimento.SendKeys("1989-09-20");
            cliente.endereco.SendKeys("Avenida Paris, 100");
            cliente.bairro.SendKeys("Bonsucesso");
            cliente.cep.SendKeys("22111333");
            cliente.municipio.SendKeys("Rio de Janeiro");
            cliente.telefone.SendKeys("40042020");
            cliente.celular.SendKeys("980802020");
            cliente.uf.SendKeys("RJ");
            cliente.ie.SendKeys("123456");
            cliente.salvar.Click();

            Thread.Sleep(5000);

            Assert.IsTrue(Driver.PageSource.Contains("Wallace Costa"));
            //Vai salvar em C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE
            StreamWriter file = new StreamWriter("relatorio.txt");
            file.Write("Teste realizado com sucesso!");
            file.Close();
        }
    }
}
```

Figura 17

No código acima irá realizar as ações de escrever onde utilizou-se SendKeys e a ação de Click para clicar no botão salvar. Pode reparar que o código está fazendo exatamente o que uma pessoa faria ao cadastrar um usuário e o código está bem limpo e claro no que está realizando. Em seguida definiu-se um thread para realizar um intervalo, pois pode ser esperado nesse momento uma demora da resposta do servidor para salvar os dados e carregar a próxima página. Logo após foi feito um Assert.IsTrue que nesse caso procura no código fonte da página se contém o cliente recém cadastrado e por fim definiu-se um arquivo onde guarda o log do teste realizado. Este log poderia salvar inúmeros detalhes do teste por exemplo.

9. Rodando o Teste

