

GUIDE DU DÉVELOPPEUR RGAA 3

LICENCE D'UTILISATION



Ce document est la propriété du Secrétariat général à la modernisation de l'action publique français (SGMAP). Il est placé sous la [licence ouverte 1.0 ou ultérieure](#), équivalente à une licence *Creative Commons BY*. Pour indiquer la paternité, ajouter un lien vers la version originale du document disponible sur le [compte GitHub de la DInSIC](#).

SOMMAIRE

Introduction.....	4
À qui s'adresse ce guide ?.....	4
Mode d'emploi du guide.....	4
Ressources connexes et références.....	4
Fiche 1 : Ordre de tabulation et piège au clavier.....	5
Synthèse.....	5
Introduction - cas utilisateurs.....	5
Ordre de tabulation cohérent.....	5
Piège au clavier.....	7
Pour aller plus loin.....	7
Critères RGAA 3.....	7
Fiche 2 : Compatibilité et accès au clavier	8
Synthèse.....	8
Introduction - cas utilisateur.....	8
Compatibilité des composants et fonctionnalités JavaScript.....	8
Accès au clavier et à la souris.....	10
Pour aller plus loin.....	12
Critères RGAA 3.....	12
Fiche 3 : Changement de contexte et alerte non sollicitée	13
Synthèse.....	13
Introduction — cas utilisateurs.....	13
Changement de contexte.....	13
Alerte non sollicitée (AAA).....	15
Pour aller plus loin.....	15
Critères RGAA 3.....	15
ARIA.....	16
Introduction.....	16
Fiche 4 : Accessible Rich Internet Application (WAI ARIA)	17
Introduction.....	17
Généralités.....	17
Pour aller plus loin.....	19
Fiche 5 : Comment un lecteur d'écran sait-il de quoi il parle ?.....	20
Introduction - cas utilisateurs.....	20
Du contenu à l'utilisateur.....	20
Le lecteur d'écran, HTML et ARIA.....	24
Pour aller plus loin.....	25
Fiche 6 : Motif de conception ARIA.....	26
Introduction - cas utilisateur.....	26
Motif de conception ARIA.....	26
Exemple : une fenêtre de dialogue.....	26

Interactions au clavier.....	29
Que faire quand il n'existe pas de motif de conception.....	30
Garantir l'accessibilité des composants riches.....	31
Pour aller plus loin.....	31
Fiche 7 : Base de référence - Tests de restitution.....	32
Introduction - cas utilisateur.....	32
Base de référence.....	32
Méthodologie de test ARIA.....	34
Pour aller plus loin.....	36
Fiche 8 : Utiliser ARIA.....	37
Introduction - cas utilisateur.....	37
Règles d'utilisation de ARIA dans le HTML.....	37
Utilisation de rôles ou de propriétés spécifiques de l'API ARIA.....	40
Pour aller plus loin.....	42

INTRODUCTION

Ce guide du développeur vous est proposé dans le cadre des ressources accompagnant la prise en main de la version 3 du référentiel général d'accessibilité pour les administrations (RGAA 3).

Toutes les règles et tous les exemples d'implémentation donnés ici se réfèrent à la version 3 du RGAA.

Le RGAA 3 est composé d'un [document d'introduction](#), d'un [guide d'accompagnement](#) et d'un [référentiel technique](#). Cet ensemble de documents a une portée réglementaire puisqu'ils ont été rendus officiels par l'[arrêté du 29 avril 2015](#), lui-même venant préciser l'[article 47 de la loi 2005-102 du 11 février 2005](#) et l'[arrêté 2009-546 du 14 mai 2009](#).

Les ressources complémentaires sont des supports sans valeur réglementaire et visent à vous aider à rendre vos contenus numériques accessibles et conformes au RGAA 3.

À QUI S'ADRESSE CE GUIDE ?

Ce guide s'adresse aux développeurs d'application ou de composant web. Il se focalise sur les problématiques liées à l'utilisation de JavaScript et de l'API ARIA.

Une bonne maîtrise de HTML, CSS et JavaScript est requise pour la lecture de ce guide.

À CHACUN SON MÉTIER

Ne seront abordées dans ce guide que les problématiques liées à l'accessibilité des fonctionnalités ou composants développés par l'intermédiaire de JavaScript et ARIA. Les autres problématiques sont prises en charge par les autres guides comme le [guide de l'intégrateur](#).

MODE D'EMPLOI DU GUIDE

Le guide a été pensé comme une série de fiches pratiques sur le même modèle pour en faciliter l'utilisation. Chaque fiche étant indépendante, vous pouvez les lire en fonction de vos besoins, sans avoir à suivre un ordre particulier.

Vous retrouverez dans certaines fiches :

- Synthèse : les principaux points à retenir ;
- Correspondances RGAA 3 : les correspondances avec les critères du référentiel technique du RGAA 3.

RESSOURCES CONNEXES ET RÉFÉRENCES

- [Guide de l'intégrateur RGAA 3](#)
- [Méthodologie de test RGAA 3](#)
- [Guide « Défauts d'accessibilité : Impacts sur les utilisateurs »](#)
- [Comprendre les WCAG : description de cas utilisateurs et d'impacts par type de handicap](#)
- [Liste des techniques et échecs WCAG 2.0 \(ressource en anglais\)](#)

FICHE 1 : ORDRE DE TABULATION ET PIÈGE AU CLAVIER

SYNTHÈSE

- S'assurer que l'ordre dans lequel les éléments sont accédés à la tabulation reste cohérent lorsqu'on manipule le focus avec JavaScript
- Vérifier impérativement qu'il n'existe aucun piège au clavier et que l'utilisateur peut toujours accéder à l'élément suivant ou précédent avec la touche de tabulation

INTRODUCTION - CAS UTILISATEURS

Les personnes aveugles, utilisateurs de lecteurs d'écran, accèdent aux contenus de manière séquentielle : c'est-à-dire dans l'ordre dans lequel les contenus sont proposés dans le code généré de la page. Les personnes handicapées moteur, qui ne peuvent pas utiliser un système de pointage, vont parcourir les contenus interactifs de la même manière en utilisant essentiellement la tabulation.

Dans les deux cas, il est donc important que l'ordre de tabulation reste cohérent, particulièrement lorsque les contenus sont traités avec JavaScript. Exemples : l'ouverture d'une fenêtre modale, la mise à jour d'un contenu dans la page ou la simple gestion d'un menu de navigation déroulant.

Il faut aussi s'assurer qu'il n'existe aucun « piège au clavier » qui capture l'utilisateur dans une zone ou un composant de la page sans qu'il puisse en sortir.

ORDRE DE TABULATION COHÉRENT

Cela recouvre deux cas : l'ordre de parcours des contenus et la gestion de la tabulation dans un composant riche ou dans des contenus contrôlés par JavaScript.

Un ordre de tabulation cohérent ne signifie pas que l'ordre de parcours des contenus doit correspondre à l'ordre naturel de lecture (de gauche à droite et de bas en haut). Les documents et les applications Web peuvent être structurés dans un ordre particulier, différent de l'ordre de lecture naturel, particulièrement dans le cas d'application complexe. De même, l'ordre de parcours des contenus peut être adapté aux capacités d'un périphérique comme un téléphone portable par exemple.

Dans tous les cas, quelle que soit la structure de la page ou de l'application, l'utilisateur doit pouvoir accéder à tous les contenus de manière logique et efficace.

QUELQUES POINTS DE VIGILANCE

Menu de navigation

Dans le cas d'un menu déroulant, après ouverture d'un niveau de ce menu, la tabulation doit permettre d'accéder immédiatement au sous-menu. De fait, si le menu et ses sous-menus sont mal structurés, par exemple avec des listes successives au lieu de listes imbriquées, certains utilisateurs pourraient être dans l'incapacité d'atteindre les sous-menus à la tabulation.

Affichage et masquage de contenus.

Dans le cas d'une fonctionnalité permettant d'afficher un contenu masqué, par exemple un formulaire de connexion ou une zone de recherche, il est logique que la prochaine tabulation permette à l'utilisateur d'interagir avec le contenu affiché. Si dans le code, le contenu affiché est situé immédiatement après le lien ou le bouton permettant de l'afficher, l'ordre de tabulation sera naturellement satisfaisant, quelle que soit la position de la zone affichée dans la page. Dans le cas inverse, il sera nécessaire de transférer le focus de tabulation sur la zone affichée afin de restaurer un ordre de tabulation cohérent.

Dans certains cas, comme pour une fenêtre modale, ce parcours de tabulation devra correspondre à un modèle précis (cf. plus bas).

Zone de mise à jour

Comme pour les zones affichées ou masquées, lorsqu'une fonctionnalité JavaScript met à jour une zone de la page, par exemple une zone d'actualité, il est logique que la prochaine tabulation permette à l'utilisateur de parcourir la zone mise à jour.

Attention : ce n'est pas forcément un cas systématique !

Pour un panier mis à jour suite à la sélection d'un produit, il serait particulièrement laborieux pour l'utilisateur de devoir passer par le panier à chaque mise à jour. Un autre dispositif (les « live region » ARIA) permet de prendre ces cas en charge pour optimiser l'expérience de l'utilisateur. Cette problématique sera abordée dans le chapitre ARIA.

CAS DES COMPOSANTS RICHES BASÉS SUR UN MOTIF DE CONCEPTION ARIA



Lorsque le contenu est un composant riche (une fenêtre de dialogue (`dialog`), un système d'onglets (`tabpanel`), ...) l'API ARIA définit des « motifs de conception » qui décrivent précisément le comportement de la tabulation. Celui-ci peut être très différent du comportement dans des contenus textuels.

Par exemple, dans un système d'onglets, l'accès aux différents onglets ne sera pas pris en charge par la tabulation, mais par les flèches de direction, à l'image de la manipulation de ce type de composant dans les interfaces logicielles. Le principe de ces parcours de tabulation

sera présenté dans le chapitre traitant de la compatibilité des composants riches avec les technologies d'assistance.

PIÈGE AU CLAVIER

Il y a un « piège au clavier » lorsque l'utilisateur ne peut pas quitter sa position actuelle. Autrement dit, l'utilisateur ne peut atteindre ni l'élément focusable suivant, ni l'élément focusable précédent avec la touche tabulation. Dans cette situation, s'il ne navigue qu'avec la touche tabulation, sa seule échappatoire sera de fermer son navigateur.

Ce problème vient très souvent d'une mauvaise conception de la fonctionnalité ou du composant contrôlé qui se révèle être utilisable uniquement à la souris.

Chaque fonctionnalité développée qui met en jeu le focus doit être vérifiée au clavier pour éviter ce genre de catastrophe pour l'utilisateur.

Note importante : dans certains navigateurs (Firefox par exemple), l'insertion d'une balise object peut provoquer un piège au clavier. Il s'agit d'un bug et ce n'est pas considéré comme une non-conformité, du point de vue du RGAA.

