**Teste de acessibilidade automatizado**

À medida que mais e mais empresas se concentram em tornar seus aplicativos acessíveis, uma pergunta que geralmente surge é “Como podemos garantir que não deixemos as coisas escaparem?” Fizemos todo esse trabalho para remediar nosso aplicativo, mas como podemos ter certeza de que não voltaremos a nos afundar seis meses depois e terminaremos em uma situação semelhante à que estamos agora? Como mantemos as coisas acessíveis?

Existem algumas soluções. O primeiro é educação e treinamento. Todos na empresa precisam ser defensores da acessibilidade e entender as práticas recomendadas de acessibilidade. O segundo é melhores processos organizacionais. As empresas devem incluir auditorias de acessibilidade em pontos de verificação naturais ao longo do ciclo de vida de desenvolvimento de software, como quando a UX fornece maquetes de design para a equipe de engenharia ou quando a equipe de engenharia está completando o código de um novo recurso. O terceiro é o teste automatizado, e é nisso que eu gostaria de me concentrar hoje.

**Isenção de responsabilidade**

Como um breve aviso antes de começarmos, quero enfatizar que, quando se trata de acessibilidade, não há alternativa adequada para um bom teste manual com mouse, teclado e leitor de tela. Pergunte a qualquer consultor de acessibilidade e eles lhe dirão a mesma coisa.

O problema é que os engenheiros geralmente ficam insatisfeitos com essa resposta. Engenheiros gostam de automatizar tudo. O teste manual parece tedioso e não é dimensionado.

E, você estaria certo. Essas são preocupações justas. Então, vamos dar uma olhada em algumas das ferramentas automatizadas que temos disponíveis e examinar seus benefícios, bem como suas desvantagens.

**Ferramentas de acessibilidade automatizadas**

Existem várias boas ferramentas que podem nos auxiliar em nossos esforços de acessibilidade. Algumas das ferramentas comuns que usei são plugins ESLint como eslint-plugin-jsx-a11y, ferramentas do Deque como a extensão ax DevTools Chrome ou o rastreador web axe Monitor, e ferramentas CI como Google Lighthouse ou GitLab CI/CD com Pa11y.

A coisa a saber sobre todas essas ferramentas é que elas são todas verificadores de análise estática.

Os verificadores de análise estática examinam o código, seja o código-fonte JavaScript ou o HTML criado na página da Web, e relatam possíveis violações com base em um conjunto de regras.

E aqui está o kicker: esses verificadores de análise estática podem detectar apenas cerca de 10 a 30% dos problemas de acessibilidade em seu aplicativo.

Sim, você leu esse número corretamente. 10-30%. Por que esse número é tão baixo? Para entender melhor o porquê, devemos olhar para os tipos de coisas que essas ferramentas são boas em identificar, bem como as coisas que elas são ruins em identificar.

**O que os verificadores de análise estática são bons em identificar**

Os verificadores de análise estática são bons para identificar usos inválidos de HTML. Por exemplo, eles pegam quando você usa uma marca âncora (<a>) sem um atributo href. Talvez você tenha colocado um manipulador de cliques na tag âncora para fazê-la funcionar mais como um botão, o que seria inválido. O verificador de análise estática reportaria uma violação e informaria que você deveria usar um elemento <button> com um manipulador de cliques ou então fornecer um atributo href válido para seu elemento <a> se você realmente pretendesse que ele fosse um link.

Como outro exemplo, verificadores de análise estática podem identificar quando você usou elementos de título (<h1> a <h6>) na ordem errada. A regra é que os níveis de cabeçalho só podem aumentar em um, então você não pode ter um elemento <h1> seguido por um elemento <h4>. Se o verificador de análise estática vir isso em seu aplicativo, ele relatará uma violação.

Como terceiro exemplo, um verificador de análise estática também pode identificar se você aninha incorretamente elementos em uma lista. Os descendentes diretos de elementos <ul> ou <ol> precisam ser elementos <li>, portanto, se você tiver algo como um <div> como filho de seu contêiner de lista <ul> ou <ol>, o verificador de análise estática reclamar.

Os verificadores de análise estática também são bons para identificar usos ruins de funções e manipuladores de interação. Um erro comum que vejo com frequência é alguém usando um elemento <div> com um manipulador de clique em vez de um elemento <button>. O problema com essa abordagem sozinha é que você perde muito da funcionalidade que o elemento semântico <button> fornece para você fora da caixa. Por exemplo, o elemento <button> responde a cliques, bem como a pressionamentos de tecla Enter e Space, e comunica corretamente sua função (“botão”) aos leitores de tela. Um verificador de análise estática que analisa seu código-fonte (como eslint-plugin-jsx-a11y) relatará essas violações e informará que, se você tiver um manipulador de cliques, também precisará de um manipulador de interação de teclado, bem como uma função apropriada no seu elemento.