Feito todos os passos de parametrização e escrita do roteiro, chegou a hora de executar o teste e essa parte com o NUnit fica fácil e já foi preparada anteriormente. Primeiro compilou-se o nosso projeto selecionando F6 ou Build Solution como segue:

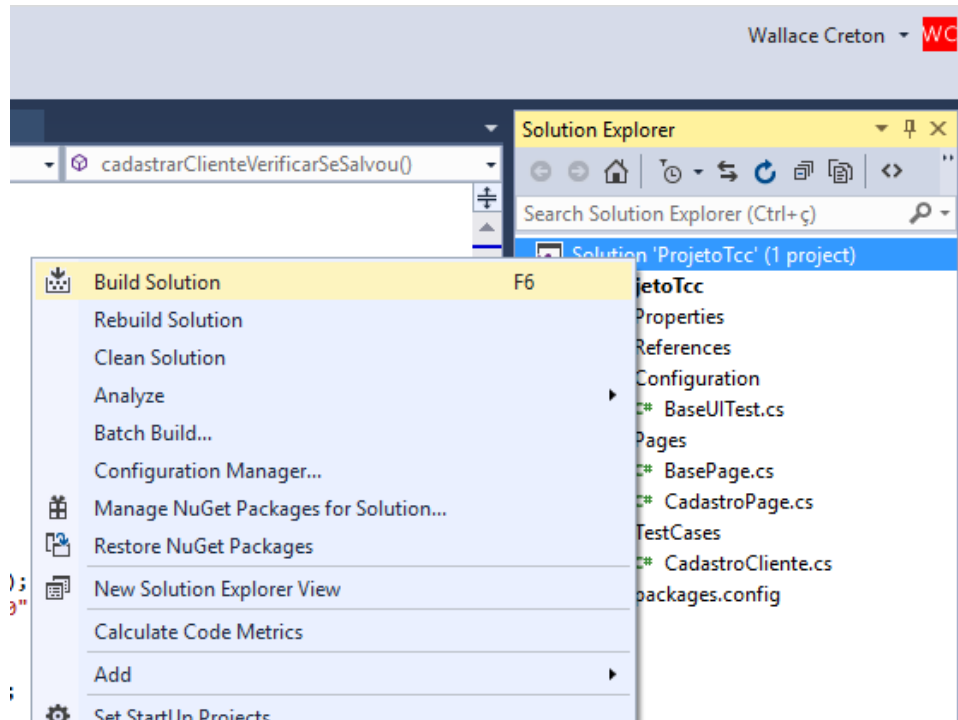


Figura 18

Compilado o projeto então seguiu no menu selecionando a opção Test > Windows > Test Explorer

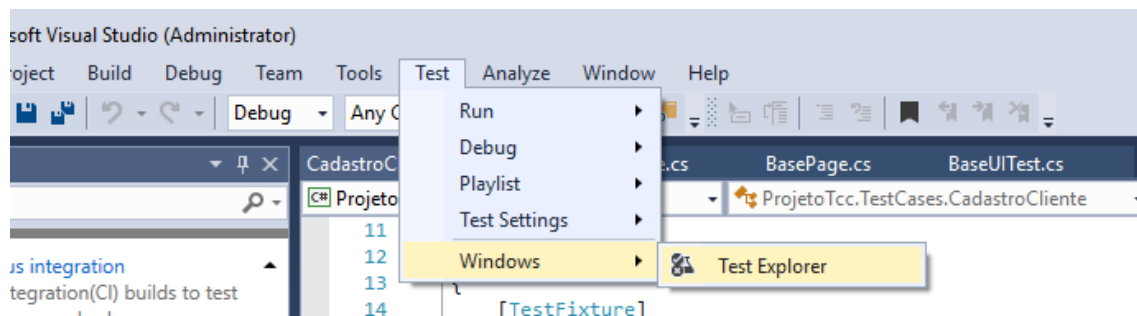


Figura 19

Isso abrirá a seguinte janela com o caso de teste criado:

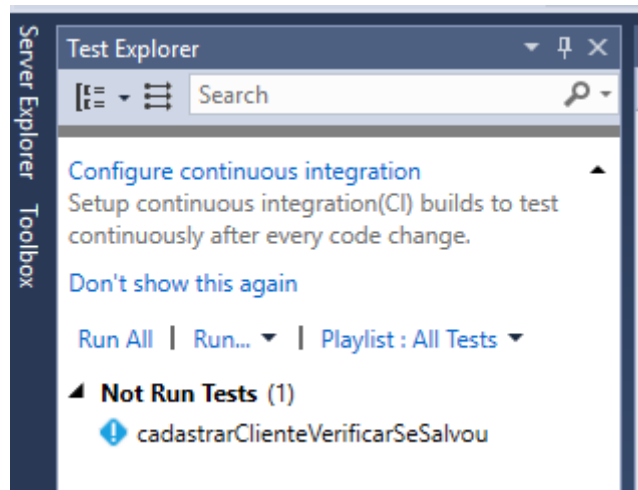


Figura 20

Agora deve selecionar Run All para rodar o teste. Este teste irá abrir o navegador, preencher os dados do formulário, clicar em salvar, confirmar se foi criado o usuário, gravar o log, fechar o navegador e exibir se o teste foi positivo ou falhou como poderia acontecer se por exemplo a conexão com o banco dessa aplicação estivesse com problemas. Iremos demonstrar as telas a seguir:

The image shows a web browser window with the address 'lojadotcc.000webhostapp.com/customers/add.php'. The page title is 'Sistema TCC' and 'Clientes'. The main heading is 'Novo Cliente'. The form has several input fields: 'Nome / Razão Social' (Wallace Costa), 'CNPJ / CPF' (123.456.789-10), 'Data de Nascimento' (1989-09-20), 'Endereço' (Avenida Paris, 100), 'Bairro' (Bonsucesso), 'CEP' (22111333), 'Município' (Rio de Janeiro), 'Telefone' (40042020), 'Celular' (980802020), 'UF' (RJ), and 'Inscrição Estadual' (123456). The 'Inscrição Estadual' field is highlighted with a blue border. At the bottom, there are 'Salvar' and 'Cancelar' buttons.

©2017 - Uma abordagem prática de testes automatizados utilizando Selenium Webdriver.
Wallace Creton da Costa

Figura 21

Neste momento são preenchidos os dados e selecionado o botão salvar como o caso de teste prevê.