POUR ALLER PLUS LOIN

CRITÈRES DE SUCCÈS WCAG

- [2.1.1](#)
- [2.1.2](#)
- [2.4.3](#)

TECHNIQUES ET ÉCHECS WCAG

- [G21](#)
- [G59](#)
- [H4](#)
- [H91](#)
- [F10](#)
- [F44](#)
- [F85](#)
- [SRC26](#)
- [SRC27](#)
- [SRC37](#)
- [C27](#)

CRITÈRES RGAA 3

- 12.13 [A]
- 12.14 [A]

FICHE 2 : COMPATIBILITÉ ET ACCÈS AU CLAVIER

SYNTHÈSE

- S'assurer que les composants sont compatibles avec l'accessibilité
- S'assurer que les composants et fonctionnalités sont accessibles et utilisables au clavier

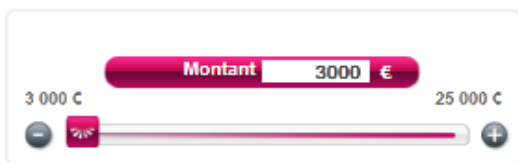
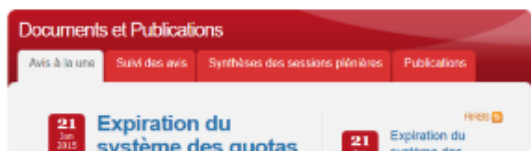
INTRODUCTION - CAS UTILISATEUR

De plus en plus, les pages web se retrouvent enrichies de composants applicatifs complexes : fenêtre de dialogue, menu, système d'onglets, slider... Ces derniers sont destinés à créer de véritables applications web contrôlées par JavaScript. Pour certains utilisateurs, plus particulièrement les aveugles qui se servent d'un lecteur d'écran et les handicapés moteurs qui ne peuvent pas employer de souris, ces nouveaux composants peuvent poser des problèmes complexes.

L'API ARIA a été spécifiquement conçue pour rendre ces composants accessibles et utilisables par tous. Elle propose notamment des « motifs de conception » qui décrivent très précisément l'implémentation et le comportement au clavier que doivent adopter ces composants.

Lorsqu'il existe un « motif de conception », une grande partie du travail consiste à vérifier que celui-ci est scrupuleusement respecté. Lorsqu'il n'en existe pas, il faut s'assurer que le composant ou la fonctionnalité est parfaitement accessible à tous.

COMPATIBILITÉ DES COMPOSANTS ET FONCTIONNALITÉS JAVASCRIPT



S'agissant des composants développés avec JavaScript et des fonctionnalités associées, deux cas de figure peuvent se présenter.

COMPOSANT CORRESPONDANT À UN « MOTIF DE CONCEPTION » ARIA

Imaginons l'implémentation d'une fenêtre de dialogue modale. Cela correspond au motif de conception dialog. Le motif de conception est assez simple à décrire :

- le composant doit posséder un rôle `role="dialog"` ;
- le composant doit être identifié avec une propriété `aria-label` ou `aria-labelledby` ;
- lorsque le composant est activé, le focus doit être transmis au premier élément interactif (qui peut prendre le focus) de la fenêtre ;
- tant que le composant est actif, la navigation par la tabulation doit être circonscrite à l'intérieur du composant et ne doit jamais atteindre les autres éléments interactifs de la page ;
- la navigation par tabulation doit être circulaire dans le composant : ce qui veut dire que lorsque le dernier élément est atteint, la tabulation suivante active le premier et inversement ;
- la méthode de fermeture, généralement un bouton, doit être activable avec la touche ESC ;
- lorsque le composant est fermé, le focus doit être rendu au composant ayant permis de l'ouvrir.

Vous pouvez trouver les détails de ce motif de conception dans la documentation ARIA : [Dialog \(Modal\)](#)¹.

Dans ce cas de figure le motif de conception est connu. Il faut vérifier que l'implémentation ARIA (présence et pertinence des propriétés ARIA requises) et le comportement au clavier sont conformes à ce qui est prévu.

Pour le comportement au clavier, qui peut être particulièrement complexe, le RGAA réduit l'exigence à un certain nombre de touches de base : TAB, ESC, ESPACE, ENTRÉE et les touches de direction HAUT, BAS, DROITE, GAUCHE.

Après s'être assuré de la conformité du motif de conception, il faut tester que la restitution est correcte avec les lecteurs d'écran en utilisant une base de référence de test.

La base de référence est un jeu de combinaisons associant un lecteur d'écran, un navigateur et un système d'exploitation représentatifs des usages sur le terrain pour la majorité des utilisateurs.

À titre d'exemple, voici la base de référence la plus utilisée dans les cas communs, elle couvre environ 80% des situations :

1. NVDA + Firefox + Windows
2. JAWS + IE + Windows
3. Voice Over + Safari + MacOS

D'autres combinaisons sont disponibles et vous pouvez ajouter des combinaisons particulières en fonction des contraintes d'utilisation de votre application. La description de la base de référence est disponible dans le RGAA : [base de référence](#)². Vous noterez que des contraintes supplémentaires complètent les combinaisons.

1 https://www.w3.org/TR/wai-aria-practices/#dialog_modal

2 <http://references.modernisation.gouv.fr/referentiel-technique-0#5-base-de-rfrence>

Les tests sur la base de référence sont indispensables.

Au-delà du motif de conception ARIA, le composant peut être affecté d'autres comportements, enrichi de propriétés ARIA complémentaires ou interagir avec d'autres composants de la page, ce qui pourrait perturber la restitution.

Une fois l'ensemble de ces tests effectués, vous pourrez considérer que le composant est « compatible avec l'accessibilité ».

Ces tests, qui peuvent sembler très contraignants, sont facilités par l'utilisation de bibliothèques de composants, prêts à l'emploi, qui implémentent déjà les motifs de conception ARIA.

Dans le cadre de ses ressources, le RGAA propose une étude très complète de plusieurs bibliothèques communément utilisées avec, si nécessaire, les corrections à effectuer : [Tutoriel "composants d'interface JavaScript"](#).

CAS D'UN COMPOSANT NON CONFORME

Dans le cas où il n'est pas possible de rendre un composant conforme, souvent du fait de limitations de la bibliothèque utilisée, vous devrez proposer une alternative pertinente qui permette à l'utilisateur d'accéder aux mêmes contenus et à des fonctionnalités similaires.

Cette alternative peut être un autre composant conforme ou une méthode qui ne requiert pas JavaScript pour fonctionner. Par exemple : un champ de saisie à la place d'un datepicker ou même un lien vers une version de la page fonctionnelle sans les composants en cause.

ACCÈS AU CLAVIER ET À LA SOURIS

La dernière chose à vérifier est l'accessibilité au clavier de tous les composants implémentés dans votre page.

Lorsque le composant correspond à un motif de conception, l'accessibilité au clavier est prise en charge par le motif de conception lui-même.

En revanche, quand ce n'est pas le cas, il faut vérifier que les éléments déclenchant une action sont accessibles avec la touche de tabulation et activables avec la touche ENTRÉE au moins.

TECHNIQUE POUR RENDRE UN ÉLÉMENT ACCESSIBLE AU CLAVIER

Lorsqu'un élément ne peut pas nativement recevoir le focus de tabulation (ce qui le rend inutilisable au clavier) il est possible de forcer ce comportement avec l'attribut `tabindex="0"`.

Cependant attention, si cela suffit pour le rendre accessible au clavier, cela peut rester insuffisant, notamment si cet élément se comporte comme un bouton HTML.

Dans ce cas, le motif de conception `button` devra vraisemblablement être utilisé. Ce motif prévoit par exemple que le composant soit activable avec la touche ENTRÉE et la touche ESPACE.

Une autre solution qui doit être privilégiée est d'employer l'élément `<button>` qui vous déchargera de toute la gestion au clavier.

CAS DES LIENS (A HREF OU A) UTILISÉS COMME BOUTON

Une très mauvaise habitude de développement est d'utiliser un lien `` ou une ancre a dépourvue d'attribut href pour simuler un bouton d'action.

Cette pratique pose deux problèmes :

- dans le cas de l'utilisation d'une ancre, **l'élément est inutilisable au clavier** puisqu'une balise a sans attribut href ne peut pas recevoir le focus à moins d'y ajouter un attribut `tabindex="0"` ;
- qu'il s'agisse d'une ancre ou d'un lien, le lecteur d'écran annoncera cet élément comme étant un lien. L'utilisateur s'attendra donc à atteindre une autre page/ressource ou un autre contenu dans la page, mais certainement pas à une ouverture de fenêtre modale par exemple.

Vous devez impérativement réserver l'utilisation des liens à l'affichage d'une ressource extérieure, une autre page, une fonctionnalité de mise à jour dans la page ou un lien vers une ancre.

Dans tous les autres cas, il faut utiliser un bouton, c'est-à-dire un élément `<button>` ou le motif de conception `button`.

Ceci est très important pour que l'utilisateur, qui ne voit pas la page, puisse faire la différence, en restitution, entre l'affichage d'une autre page et l'affichage d'une zone dans la page (ouverture d'une fenêtre modale par exemple).

Le motif de conception `button` est très simple :

- le composant doit avoir un rôle `button` ;
- le composant doit avoir un attribut `tabindex="0"` ;
- le composant doit être activable avec la touche ENTRÉE et ESPACE ;

Il peut permettre de réparer rapidement une simulation de bouton via un lien (ou tout autre élément) en prenant en charge le comportement du bouton avec JavaScript :

```
<a href="#" role="button" >faux bouton</a>
<a role="button" tabindex="0">faux bouton</a>

```

L'idéal étant d'utiliser des éléments natifs comme `<button>` ou `<input type="button">` par exemple.

POUR ALLER PLUS LOIN

CRITÈRES DE SUCCÈS WCAG

- [1.1.1](#)
- [4.1.2](#)
- [1.3.1](#)
- [2.1.1](#)
- [2.1.3](#)
- [2.4.7](#)

TECHNIQUES ET ÉCHECS WCAG

- [G10](#)
- [G90](#)
- [G135](#)
- [G136](#)
- [G202](#)
- [ARIA4](#)
- [ARIA5](#)
- [ARIA18](#)
- [ARIA19](#)
- [SRC2](#)
- [SRC20](#)
- [SRC21](#)
- [SRC29](#)
- [SRC35](#)
- [F15](#)
- [F19](#)
- [F20](#)
- [F42](#)
- [F54](#)
- [F55](#)
- [F59](#)
- [F79](#)

CRITÈRES RGAA 3

- 7.1 [A]
- 7.2 [A]
- 7.3 [A]

FICHE 3 : CHANGEMENT DE CONTEXTE ET ALERTE NON SOLLICITÉE

SYNTHÈSE

- S'assurer que l'utilisateur est averti et peut contrôler les changements de contexte
- S'assurer que les alertes non sollicitées sont contrôlables par l'utilisateur

INTRODUCTION — CAS UTILISATEURS

Un changement de contexte est un changement dans la page risquant d'être ignoré ou incompris par un utilisateur qui ne peut pas voir la page dans sa globalité. C'est le cas notamment des aveugles qui utilisent un lecteur d'écran et des malvoyants qui naviguent avec une loupe d'écran.

Exemple : la mise à jour de champs de formulaire dynamique ou un changement de contenu dynamique de la page.