Por fim, os verificadores de análise estática executados no aplicativo renderizado em seu navegador também são ótimos para detectar problemas de contraste de cores quando a taxa de contraste de cores para qualquer combinação de primeiro plano e plano de fundo fica abaixo do limite necessário.

Como uma revisão rápida, estas são algumas das principais coisas que os verificadores de análise estática são bons em identificar:

* Usos inválidos de HTML.
* Mau uso de funções e manipuladores de interação.
* Problemas de contraste de cores.

**O que os verificadores de análise estática são ruins em identificar**

Agora, vamos falar sobre o que os verificadores de análise estática são ruins em identificar. A resposta curta é que eles serão ruins em identificar coisas que tenham código-fonte tecnicamente correto, mas que forneçam uma experiência de usuário ruim para humanos.

Por exemplo, vamos considerar a ordem de tabulação de uma página. Para a maioria dos idiomas ocidentais que lêem da esquerda para a direita, a ordem de tabulação na página geralmente vai da esquerda para a direita, de cima para baixo. Você pode ter um layout de coluna em sua página; nesse caso, a ordem de tabulação desceria uma coluna antes de passar para a próxima coluna. Ao percorrer uma página, às vezes você pode encontrar o foco da guia movendo-se para um elemento que você não esperava, talvez pulando alguns outros botões ou simplesmente indo para algum lugar completamente não relacionado. Isso é desorientador para um humano, mas não é algo que um verificador de análise estática seria capaz de capturar. Apenas um humano pode dizer qual ordem de tabulação faz sentido ou não.

Outro exemplo seriam rótulos de ária inúteis. Um verificador de análise estática será bom para informar quando um rótulo estiver ausente, como para um botão somente de ícone que não possui um rótulo de aria. Mas um verificador de análise estática não poderá dizer se o rótulo da ária é útil ou faz sentido. Você pode digitar alguns caracteres sem sentido como o valor aria-label para passar pelo verificador de análise estática, mas não será útil para seus usuários.

Terceiro, verificadores de análise estática não podem identificar armadilhas de teclado. Essas são armadilhas de foco não intencionais em que um usuário apenas de teclado não pode escapar sem usar o mouse. Você pode encontrar uma armadilha de teclado ao interagir com o conteúdo pop-up, como um modal ou uma dica de ferramenta ou um menu suspenso. Um usuário apenas de teclado precisa ser capaz de entrar e sair de armadilhas de teclado, então, se eles não puderem escapar, isso é um problema

Quarto, os verificadores de análise estática não conseguem identificar quando faltam funcionalidades alternativas na página para acomodar todos os usuários. Por exemplo, o comportamento de arrastar e soltar. A funcionalidade de arrastar e soltar é inerentemente inacessível porque requer o uso do mouse e controle motor fino para mover o ponteiro do mouse de uma posição específica para outra. Isso não é um problema por si só, mas você precisa fornecer métodos alternativos para realizar a mesma tarefa. Então, para algo como usar a funcionalidade de arrastar e soltar para reordenar itens em uma lista, você também pode fornecer controles de teclado para permitir que usuários apenas de teclado pressionem a tecla Enter para ativar o “modo de reordenação” e, em seguida, use as teclas de seta para mover itens para cima ou para baixo na lista. Os verificadores de análise estática não podem saber quando você tem métodos alternativos suficientes para realizar qualquer tarefa.

Quinto, os verificadores de análise estática não podem identificar áreas onde os usos semânticos de HTML podem ser melhorados. Por exemplo, talvez você tenha construído uma tabela com elementos <div>. Visualmente, parece uma tabela, mas não terá o mesmo comportamento de navegação para usuários de leitores de tela e não será comunicado como uma tabela para usuários de leitores de tela. Os verificadores de análise estática não reclamarão porque o código HTML real que você escreveu está tecnicamente correto sem nenhuma sintaxe inválida. O verificador de análise estática não sabe que você pretendia que isso representasse uma tabela.

Da mesma forma, você pode ter uma lista de itens na página que são criados usando elementos de parágrafo (<p>) em vez de elementos <ul>/<ol> e <li>. Ou talvez você tenha um modal de diálogo, mas está faltando toda a marcação modal necessária, como aria-modal="true", role="dialog" e um aria-label fornecendo um título para o modal. Os leitores de tela verão o HTML tecnicamente correto, mas não saberão sua intenção por trás do tipo de widgets ou informações que você está tentando transmitir.

Novamente, como uma revisão rápida, estas são algumas das principais coisas que os verificadores de análise estática são ruins em identificar:

* Ordem de tabulação confusa.
* Rótulos de ária inúteis.
* Armadilhas de teclado.
* Funcionalidade alternativa ausente.
* Áreas onde o HTML semântico pode ser melhorado.