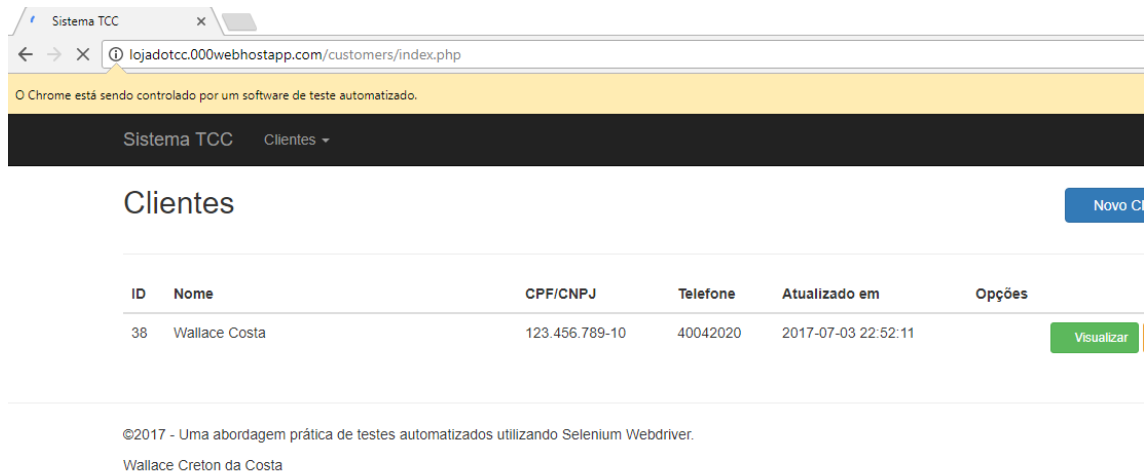


Figura 22

Neste momento temos o usuário cadastrado e irá acontecer a validação do Assert, após é guardado o log no arquivo relatorio.txt e exibido no NUnit o resultado do teste.

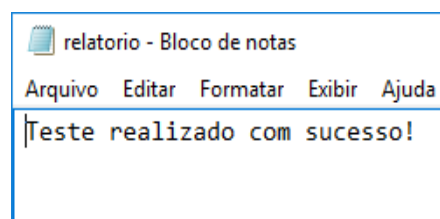


Figura 23

Fim do teste e neste caso pode-se verificar que não houve problemas de conexão com o banco e os cadastros não estão sendo prejudicados.

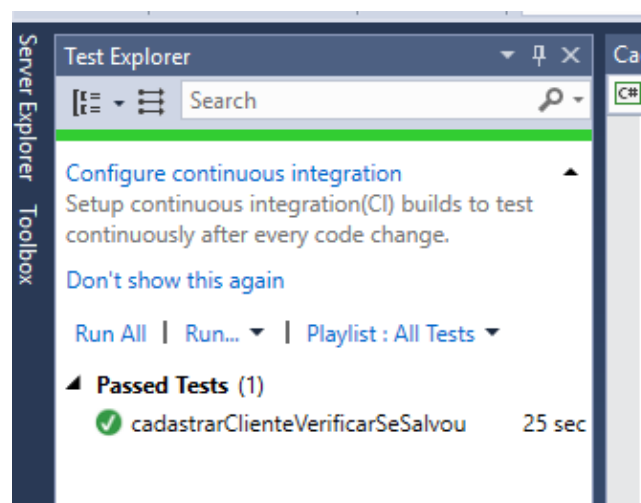


Figura 24

10. Conclusão

Nesse artigo procurou-se abordar de forma clara e simples como implementar o Selenium WebDriver, demonstrando suas qualidades e aplicações, para melhoria dos testes e qualidade dos sistemas web, para assim conseguir automatizar e ganhar agilidade em processos repetitivos. Viu-se que o mesmo pode manipular elementos HTML, CSS e Javascript através do DOM e assim fazendo um trabalho muito próximo do que um usuário faria. Assim entende-se que o Selenium WebDriver é uma ferramenta poderosa para busca contínua por mais qualidade, menos riscos e melhores resultados quando se fala de aplicações web.

Referência Bibliográfica

BARTIÉ, Alexandre (2002), Garantia da Qualidade de Software, Gulf Professional Publishing Editora.

MOLINARI, Leonardo (2013), Testes de Software – Produzindo Sistemas Melhores e Mais Confiáveis, Editora Érica Ltda.

RIOS, Emerson e TRAYAHÚ, Moreira Filho(2013), Teste de Software 3ª Edição revisada e atualizada, Editora Alta Books, 3ª edição