Afin de leur permettre de comprendre ces changements, il faut qu'ils en soient préalablement prévenus ou que ces changements résultent d'une action explicite de leur part.

Parallèlement, des handicapés mentaux ou certains utilisateurs ayant des troubles de l'attention peuvent avoir des problèmes à interpréter correctement les fenêtres d'alerte quand ils ne les ont pas sollicitées. Dans ce cas, il faut leur offrir le moyen de les contrôler, notamment en les désactivant.

CHANGEMENT DE CONTEXTE

Les changements de contexte sont nombreux sur une page ou une application web. On considère qu'il y a un changement de contexte dans les cas suivants :

- changement d'agent utilisateur, par exemple sur un lien de téléchargement ;
- changement d'espace de restitution, par exemple l'affichage d'une nouvelle page ;
- changement de focus, par exemple le fait de transférer le focus d'un endroit à un autre ;
- changement de contenu qui modifie le sens de la page ou d'un élément, par exemple une mise à jour dynamique de contenu.

Ces quatre cas génériques recouvrent des situations, généralement prises en charge par des critères spécifiques dans le référentiel RGAA : comme l'obligation d'indiquer le type de fichier dans un lien de téléchargement, l'obligation de rendre les liens explicites ou encore l'obligation de préserver un ordre de tabulation cohérent.

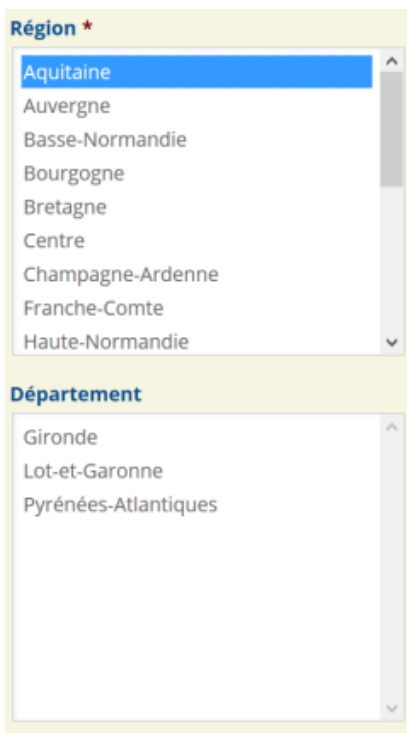
Toutefois, il peut y avoir des cas qui échappent aux critères spécifiques du RGAA.

Dans ce cas, en présence d'un changement de contexte avéré, il faut s'assurer que l'utilisateur est prévenu du changement de contexte par l'un des moyens suivants :

- un texte explicatif avant le changement de contexte ;
- l'utilisation d'un bouton pertinent pour initier le changement de contexte ;

- l'utilisation d'un lien pertinent pour initier le changement de contexte.

EXEMPLE : MISE À JOUR DYNAMIQUE DE CHAMPS DE FORMULAIRE



The image shows a form with two dropdown menus. The first menu, labeled 'Région *', is open and displays a list of French regions: Aquitaine, Auvergne, Basse-Normandie, Bourgogne, Bretagne, Centre, Champagne-Ardenne, Franche-Comte, and Haute-Normandie. The second menu, labeled 'Département', is also open and displays a list of departments: Gironde, Lot-et-Garonne, and Pyrénées-Atlantiques. This illustrates how the second menu's content is dynamically updated based on the selection in the first menu.

Imaginons un formulaire qui contiendrait deux champs : un champ pour la région et un champ désactivé pour le département. Lorsqu'une région est sélectionnée, le champ Département est activé et mis à jour avec la liste des départements concernés.

Il y a potentiellement une situation de changement de contexte.

Deux moyens sont possibles pour adapter ce comportement :

- interfacier un bouton entre les deux champs de telle sorte que ce soit l'utilisateur qui provoque l'activation et la mise à jour du champ région ;
- utiliser un `fieldset` et une légende pertinente comme « choisir la région puis le département ».

Dans les deux cas, l'utilisateur sera bien prévenu de ce qui se passera et sera en mesure d'utiliser le mécanisme de manière satisfaisante.

EXEMPLE : LISTE DE NAVIGATION

Imaginons une liste `select` affichant une liste des pages d'un document. Lorsque l'utilisateur sélectionne un item de la liste, la page est mise à jour.

Cette situation de changement de contexte très commune est problématique. Si l'utilisateur ne voit pas le changement opéré, il peut simplement l'ignorer et croire qu'il ne s'est rien passé alors que la page a été changée.

Le seul moyen d'adapter ce dispositif est de s'en tenir au comportement normal d'une liste de formulaire `select` : sélectionner un item dans la liste et activer un bouton.

Attention : Cette remarque ne concerne que les listes `select`. Si vous utilisez un composant permettant d'afficher/masquer une liste constituée de liens par exemple, alors il n'y aura rien à faire. L'utilisateur comprendra que la page va être mise à jour s'il active un des liens.

ALERTE NON SOLLICITÉE (AAA)

Il est habituel de vouloir alerter l'utilisateur lors d'actions importantes ou tout au long d'un processus. Lorsque ces informations sont faites sous forme d'alertes, elles peuvent être perturbantes, désorienter l'utilisateur ou rendre ses saisies très laborieuses.

Il faut donc toujours donner la possibilité à l'utilisateur de désactiver et de réactiver ces alertes.

Font exceptions les cas d'urgence d'un événement ou d'une situation soudaine et imprévue qui exigent une action immédiate afin de préserver la santé, la sécurité ou la propriété. Dans ces cas, il n'est pas requis que les alertes soient contrôlables par l'utilisateur.

POUR ALLER PLUS LOIN

CRITÈRES DE SUCCÈS WCAG

- [3.2.1](#)
- [3.2.2](#)
- [3.2.3](#)

TECHNIQUES ET ÉCHECS WCAG

- [G13](#)
- [G76](#)
- [G80](#)
- [G107](#)
- [H32](#)
- [H84](#)
- [SRC19](#)
- [F9](#)
- [F22](#)
- [F36](#)
- [F37](#)
- [F41](#)

CRITÈRES RGAA 3

- 7.4 [A]
- 7.5 [AAA]

ARIA

INTRODUCTION

ARIA est une API destinée à rendre les composants et les applications développées avec JavaScript accessibles à tous.

Cette partie présente les points essentiels de l'API, son fonctionnement en liaison avec les technologies d'assistance et les règles de bonnes pratiques à respecter.

Néanmoins, ARIA est un sujet complexe, basé sur des principes d'implémentation simples, qui nécessite un véritable apprentissage et une veille constante du fait de l'évolution et de la disparité du support de ses propriétés dans les navigateurs et les technologies d'assistance.

Seront abordés les bases essentielles de l'API, la manière dont les technologies d'assistance peuvent en bénéficier, les notions de « motif de conception » et de « base de référence », les règles d'utilisation d'ARIA dans les applications et les documents Web.

Cette partie s'articule sur 5 fiches :

- Fiche 4 : Accessible Rich Internet Application (WAI ARIA)
- Fiche 5 : Comment un lecteur d'écran sait-il de quoi il parle ?
- Fiche 6 : Motif de conception ARIA
- Fiche 7 : Base de référence - Tests de restitution
- Fiche 8 : Utiliser ARIA

FICHE 4 : ACCESSIBLE RICH INTERNET APPLICATION (WAI ARIA)

INTRODUCTION



Les composants développés avec JavaScript sont inconnus du navigateur qui ne peut pas communiquer les propriétés et les états de ces éléments aux technologies d'assistance par l'intermédiaire des API d'accessibilité. Par ailleurs, si ces composants doivent interagir avec l'utilisateur, le navigateur ne connaît pas non plus les comportements à implémenter, au clavier notamment.

L'API ARIA a pour rôle de permettre aux développeurs d'informer le navigateur de l'ensemble des propriétés et des états des composants développés avec JavaScript et, avec des motifs de conceptions rigoureux, de proposer des modèles d'utilisation au clavier.

ARIA propose un dispositif simple pour indiquer les grandes structures du document et quelques propriétés particulières pour rendre accessibles les zones de la page mises à jour dynamiquement.

GÉNÉRALITÉS

RÔLE, ÉTAT ET PROPRIÉTÉ

L'API décrit trois types de propriétés : des rôles, des états et des propriétés.

Les rôles permettent d'indiquer le type de composant ou d'élément. ARIA propose une soixantaine de rôles. Certains sont spécifiques à des modèles de composants, d'autres équivalent à des éléments HTML.

Quelques exemples : `dialog` pour une fenêtre de dialogue, `slider` pour un potentiomètre de saisie, `tabpanel` pour un système d'onglets, `heading` équivalent à `hx`, `list` et `listitem` équivalent aux balises `ul` et `li`, `button` équivalent à la balise `button`.

Les états et les propriétés permettent de décrire le composant et d'informer de ses changements d'état. Les états et les propriétés sont toujours préfixés par `aria-`.

Quelques exemples de propriétés : `aria-valuemin` pour indiquer la valeur minimum d'un composant, `aria-level` pour indiquer le niveau d'un élément dans un groupe (par exemple avec `heading` ou `listitem`), `aria-required` pour indiquer si une saisie de valeur est obligatoire.

Quelques exemples d'états : `aria-selected` pour indiquer si un composant est sélectionné, `aria-expanded` pour indiquer si un composant est ouvert ou fermé, `aria-invalid` pour indiquer si une valeur est fausse.

Enfin, certains rôles, états ou propriétés sont spécifiques à des besoins particuliers : comme la description des relations entre les différentes parties d'un composant, la description des grandes structures d'un document ou d'une interface, la gestion des mises à jour dynamiques, la nature du contenu (application ou document), le statut de restitution.

Quelques exemples : `aria-haspopup` signale qu'un composant contrôle un menu contextuel ou un sous-menu, `main` signale la partie principale d'un document, `aria-live` contrôle la restitution d'un contenu dynamique, `application` signale que le contenu se comporte comme une application, `aria-hidden` signale que la restitution est inhibée.

Implémentation

Le principe d'implémentation est très trivial : ARIA s'implémente comme un jeu d'attributs dont les valeurs qui le nécessitent vont être contrôlées par JavaScript.

Par exemple, une case à cocher s'implémentera de la manière suivante :

```
<span role="checkbox" tabindex="0" aria-checked="false" aria-  
labelledby="labelcheckbox">  
<span id="labelcheckbox">Oui</span>
```

OBJECTIFS DE L'API ARIA

Définir les composants d'interface et de structure

ARIA va donc permettre de définir l'intégralité d'un composant : son rôle, ses états et propriétés, sa structure et son comportement.

Informé de l'état et des propriétés des composants d'interface

En s'appuyant sur l'implémentation d'ARIA par le développeur, le navigateur va transmettre l'ensemble des informations aux APIs d'accessibilité du système.

Attention ! ARIA est une API purement descriptive. Ce n'est pas parce que vous allez déclarer qu'un `span` a un rôle `checkbox` que le navigateur va implémenter le comportement d'une case à cocher. **L'intégralité du comportement, les événements, la mise à jour des états et des propriétés, le comportement au clavier incombent au développeur via JavaScript.**

Informé et gérer les mises à jour de contenus dynamiques

Il existe une exception : la gestion des zones mises à jour dynamiquement, ce qu'on appelle une *live region*. Une région contrôlée par `aria-live` ou une fenêtre d'alerte contrôlée par le rôle `alert` sont prises en charge par le navigateur qui informera l'API d'accessibilité des changements opérés sur ces zones.