**Humanos vs. Computadores**

Então, temos um dilema aqui. Como engenheiros, queremos ser capazes de automatizar nossos testes de acessibilidade. Mas as ferramentas que temos à nossa disposição não são suficientes por si só para nos dar confiança de que nosso aplicativo é de fato acessível. O que fazemos sobre isso?

A chave aqui é entender que os computadores são bons em algumas coisas e os humanos são bons em algumas coisas.

Os computadores são rápidos, não precisam de descanso e podem executar instruções com perfeição (mesmo quando nós, humanos, damos instruções incorretas!).

Os seres humanos, por outro lado, são melhores em pensamento e raciocínio de nível superior. Quando se trata de auditar a acessibilidade do nosso aplicativo, podemos dar um passo atrás e perguntar: “Isso faz sentido? Posso usar isto? O que construímos oferece uma boa experiência ao usuário?”

Então, em vez de competir, por que não deixar humanos e computadores trabalharem juntos para fornecer o melhor dos dois mundos?

Como seres humanos, podemos decidir quais critérios são importantes, o que vale a pena testar e qual deve ser o comportamento esperado. Podemos codificar esses requisitos como testes automatizados. Os computadores podem executar nossos testes e podemos incluir esses testes em um pipeline de integração contínua (CI) para evitar regressões de acessibilidade em nosso aplicativo.

Vejamos alguns exemplos.

**Exemplo 1: Modal**

Para nosso primeiro exemplo, vamos imaginar que estamos construindo um modal. Podemos encontrar orientações sobre como podemos construir modais acessíveis por meio das Diretrizes de Acessibilidade de Conteúdo da Web (WCAG), bem como dos documentos WAI-ARIA Authoring Practices.

Nossos critérios modais ficarão assim:

* **Modal é aberto quando o botão de gatilho é clicado.**
* **Modal tem marcação de ária apropriada (aria-modal="true", role="dialog", aria-label)​.**
* **O foco é enviado para o primeiro item focalizável dentro do modal quando ele é aberto.**
* **O foco está preso dentro do modal.**
* **O modal é fechado quando o botão Fechar é clicado e o foco é retornado ao botão de disparo​.**
* **O modal é fechado quando a tecla Escape é pressionada e o foco é retornado ao botão de disparo​.**
* **O modal é fechado quando qualquer lugar fora do modal é clicado e o foco é retornado ao botão de disparo​.**

Nossas próximas perguntas seriam, naturalmente, em que nível devemos testar esses critérios e como podemos escrever esses testes?

Ao escrever testes de acessibilidade, o nível correto para testá-los quase sempre será como testes unitários. Você não precisa escrever um teste de ponta a ponta para verificar se seu modal tem a marcação de ária correta. Testes unitários serão suficientes.

Então, como podemos escrever testes de unidade para esses critérios? Usando as mesmas ferramentas que você já usa para seus outros testes de unidade. Eu trabalho principalmente em React, então minhas ferramentas de escolha são Jest como minha estrutura de teste com React Testing Library e a biblioteca User Event como minhas bibliotecas de teste.

A React Testing Library é ótima para renderizar e interagir com componentes. User Event é uma biblioteca complementar que ajuda a simplificar ainda mais o teste de interações do usuário. É ótimo para testar coisas como o comportamento da guia ou disparar eventos que o documento está ouvindo.

**Exemplo 2: botão div clicável**

Vamos considerar outro exemplo. Discutimos os elementos <div> clicáveis anteriormente neste artigo e algumas das funcionalidades que você precisa reimplementar por conta própria se optar por usar um elemento diferente do elemento semântico <button>.

Nossos critérios de aceitação para este botão serão assim:

* **O manipulador de cliques é chamado ao clicar**
* **O manipulador de cliques é chamado ao pressionar a tecla Enter**
* **O manipulador de cliques é chamado ao pressionar a tecla Space​**
* **O manipulador de cliques não é chamado em nenhum outro pressionamento de tecla**
* **O elemento tem o atributo role="button"**

Então, onde e como podemos testar esses critérios? Sua resposta deve ser a mesma da última vez. Podemos escrever testes de unidade para isso usando nossa estrutura de teste e bibliotecas de escolha.

**Principais conclusões**

Nós cobrimos muitas informações aqui hoje. Se houver algo que você se lembre deste artigo, espero que sejam esses pontos:

Os verificadores de análise estática por si só não são ferramentas suficientes para garantir que seu aplicativo seja acessível.

É importante fazer testes exploratórios manuais para validar se os humanos podem realmente usar seu aplicativo com mouse, teclado e/ou leitor de tela.

Podemos tirar essas descobertas de nossos testes manuais, corrigir os bugs e escrever testes automatizados para evitar regressões.

**Obrigado por ler e obrigado por ser um defensor da acessibilidade.**