Rendre les composants utilisables au clavier

Tout composant doit être opérable au clavier. Pour chaque composant qu'il définit, ARIA propose donc un comportement au clavier normalisé de telle sorte que les composants interagissent de manière uniforme en reprenant les concepts à l'œuvre dans les logiciels notamment.

Cette partie de la conception d'un composant avec ARIA peut devenir complexe car elle met en jeu des interactions au clavier particulièrement riches.

POUR ALLER PLUS LOIN

- [L'API](#) (role, state et propriétés)
- [Les Best practices](#) (Design pattern, Comportement et gestion clavier)
- [Using WAI-ARIA in HTML](#)

FICHE 5 : COMMENT UN LECTEUR D'ÉCRAN SAIT-IL DE QUOI IL PARLE ?

INTRODUCTION - CAS UTILISATEURS

Un lecteur d'écran est un logiciel particulièrement complexe. Son rôle est de restituer vocalement l'ensemble des contenus affichés à l'écran. Il restitue le contenu textuel, mais également les éléments interactifs ainsi que l'ensemble des changements opérés sur ces derniers.

Outre ce rôle qui permet à un aveugle d'accéder à un ordinateur, il a pour charge de lui fournir les moyens d'interagir avec les contenus et les logiciels.

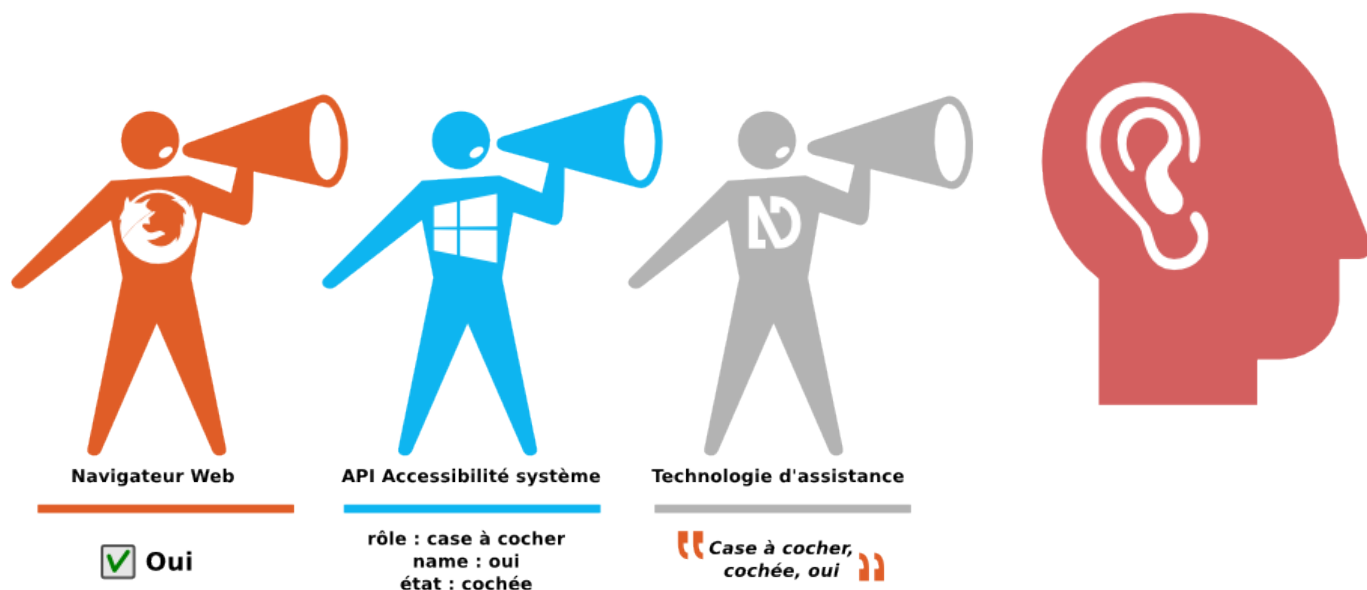
Pour ce faire, il s'appuie essentiellement sur les API d'accessibilité du système, au travers desquelles transite l'ensemble des informations nécessaires et sur les logiciels eux-mêmes auxquels il va transférer des commandes utilisateurs.

La caractéristique d'un lecteur d'écran est d'être totalement dépendant des informations fournies par les API d'accessibilité, par les logiciels et par les producteurs des contenus et des fonctionnalités.

Les développeurs ont donc une responsabilité essentielle car ils sont le dernier maillon entre l'utilisateur et les contenus et fonctionnalités qu'ils proposent.

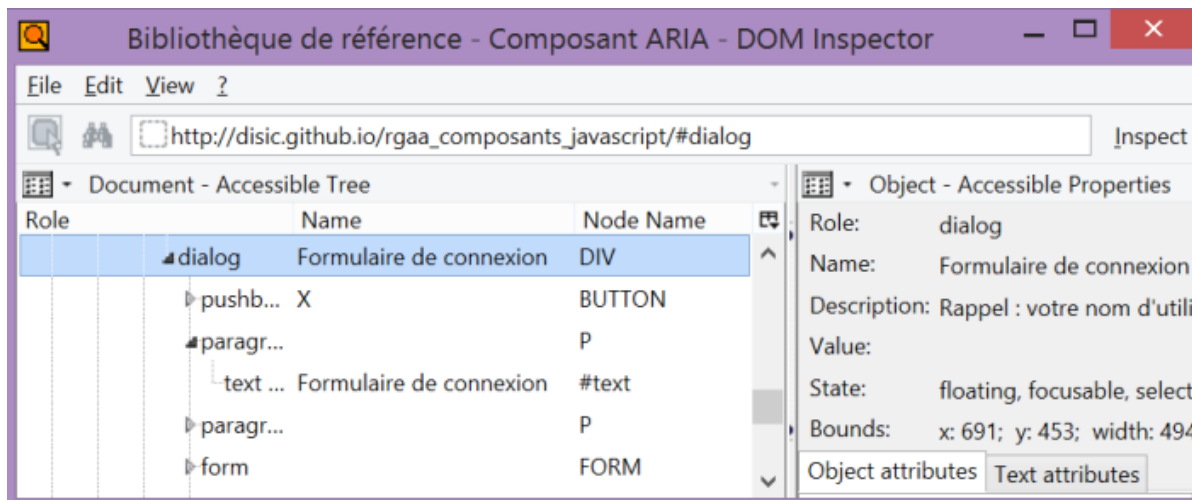
Dans ce système complexe, l'API ARIA va jouer un rôle crucial. C'est à travers elle que les développeurs vont fournir l'ensemble des informations sans lesquelles le contenu et ses fonctionnalités seront simplement inexploitable par l'utilisateur.

DU CONTENU À L'UTILISATEUR



Pictogramme Olivier Guin - CC BY 3.0

NOTIONS DE BASE SIMPLIFIÉES



Comme indiqué plus haut, les systèmes d'exploitation fournissent des API pour centraliser toutes les informations sur les « objets » en cours d'utilisation. Ces informations, qui peuvent être très nombreuses, sont disponibles à tout moment et mises à jour dès que nécessaire.

Une partie de ces API est consacrée à l'accessibilité en utilisant des propriétés communes ou dédiées, par exemple le type, le nom, la valeur et les états.

Les logiciels ont pour charge de transmettre à l'API système les propriétés caractéristiques des objets qu'ils contrôlent.

Les navigateurs Web vont fournir en complément une arborescence d'objets, appelée l'« accessible tree », mis à jour à chaque modification en s'appuyant sur le DOM et en ajoutant pour chaque objet les propriétés nécessaires à l'accessibilité.

Les lecteurs d'écran vont s'appuyer sur cette arborescence d'objets et sur les propriétés des éléments, fournis par le système, pour restituer les contenus.

« Case à cocher, non cochée, oui »

Pour comprendre de manière simplifiée comment fonctionne un lecteur d'écran nous allons prendre l'exemple d'une case à cocher HTML :

```
<input type="checkbox" id="checkbox" />  
<label for="checkbox">Oui</label>
```

Oui

```
LegacyIAccessible.ChildId: 0
LegacyIAccessible.DefaultAction: "check"
LegacyIAccessible.Description: ""
LegacyIAccessible.Help: ""
LegacyIAccessible.KeyboardShortcut: ""
LegacyIAccessible.Name: "Oui"
LegacyIAccessible.Role: case à cocher (0x2C)
LegacyIAccessible.State: défilant,pouvant être actif (0x102000)
LegacyIAccessible.Value: ""
```

Oui

```
LegacyIAccessible.ChildId: 0
LegacyIAccessible.DefaultAction: "uncheck"
LegacyIAccessible.Description: ""
LegacyIAccessible.Help: ""
LegacyIAccessible.KeyboardShortcut: ""
LegacyIAccessible.Name: "Oui"
LegacyIAccessible.Role: case à cocher (0x2C)
LegacyIAccessible.State: coché,défilant,pouvant être actif (0x102010)
LegacyIAccessible.Value: ""
```

Étudions quelles sont les informations mises à

disposition par le navigateur et l'API MSAA sous Windows (le principe est le même pour OSX ou Linux, seules les API diffèrent).

Le composant « case à cocher » (`type="checkbox"`) est connu du navigateur. Ce dernier transmet donc à l'API les informations suivantes : un rôle « checkbox », un nom « oui », un état « défilant pouvant être actif » et une valeur nulle. Une case à cocher n'a pas de valeur, juste un état. À noter que l'état correspond ici à des caractéristiques héritées, communes à certains types d'éléments interactifs.

Lorsque l'utilisateur coche la case, le navigateur transmet l'information de mise à jour de l'état qui va devenir « coché, défilant pouvant être actif ». « coché » signale qu'il est coché.

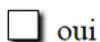
Le lecteur d'écran est donc en mesure de restituer le composant. NVDA vocalisera « case à cocher non cochée oui » pour l'état initial et « oui, case à cocher, cochée » lorsque l'utilisateur actionne le composant.

Chaque lecteur d'écran vocalise comme il le souhaite les composants. Il n'y a pas de règle en la matière, mais une origine commune : les informations transmises par le navigateur aux API système.

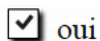
« Cliquable »

Regardons ce qui se passe maintenant avec le même composant simulé avec JavaScript et CSS :

```
<span></span> oui
```



LegacyIAccessible.ChildId: 0
LegacyIAccessible.DefaultAction: "click"
LegacyIAccessible.Description: ""
LegacyIAccessible.Help: ""
LegacyIAccessible.KeyboardShortcut: ""
LegacyIAccessible.Name: ""
LegacyIAccessible.State: normal (0x0)
LegacyIAccessible.Value: ""



LegacyIAccessible.ChildId: 0
LegacyIAccessible.DefaultAction: "click"
LegacyIAccessible.Description: ""
LegacyIAccessible.Help: ""
LegacyIAccessible.KeyboardShortcut: ""
LegacyIAccessible.Name: ""
LegacyIAccessible.State: normal (0x0)
LegacyIAccessible.Value: ""

Pour le navigateur, un span vide, même avec une magnifique image d'une coche en propriété de fond, ne représente rien. Y compris lorsque l'image est modifiée par JavaScript suite à un clic sur le span. Il va donc transmettre la seule information dont il dispose : un événement (click) est implémenté sur l'objet.

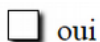
C'est ce qu'indique l'API qui ne renvoie aucun type, un nom nul et un état « normal » qui signifie simplement que le navigateur n'a rien à dire sur cet élément.

La case à cocher simulée va être vocalisée par NVDA « cliquable », seule information disponible et, de fait, inutile. C'est là le quotidien des utilisateurs aveugles sur le Web.

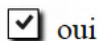
« Case à cocher, cochée, oui »

Reprenons l'exemple précédent. Avec ARIA, nous allons pouvoir transmettre au lecteur d'écran par le navigateur et le système les informations nécessaires à la restitution de la case à cocher simulée.

```
<span role="checkbox" aria-checked="false" tabindex="0" aria-labelledby="foo"></span>  
<span id="foo">Oui</span>
```



LegacyIAccessible.ChildId: 0
LegacyIAccessible.DefaultAction: "click"
LegacyIAccessible.Description: ""
LegacyIAccessible.Help: ""
LegacyIAccessible.KeyboardShortcut: ""
LegacyIAccessible.Name: "oui"
LegacyIAccessible.Role: case à cocher (0x2C)
LegacyIAccessible.State: défilant,pouvant être actif (0x102000)
LegacyIAccessible.Value: ""



LegacyIAccessible.ChildId: 0
LegacyIAccessible.DefaultAction: "click"
LegacyIAccessible.Description: ""
LegacyIAccessible.Help: ""
LegacyIAccessible.KeyboardShortcut: ""
LegacyIAccessible.Name: "oui"
LegacyIAccessible.Role: case à cocher (0x2C)
LegacyIAccessible.State: coché,défilant,pouvant être actif (0x102010)
LegacyIAccessible.Value: ""

role="checkbox" signale au navigateur le type d'élément. aria-checked transmet l'état. aria-labelledby permet de lier le passage de texte « Oui » pour nommer le composant. Enfin, tabindex="0" permet de rendre le composant opérable au clavier.

Avec JavaScript, il suffira de mettre à jour l'état `aria-checked` et de gérer le comportement au clavier en ajoutant à l'événement `click` un équivalent au clavier pour les touches ENTRÉE et ESPACE, car une case à cocher peut s'activer au clavier avec ces deux touches.

Regardons maintenant le résultat du point de vue de l'API système. Le composant retrouve les mêmes informations qu'une case à cocher HTML, le rôle « case à cocher », un nom « Oui » et les deux états possibles « défilant pouvant être actif » et « coché, défilant pouvant être actif ».

NVDA restituera cet élément de la manière suivante : « case à cocher, non cochée, oui » et « case à cocher, cochée, oui » de manière quasi identique à l'original HTML. L'utilisateur, lui, n'y verra aucune différence.

LE LECTEUR D'ÉCRAN, HTML ET ARIA

Il en va de même pour tout ce qui est affiché à l'écran, les paragraphes, les titres, les listes, les liens et ainsi de suite du plus simple au plus complexe. Lorsque ces objets correspondent à un type et un comportement connu, ce qui est le cas de tous les objets définis par la spécification HTML, la chaîne d'informations qui va du contenu à l'utilisateur est robuste et les restitutions sont cohérentes.

En revanche, lorsque le développeur ne fait pas correctement son travail, par exemple en utilisant des `span` stylés pour faire des titres, la chaîne d'informations est rompue. La restitution est défectueuse et les fonctionnalités d'interaction que proposent le lecteur d'écran sont inopérables.

De même, lorsque le développeur propose des composants « maison », le navigateur est dans l'incapacité de transmettre les informations nécessaires au lecteur d'écran.

Seul le recours à ARIA permettra, en respectant strictement les définitions de rôles, d'états, de propriétés et les directives de comportement qui y sont associées, de rendre ces composants inconnus accessibles à tous.

NOTE AU SUJET DU NOM ACCESSIBLE.

Il existe toujours plusieurs manières de donner un nom à un composant.

Cela peut-être : une méthode native fournie par HTML par exemple la balise `label` pour un champ de formulaire, une méthode complémentaire également fournie par HTML par exemple avec l'attribut `title` et des méthodes de surcharges fournies par l'API ARIA par exemple avec les propriétés `aria-label` ou `aria-labelledby`.

Le navigateur et le lecteur d'écran doivent donc effectuer un calcul pour savoir, lorsque plusieurs de ces méthodes sont utilisées, qu'elle est celle qui détermine le **nom accessible** d'un composant.

WCAG fournit un document qui détaille, pour chaque élément, la méthode de calcul du **nom accessible**. Cette méthode de calcul respecte toujours le même modèle : les propriétés ARIA lorsqu'elles sont présentes remplacent la méthode native HTML qui remplace la méthode complémentaire.

Il est important de bien comprendre ce principe pour gérer correctement les différentes méthodes permettant de nommer un composant en HTML, mais également de vérifier, lorsque plusieurs d'entre elles sont utilisées, qu'elles le sont de manière cohérente.

Le référentiel RGAA comporte à ce sujet quelques tests qui permettent de s'assurer qu'une propriété ARIA mal utilisée ne vient pas remplacer le nom légitime d'un composant.

LE LECTEUR D'ÉCRAN ET L'UTILISATEUR

Afin de pouvoir faire interagir l'utilisateur avec le contenu, le lecteur d'écran va s'interfacer entre l'utilisateur et le navigateur de la manière suivante : lorsque l'utilisateur va utiliser une des commandes mises à sa disposition, le lecteur d'écran va commander le navigateur par exemple en déplaçant le focus sur tel ou tel élément. À l'inverse, lorsque l'utilisateur actionne un composant connu, par exemple le parcours d'une liste `select`, le lecteur d'écran va se contenter d'écouter ce que renvoie le navigateur en retour des actions de l'utilisateur.

Nous verrons par la suite que ce mode d'interaction peut avoir des conséquences importantes sur la conception de composants complexes.

POUR ALLER PLUS LOIN

- [Le modèle de conception checkbox](#) (en anglais)
- [Accessible Name and Description calculation](#)

FICHE 6 : MOTIF DE CONCEPTION ARIA

INTRODUCTION - CAS UTILISATEUR

Afin de pouvoir accompagner les développeurs dans la conception de composants riches, ARIA propose des motifs de conception (Design pattern en Anglais) pour environ 39 composants prédéterminés.

À ces motifs de conception, s'ajoute un certain nombre de principes de conception par exemple pour la gestion du focus, etc.

Présentés dans un guide de développement, ces motifs de conception sont essentiels pour proposer des composants qui fonctionnent de manière homogène et prédictive pour les utilisateurs.

Les motifs de conception, particulièrement pour ce qui concerne la gestion au clavier, sont inspirés par le fonctionnement des interfaces logicielles.

MOTIF DE CONCEPTION ARIA

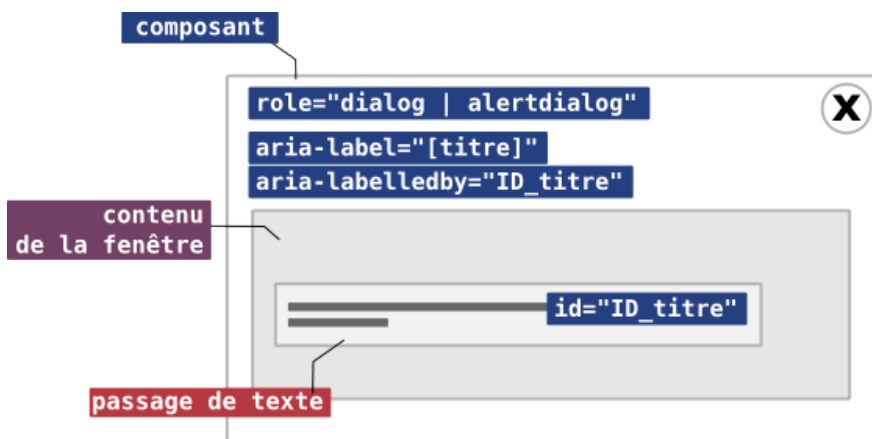
Un motif de conception est composé de deux volets : l'un va décrire la structure du composant, les rôles, propriétés et états à utiliser, le second décrit très précisément le comportement attendu du composant et l'ensemble des interactions au clavier qui doivent être implémentées.

Ces motifs de conception sont particulièrement importants car ils permettent aux navigateurs et aux technologies d'assistance de disposer de composants dont le comportement est prédictif. Tout ceci permet de s'assurer que les restitutions, et surtout la manipulation de ces composants par les utilisateurs, seront cohérentes.

Les motifs de conception sont liés à des rôles. Cela implique que l'utilisation de ces rôles respecte le modèle proposé.

EXEMPLE : UNE FENÊTRE DE DIALOGUE

IMPLÉMENTATION DES RÔLES, PROPRIÉTÉ ET ÉTATS ARIA



Prenons par exemple un motif

de conception simple lié au rôle dialog. Ce rôle permet de proposer un contenu sous la forme d'une fenêtre de dialogue qui peut être modale ou non modale.

La fenêtre de dialogue doit être présentée sous la forme d'un élément HTML unique et doit, comme tout composant interactif, être dotée d'un nom.

Le motif de conception dialog propose deux méthodes pour le nom. Elles s'appuient sur deux propriétés génériques que l'on retrouvera sur la majorité des composants :

- `aria-label` qui permet de définir l'étiquette d'un objet ;
- `aria-labelledby` qui permet d'utiliser un passage de texte identifié comme étiquette du composant.

Voici deux manières d'implémenter le modèle proposé.

```
<div role="dialog" id="mymodale" aria-label="Connection">
[... ]
</div>
```

```
<div role="dialog" id="mymodale" aria-labelledby="foo">
<h1 id="foo">Titre</h1>
[... ]
</div>
```

Du point de vue de l'implémentation des propriétés ARIA, il n'y a rien d'autre à faire. Le rôle `dialog` signalera au lecteur d'écran qu'il s'agit d'une fenêtre de dialogue. NVDA le restituera par « Connexion dialogue » et JAWS par « Connexion, boîte de dialogue » par exemple.

COMPORTEMENT ET GESTION AU CLAVIER

Le comportement basique d'une fenêtre de dialogue est d'être affichée et fermée, rien de plus. Néanmoins, se pose immédiatement un problème : que faire de la position active de l'utilisateur (la position du focus) lorsqu'il va déclencher l'apparition de la fenêtre ?

Le motif de conception nous indique que le focus doit être donné sur le premier élément interactif capable de le recevoir dans la fenêtre. C'est extrêmement important puisque c'est cette prise de focus qui, en activant le composant, va déclencher la cascade de mise à jour jusqu'au lecteur d'écran pour qu'il puisse le restituer.

Notre modèle va donc être complété en insérant un bouton de fermeture et transférant le focus sur ce bouton à l'ouverture de la fenêtre.

```
<div role="dialog" id="mymodale" aria-labelledby="foo" tabindex="-1">
  <h1 id="foo">Titre</h1>
  <button id="close">Fermer</button>
  [...]
</div>
```

À l'ouverture de la fenêtre, le focus sera transmis au bouton de fermeture, ce qui va déclencher la vocalisation de l'ensemble du contenu de la fenêtre par les lecteurs d'écran.

La question se pose maintenant de savoir quoi faire à la fermeture. Le modèle indique qu'il faudra simplement redonner le focus à l'élément qui a permis d'ouvrir la fenêtre. À la souris naturellement, mais également au clavier.

ARIA demande deux méthodes pour fermer une fenêtre :

- en activant le bouton de fermeture ;
- en appuyant sur la touche ESC.

Cela implique donc d'ajouter aux méthodes gérant les clics de souris :

- une méthode pour la touche ENTRÉE ;
- et une autre pour la touche ESC.

Note : il peut être utile de doter certains composants d'un attribut `tabindex="-1"` ce qui aura pour effet de pouvoir les manipuler avec la méthode JavaScript `element.focus()`. Cela sera utile pour surveiller que le focus est toujours capturé par la fenêtre.

MODALE OU NON MODALE

Une fenêtre modale est utilisée généralement lorsque l'on fait dépendre une action d'une opération réalisée par l'utilisateur. Ce sera le cas par exemple d'un formulaire de saisie de login ou encore d'une demande de confirmation affiché sous la forme d'une fenêtre.

Généralement, chaque fois qu'un contenu va être proposé sous la forme d'une fenêtre superposée au contenu principal, il s'agira d'une fenêtre modale.

Mais il peut y avoir des cas où un contenu proposé sous la forme d'une fenêtre ne requiert aucune action de l'utilisateur (par exemple un texte d'aide affiché lors de la saisie d'un formulaire). Bien que d'autres modèles de conception puissent être utilisés (comme le rôle `tooltip` ou `alertcode`), si c'est le rôle `dialog` ou `alertdialog` qui est utilisé, on parlera d'une **fenêtre non modale**.

La différence de comportement est simple à comprendre. Une fenêtre modale doit impérativement « capturer » l'utilisateur qui ne doit pouvoir réagir qu'avec le contenu de la fenêtre tant qu'elle est affichée. Dans une fenêtre non modale, l'utilisateur est libre de poursuivre sa consultation indépendamment de l'affichage ou non de la fenêtre.

Comportement d'une fenêtre modale

Le dernier point concerne le fonctionnement de la fenêtre. Une fenêtre modale doit capturer les actions de l'utilisateur tant qu'elle est affichée, ce qui implique qu'il ne puisse pas naviguer ailleurs que dans la fenêtre. De plus, il faudra modifier le comportement de la tabulation. Tant que la fenêtre est active, l'utilisateur ne doit tabuler que dans les éléments tabulables de la fenêtre.

Lorsque la fenêtre n'est pas modale en revanche il peut en sortir pour suivre un parcours de tabulation habituel.

RÉSUMÉ

Ce modèle est assez simple et peut se résumer de la manière suivante :

1. la fenêtre doit être implémentée sous la forme d'un élément englobant doté du rôle `dialog` ;
2. la fenêtre doit posséder un nom en utilisant `aria-label` ou `aria-labelledby` ;
3. à l'ouverture, le focus doit être transféré sur le premier élément qui peut recevoir le focus de tabulation ;
4. à la fermeture, le focus doit être rendu à l'élément ayant permis d'ouvrir la fenêtre ;

5. la fenêtre doit pouvoir se fermer avec la touche ESC au moins ;
6. si la fenêtre est une fenêtre modale :
 - le focus doit être maintenu dans la modale ;
 - la tabulation doit être circonscrite aux seuls éléments de la modale.

Conséquences sur le développement de composants riches

Les motifs de conception sont un outil très puissant qui vise essentiellement deux buts : uniformiser l'expérience utilisateur sur le Web pour l'utilisation des composants et développer de nouveaux modèles d'utilisation au clavier.

L'uniformisation du comportement des composants riches est un enjeu majeur du Web, particulièrement pour les applications. Il serait particulièrement compliqué pour les utilisateurs de manipuler ces composants s'ils devaient en réapprendre l'utilisation sur chaque nouveau site ou application.

L'enrichissement des interactions au clavier est tout aussi important dans le cas de composants complexes comme un système d'onglets, un calendrier ou un menu en accordéon. En effet la méthode basique de HTML, c'est-à-dire la tabulation, atteint très rapidement des limites qui rendraient l'utilisation de ces composants très laborieuse.

INTERACTIONS AU CLAVIER

Comme dit précédemment, une grande partie d'ARIA est inspirée par l'état de l'art du développement d'applications logicielles accessibles.

Le but visé est de tenter d'uniformiser l'expérience utilisateur afin que les composants complexes fonctionnent sensiblement de la même manière que les applications logicielles dont les utilisateurs ont l'habitude.

Les conséquences sur le développement sont particulièrement lourdes car cela nécessite d'implémenter de nombreux raccourcis clavier.

Choisir le tarif maximum recherché



Sur un slider, le curseur devra être manipulable avec les flèches de direction. Il devra également fournir des raccourcis clavier pour atteindre le minimum et le maximum et un raccourci pour augmenter ou diminuer la valeur rapidement en utilisant un intervalle.

Dans des composants encore plus complexes, comme un système d'onglets, c'est une dizaine de raccourcis clavier qu'il faudra implémenter.

Conscient des difficultés que cela peut représenter, le RGAA a réduit l'exigence aux seules interactions de base indispensables qui mettent en jeu les touches : TAB, ESC, ESPACE, ENTRÉE, et les touches de direction HAUT, BAS, DROITE, GAUCHE.

Cela ne veut pas dire qu'il ne faut pas implémenter les interactions au clavier riches proposées par les motifs de conception, au contraire.

L'implémentation des interactions au clavier change complètement la manière dont les utilisateurs vont naviguer et manipuler les composants. Par exemple, un principe générique est que les différentes parties d'un composant, quand elles ne sont pas constituées d'éléments interactifs, sont atteintes avec les touches de direction, ce qui sera le cas des panneaux d'un système d'onglets par exemple.

Ce bouleversement des interactions au clavier nécessite un temps d'appropriation par les utilisateurs qui poussent certains à vouloir ne pas les respecter et s'en tenir au développement de composants riches utilisables, en tout état de cause, avec la seule tabulation.

Il s'agit d'une grave erreur. Il est fondamental de respecter à la lettre ces modèles de navigation au clavier. Le RGAA impose, par ailleurs, que soit fournie une aide expliquant le fonctionnement des composants au clavier utilisés sur le site ou l'application Web.

QUE FAIRE QUAND IL N'EXISTE PAS DE MOTIF DE CONCEPTION

Bien que très riche, et encore en cours de développement pour la version 1.1, l'API ARIA ne peut pas couvrir tous les cas possibles.

Lorsque vous développez un composant qui n'est pas référencé par le guide de développement, il faut, par vous même, trouver les bonnes propriétés à utiliser en puisant dans l'API ou en vous inspirant d'exemples déjà en production.

Zones affichées ou masquées

C'est le cas par exemple des zones qui peuvent s'afficher ou se masquer avec une action de l'utilisateur.

Structurellement, cela peut se présenter comme ça :

```
<button>en savoir plus</button>
<div>
  [...]
</div>
```

Il faut alors déterminer les informations qui devront être remontées au lecteur d'écran pour lui permettre de restituer le fonctionnement correctement.

Ici, en l'absence de motif de conception, aucun rôle n'est à utiliser. En revanche, dans tous les cas, l'utilisateur aura besoin de savoir si la zone est affichée ou masquée et, éventuellement, de savoir de quelle zone il s'agit.

Il y a plusieurs manières de transmettre ces informations. La plus simple est d'utiliser la propriété `aria-expanded` qui indiquera que la zone est affichée ou masquée. Cette propriété peut être indifféremment positionnée sur la zone ou le bouton lui-même, mais il s'avère que l'implémenter sur le bouton semble plus efficace.

Il peut être intéressant également de signaler au lecteur d'écran la relation qui existe entre le bouton et la zone contrôlée. ARIA propose la propriété `aria-controls` à cet effet.

Voilà l'implémentation basique de ce genre de dispositif :

```
<button aria-expanded="false" aria-controls="foo">en savoir plus sur l'API  
ARIA</button>  
  <div id="foo">  
    [...]   
  </div>
```

Avec JavaScript, il suffira de changer la valeur d'état `aria-expanded` pour signaler le statut de la zone contrôlée. Avec NVDA, cela sera vocalisé par « En savoir plus sur l'API ARIA, bouton, réduit » et « développé » au moment de l'action.

GARANTIR L'ACCESSIBILITÉ DES COMPOSANTS RICHES

Mais cela ne vous garantit pas que ce composant soit réellement accessible, c'est-à-dire qu'un aveugle, utilisateur de lecteur d'écran, sera en mesure de le manipuler et d'en comprendre le fonctionnement.

Comme pour tous les composants, qu'ils soient ou non encadrés par un motif de conception ARIA, il va falloir le tester en restitution sur les lecteurs d'écran et avec les systèmes d'exploitation qui vont être utilisés en production.

C'est la seule manière de garantir que les composants développés avec JavaScript et ARIA seront réellement accessibles à tous.

POUR ALLER PLUS LOIN

Les ressources du RGAA proposent un ensemble de démonstration de composants ARIA avec les restitutions obtenues dans les lecteurs d'écran : [Ressource RGAA : Implémentation des composants ARIA](#)

RÉFÉRENCES

- [Le guide de développement ARIA](#) (en anglais)

FICHE 7 : BASE DE RÉFÉRENCE - TESTS DE RESTITUTION

INTRODUCTION - CAS UTILISATEUR

Si l'immense majorité des composants définis par la spécification HTML n'a pas besoin d'être testée, ce n'est pas le cas des composants développés avec JavaScript et ARIA.

Si le support d'ARIA par les navigateurs ne pose plus de problème, les technologies d'assistance n'offrent pas encore un support suffisamment robuste et l'on peut constater de grandes variations entre elles.

Au-delà de ce problème de support d'ARIA par les technologies d'assistance, le simple fait qu'un composant respecte strictement un motif de conception n'est pas une garantie suffisante quant à son accessibilité réelle. En effet les composants peuvent être enrichis de fonctionnalités particulières, agir dans le cadre d'une application et interagir avec d'autres composants.

Il est donc primordial que ces composants soient testés en situation réelle avec les technologies d'assistance des utilisateurs.

BASE DE RÉFÉRENCE



Il serait compliqué de tester un composant ou une application avec l'ensemble des technologies d'assistance sans peser très fortement sur la production des contenus web et des applications dont une des caractéristiques est la grande versatilité. Il faut publier vite, souvent. Une application Web n'est, en réalité, qu'une longue suite ininterrompue de commits de publication.

Par ailleurs, les technologies d'assistance ont des parts d'utilisation très diverses. Certaines d'entre elles sont confidentielles ou très spécifiques alors que d'autres couvrent une très large partie des utilisateurs.

Enfin, le support d'ARIA reste encore très inconstant. Certaines technologies d'assistances, particulièrement celles pour qui ARIA n'a qu'une utilité secondaire, sont en effet en retard. Obliger à ce que les composants développés avec JavaScript et ARIA soient compatibles avec

elles condamnerait le développement Web « accessible » à être très en retrait des technologies web modernes ou à recourir à des alternatives très coûteuses à produire, lorsque c'est possible.

Cela a amené le RGAA à définir la notion de « base de référence » qui consiste à définir des combinaisons associant des lecteurs d'écran, des navigateurs et des systèmes d'exploitation pour couvrir la part la plus importante des utilisateurs.

La base de référence est constituée des configurations (technologie d'assistance, système d'exploitation, navigateur) qui permettent de déclarer qu'un dispositif HTML5/ARIA est « compatible avec l'accessibilité », comme défini par WCAG 2.

Elle est établie par consensus à partir de la liste des technologies d'assistance dont l'usage est suffisamment répandu ou dans certains cas (par exemple pour macOS) lorsqu'elle est fournie de manière native et constitue le moyen privilégié d'accès à l'information et aux fonctionnalités.

La combinaison qui apparaît la plus satisfaisante est constituée de :

1. NVDA et Firefox sur Windows ;
2. JAWS et Firefox ou Internet Explorer9+ sur Windows ;
3. Voice Over et Safari sur OSX ;

La question des versions de lecteur d'écran retenues est importante et obéit aux règles suivantes :

- lorsque le lecteur d'écran est publié sous une licence gratuite, comme NVDA, c'est la version courante qui fait office de référence ;
- lorsque le lecteur d'écran est publié sous une licence payante, comme JAWS, c'est la version précédent la version courante qui fait office de référence.

RÈGLES COMPLÉMENTAIRES :

Un certain nombre de règles complémentaires sont associées à l'utilisation d'une base de référence :

1. L'ensemble des dispositifs HTML5/ARIA ou leurs alternatives doivent être pleinement fonctionnels, sur l'ensemble des pages du site, sans nécessiter de changement de technologie d'assistance en cours d'utilisation ;
2. Lorsque des alternatives à des dispositifs HTML5/ARIA sont proposées, elles ne doivent pas nécessiter la désactivation d'une technologie (par exemple JavaScript ou le plugin Flash), sauf s'il s'agit d'une fonctionnalité proposée par le site lui-même. Par exemple :
 - le site met à disposition une version alternative conforme pleinement fonctionnelle sans le recours aux technologies dont l'usage est non compatible avec l'accessibilité ;
 - le site met à disposition une fonctionnalité de remplacement des dispositifs HTML5/ARIA par des dispositifs alternatifs compatibles ;
3. un moyen est mis à disposition des utilisateurs de technologies d'assistance pour signaler les problèmes rencontrés et obtenir, via un dispositif de compensation, les informations qui seraient rendues indisponibles ;
4. si une déclaration de conformité est établie, elle doit comporter la liste des technologies d'assistance avec lesquelles les dispositifs HTML5/ARIA ont été testés et les résultats de

ces tests (par exemple « supporté », « non supporté », « supporté partiellement ») au moins.

CAS DES ENVIRONNEMENTS MAÎTRISÉS

Un environnement maîtrisé est constitué d'une plateforme de diffusion dont la maîtrise est complète. Cela signifie que les utilisateurs sont connus et leurs technologies contrôlées.

Par exemple, lorsque le site web est exclusivement diffusé dans un environnement GNU/Linux, les tests devront être réalisés uniquement sur les navigateurs et les technologies d'assistance utilisés sur cette plateforme. Cette base de référence se substitue à la base de référence utilisée en environnement non maîtrisé.

ÉLARGISSEMENT DE LA BASE DE RÉFÉRENCE

La base de référence n'est pas une fin en soi, mais juste un socle technique minimum qui assure que, dans la majorité des cas, les composants et les applications pourront être effectivement utilisés.

Elle peut et devrait être étendue à des technologies d'assistance dont l'usage est identifié et concerne effectivement les utilisateur des sites et applications.

Sont plus particulièrement visées les plateformes de diffusions mobiles, téléphones ou tablettes, dont la grande hétérogénéité et l'évolution constante n'ont pas permis de les prendre en compte. De même, des plateformes dont l'utilisation est moindre, comme GNU/LINUX devraient également, si c'est pertinent dans le contexte de diffusion, être prises en compte afin d'assurer une accessibilité la plus large possible.

MÉTHODOLOGIE DE TEST ARIA

La méthodologie de test suivante peut être appliquée :

1. vérifier si nécessaire la conformité du motif de conception utilisé ;
2. si possible, tester la restitution avec la base de référence choisie avant l'intégration du composant dans l'application ;
3. lors de l'intégration, vérifier avec des tests sur la base de référence choisie que le composant a une restitution cohérente et efficace.

Le test de restitution du composant avant son intégration dans l'application est toujours une bonne idée car cela permet, dans le cas où les tests en intégration sont défaillants, de cibler rapidement l'origine du problème : le composant lui-même ou des effets de bords de l'intégration et des interactions avec l'application ou d'autres composants en utilisation.

Note : Les modèles de conception ARIA sont actuellement en cours de mise à jour par les auteurs de l'API ARIA (passage de la version 1.0 à la version 1.1). Il faudra néanmoins un certain temps avant que les technologies d'assistance implémentent ces nouveaux modèles de conception.

Le RGAA propose dans ses ressources :

- Un [guide de tests des composants avec les lecteurs d'écran](#) de la base de référence ;
- une [grille de tests des motifs de conception](#) qui pourront vous guider dans la réalisation des tests.

TEST DE CONFORMITÉ DU MOTIF DE CONCEPTION UTILISÉ

Le RGAA propose dans ses ressources une plateforme de démonstration de 12 motifs de conception avec l'indication des restitutions obtenues et une grille de tests complète. Vous pouvez réutiliser la grille de tests pour vos propres besoins. **Attention : les composants proposés ne peuvent pas être utilisés en production. Ils sont destinés exclusivement à des tests et à servir de référence pour vos propres tests.**

TESTS DE RESTITUTION

Les tests de restitution doivent être effectués en utilisant les lecteurs d'écran de la base de référence, ce qui implique d'en apprendre le fonctionnement. Cette page vous propose un certain nombre de ressources.

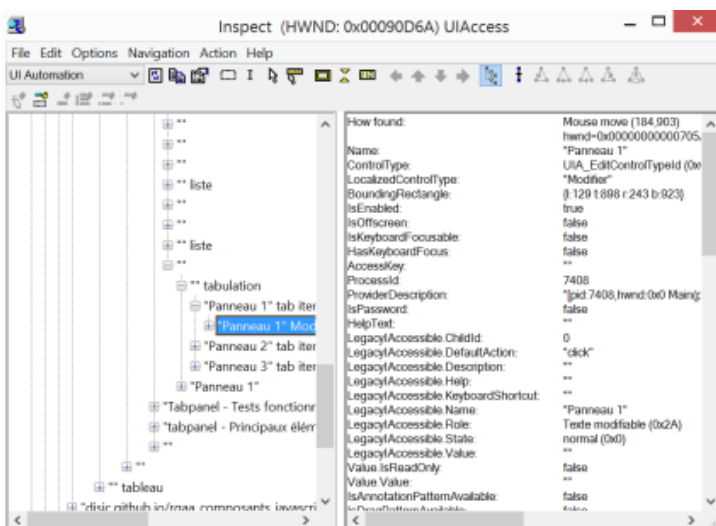
L'utilisation d'un lecteur d'écran demande un certain temps d'apprentissage. Si le test basique d'un composant ne devrait pas poser de problème, dans le cas de tests menés sur des applications entières, le recours à des prestataires spécialisés est fortement recommandé.

Soyez également prudent avec les tests réalisés avec des utilisateurs en situation réelle. Si mener des tests utilisateurs peut grandement améliorer l'ergonomie et l'utilisabilité des applications, ils peuvent échouer à remonter des erreurs d'implémentation.

OUTIL DE DÉBOGAGE

Lorsqu'un composant se montre défaillant, il est indispensable de pouvoir déterminer avec précision si l'ensemble des informations sont bien mises à disposition du lecteur d'écran. Pour le vérifier, vous pouvez utiliser des logiciels ou des plug-ins qui vous permettront d'explorer l'« accessible tree » ou les API d'accessibilité système.

Exemple : inspecteur d'objet du SDK de Windows



L'inspecteur d'objet vous permet de parcourir l'ensemble de l'arborescence système pour analyser les propriétés des objets telles qu'elles sont mises à disposition par l'API système.

Pour ce qui concerne l'accessibilité, les propriétés « LegacyIAccessible.[propriété] » sont celles qui vont nous intéresser.

Cet outil est très puissant. Il permet de choisir le mode d'interaction (tabulation ou pointeur), de scénariser des actions, d'enregistrer des logs.

Dans le cadre d'un test de composant, il peut permettre de vérifier si les propriétés d'états sont bien mises à jour par exemple.

Exemple : AViewer

Le logiciel AViewer proposé par la société The Paciello Group (TPG) poursuit le même but en ajoutant plusieurs interfaces comme IAccessible2 qui est l'API utilisée par Firefox par exemple.

Ce logiciel est tout aussi efficace que l'inspecteur d'objet de Windows, mais son fonctionnement est plus instable.

Autres logiciels ou plugins

Toutes les plateformes proposent un outil similaire, par exemple sur macOS, il s'agit de Accessibility Inspector et Accerciser pour Linux/Gnome.

Enfin, signalons également le plugin « Dom Inspector » qui propose un accès simple à l'accessible tree avec Firefox.

POUR ALLER PLUS LOIN

RÉFÉRENCES

- [RGAA - base de référence](#) ;
- [Le guide de développement ARIA \(en anglais\)](#).

RESSOURCES

- [Ressource RGAA : Implémentation des composants ARIA](#) ;
- [Basic screen reader commands for accessibility testing - TPG](#) ;
- [Guide de l'utilisateur - NVDA](#) ;
- [Keyboard Shortcuts for NVDA - WebAIM](#) ;
- [Keyboard Shortcuts for JAWS - WebAIM](#) ;
- [Using VoiceOver to Evaluate Web Accessibility - WeBAIM](#) ;
- [iOS VoiceOver Testing Techniques & Procedures for Absolute Beginners - Aidan Tierney](#) ;
- [Accessibility testing with Android Talkback - TPG](#) ;
- [iOS & Android Screen Reader Gesture Reference Cheatsheet - Interactive Accessibility](#).

OUTILS

- [Inspect - Microsoft](#) ;
- [Accessibility Viewer - TPG](#) ;
- [Accerciser - Gnome](#) ;
- [Testing for Accessibility on OS X - Apple](#) ;
- [DOM Inspector - Firefox](#)

FICHE 8 : UTILISER ARIA

INTRODUCTION - CAS UTILISATEUR

ARIA est une technologie simple et très puissante dont un des moyens est de modifier la sémantique des éléments. Cette faculté qui peut être délicate doit obéir à des règles d'utilisation afin de ne pas perturber les fonctionnalités proposées par les lecteurs d'écran.

ARIA propose quelques rôles et propriétés dont l'usage nécessite également de bien en comprendre les ressorts afin d'éviter de rendre les restitutions incompréhensibles ou inutilisables : c'est le cas par exemple des rôles application et document, ou encore de l'usage des « live region ».

L'ensemble de ces règles d'usages sont présentées en détail dans la note [Notes on Using ARIA in HTML](#) que tout développeur devrait avoir lu. Vous en trouverez ci-dessous les principales recommandations.

RÈGLES D'UTILISATION DE ARIA DANS LE HTML

PRIVILÉGIER LES ÉLÉMENTS HTML NATIFS

ARIA propose beaucoup de rôles dont l'objectif sémantique est équivalent à celui des éléments HTML natifs, par exemple `role="list"`, `role="img"`, etc.

Par exemple :

```
<span role="heading" aria-level="1">Titre de niveau 1</span>
```

Cette utilisation des rôles `heading` et `aria-level` transforme un simple `span` en un titre de niveau 1 « virtuel » qui requiert l'utilisation d'une technologie d'assistance compatible pour être correctement restitué.

Or, la première règle d'ARIA stipule qu'il faut privilégier l'utilisation d'élément HTML natif, sauf exception :

If you can use a native HTML element or attribute with the semantics and behaviour you require already built in, instead of re-purposing an element and adding an ARIA role, state or property to make it accessible, then do so.

Si vous pouvez utiliser des éléments HTML natifs ou des attributs dont la sémantique et le comportement recherchés sont déjà disponibles, plutôt que de redévelopper un élément et d'ajouter un rôle, un état ou une propriété ARIA pour le rendre accessible, faites-le.

Trois situations qui justifient le recours à ARIA peuvent être envisagées pour créer de toute pièce un élément HTML :

- si l'élément HTML n'est pas implémenté ou que son accessibilité est défaillante ;
- s'il existe des contraintes de présentation telles qu'elles sont irréalisables avec l'élément natif HTML ;
- si l'élément HTML n'existe simplement pas.

Une autre utilisation possible et qui peut justifier le recours à ARIA est de corriger un contenu HTML défaillant lorsqu'il n'est pas possible d'intervenir sur le HTML natif, même si cette utilisation ne devrait être faite qu'à titre provisoire. Par exemple, lors de l'utilisation d'un plugin qui génère du HTML et qu'on ne peut pas modifier.

PRÉSERVER LA SÉMANTIQUE NATIVE DES ÉLÉMENTS

Beaucoup de fonctionnalités d'exploration et de manipulation de contenu offertes par les technologies d'assistance sont fondées sur la sémantique native des éléments HTML. La modifier, via ARIA, sans précautions, c'est prendre le risque de rendre ces fonctionnalités essentielles inopérables.

Ainsi, la seconde règle d'ARIA préconise :

Do not change native semantics, unless you really have to.

Ne changez pas la sémantique native d'un élément sauf en dernier recours.

Par exemple :

```
<h1 role="button">titre bouton</h1>
```

Ce titre transformé en bouton sera inatteignable avec le raccourci de navigation de titre en titre fourni par une technologie d'assistance. On devrait plutôt écrire :

```
<h1><button>titre bouton</button></h1>
```

En revanche, il est possible dans certains cas d'utiliser des éléments HTML natifs comme méthode de « fallback ». Par exemple, utiliser une liste HTML `ul li` pour implémenter un rôle `tree` dont le but est de créer une arborescence interactive.

RENDRE OPÉRABLES AU CLAVIER LES ÉLÉMENTS INTERACTIFS CONTRÔLÉS PAR ARIA

Comme expliqué dans les chapitres précédents, ARIA se contente de décrire les éléments et les navigateurs n'implémentent pas tous les comportements ad hoc au clavier.

C'est pourquoi la troisième règle d'ARIA préconise :

All interactive ARIA controls must be usable with the keyboard.

Tous les contrôles interactifs ARIA doivent être utilisables au clavier.

Cette règle est implémentée comme une exigence formelle dans le RGAA via l'obligation, d'une part, de respecter les motifs de conception ARIA et, d'autre part, de rendre les composants interactifs utilisables au clavier lorsqu'il n'existe pas de tel motif de conception.

NE PAS INHIBER LA SÉMANTIQUE OU LA RESTITUTION DES ÉLÉMENTS FOCUSABLES VISIBLES

ARIA propose un rôle `presentation` qui supprime la sémantique restituée.

Par exemple :

```
<table role="presentation">
  <tr>
    <td></td>
    <td></td>
  </tr>
</table>
```

À cause du `role="presentation"`, ce tableau n'existera plus dans l'« accessible tree ». Cela peut-être utile pour supprimer la sémantique d'éléments utilisés comme fallback dans des composants complexes. En revanche, son utilisation sur des éléments focusables visibles va poser un problème car certains utilisateurs risquent de prendre le focus sur « rien », c'est à dire un élément inconnu en restitution.

De même, ARIA propose une propriété `aria-hidden` qui signale que l'élément ne doit pas être restitué. Cette propriété est généralement utilisée en conjonction avec des directives CSS d'affichage pour gérer les zones masquées d'un composant complexe.

Cette propriété peut également servir à empêcher la restitution d'un élément visible redondant ou inutile, par exemple dans certains cas de simulation de présentation de formulaire sous la forme de tableaux possédant des en-têtes visibles mais inutiles puisque chaque champ possède déjà un label valide.

En revanche, comme pour l'utilisation du rôle `presentation`, l'utilisation de la propriété `aria-hidden` sur un élément focusable visible peut poser des problèmes.

Ainsi, la quatrième règle préconise :

Do not use `role="presentation"` or `aria-hidden="true"` on a visible focusable element.

Ne pas utiliser `role="presentation"` ou `aria-hidden="true"` sur un élément focusable visible.

Cette règle devrait être **toujours** respectée.

DONNER UN NOM ACCESSIBLE (ACCESSIBLE NAME) À TOUS LES COMPOSANTS CONTRÔLÉS PAR ARIA

Cette cinquième règle de bon sens est incontournable. Elle préconise :

All interactive elements must have an accessible name.

Tout élément interactif doit avoir un nom accessible.

Comme expliqué dans les chapitres précédents, fournir un accessible name, par exemple via les propriétés `aria-labelledby` ou `aria-label`, est la seule manière de garantir que l'élément en cours d'utilisation sera correctement identifié par les utilisateurs.

Cette règle est encadrée de manière rigoureuse par le RGAA via l'obligation de respecter les motifs de conception, l'obligation de tester les restitutions, et certains critères pour des cas plus spécifiques.

UTILISATION DE RÔLES OU DE PROPRIÉTÉS SPÉCIFIQUES DE L'API ARIA

APPLICATION VS. DOCUMENT

ARIA propose deux rôles qui vont impacter de manière fondamentale l'utilisation des contenus par les aveugles utilisateurs de lecteurs d'écran : les rôles `application` et `document`.

Pour comprendre l'utilisation de ces deux rôles, il faut décrire brièvement la manière dont les lecteurs d'écran s'interfacent entre l'utilisateur et le navigateur.

Les lecteurs d'écrans sous Windows, JAWS, NVDA, Window-Eyes, proposent deux modes d'interaction différents aux utilisateurs : le « mode navigation » et le « mode formulaire ».

Le mode navigation permet à l'utilisateur de naviguer dans le contenu, stocké sous la forme d'une copie en mémoire de la page, via des raccourcis clavier ou des fonctionnalités de navigation spécifiques.

Naturellement, dans ce contexte, le lecteur d'écran a besoin de récupérer l'ensemble des actions au clavier afin de les traiter par ses propres moyens. Dans ce mode d'interaction, le navigateur est totalement passif.

Mais dès que l'élément en cours de consultation doit être géré par le navigateur car il correspond à un type connu, le lecteur d'écran va donner le contrôle des touches du clavier au navigateur.

Mode navigation

Par exemple, la lecture ligne à ligne d'un contenu textuel ne requiert aucune action du navigateur et va donc être traitée par le lecteur d'écran de manière autonome. Il en va de même du parcours de titre en titre et de liste en liste, ou du parcours des éléments d'une liste ou des cellules d'un tableau par exemple.

Ce mode est appelé le « mode navigation ».

Mode formulaire

En revanche, si l'utilisateur rencontre dans sa navigation un élément interactif comme un lien ou un champ de formulaire, le lecteur d'écran va cesser de traiter le clavier : il va passer le relais au navigateur qui va alors actionner les contenus directement.

Par exemple, le navigateur va afficher le texte dans une boîte de saisie, afficher la page référencée dans un lien, parcourir la liste des options d'un élément `select`, positionner le focus sur le prochain élément focusable avec la touche tabulation, etc.

Dans ce contexte, le lecteur d'écran se contente de restituer les informations renvoyées par le navigateur et mises à jour via les APIs d'accessibilité et l'« accessible tree ».

Ce mode est appelé, par généralisation, le « mode formulaire ».

Ces changements de mode sont automatiques et transparents pour l'utilisateur qui en est informé par un signal sonore par exemple.

Avec ARIA, l'utilisation du rôle `application` va signaler au lecteur d'écran de passer en « mode formulaire », et inversement, le rôle `document` va signaler au lecteur d'écran de passer en « mode navigation ».

Cela a une conséquence majeure pour le développement : en effet, déclencher le « mode formulaire » signifie que **c'est au développeur de prendre en charge l'intégralité de la gestion au clavier** et de vérifier que les informations de mises à jour sont correctement transmises au lecteur d'écran.

Cela sera systématiquement le cas pour tous les composants d'interface qui bénéficient d'un motif de conception nécessitant des interactions au clavier, comme les rôles `button`, `tabpanel`, `dialog`, `menu` par exemple. D'où l'importance de respecter strictement les interactions au clavier définies par les motifs de conception.

UTILISATION DU RÔLE APPLICATION

Vous ne devriez jamais utiliser le rôle `application` dans le cas général, c'est à dire l'utilisation de composants riches dans une application destinée à afficher des contenus.

La seule utilisation imaginable du rôle `application` serait le développement d'une véritable application web qui fonctionnerait exactement comme une application logicielle habituelle - ce qui est encore assez rare.

UTILISATION DU RÔLE DOCUMENT

Habituellement, une page web ayant un statut de document par défaut, vous n'aurez pas à utiliser le rôle `document`. Cela étant dit, ce rôle peut rendre quelques services.

Imaginons par exemple un système d'onglets qui affiche du contenu dans ses panneaux.

Du fait du rôle `application` par défaut, l'utilisateur ne pourra pas bénéficier de ses fonctionnalités de navigation dans les contenus des panneaux.

Si le panneau n'est constitué que d'éléments interactifs, par exemple des champs de formulaire ou des liens, même accompagnés d'un court texte de présentation, le rôle `application` restera efficace et simulera parfaitement une situation normale d'utilisation, comme la saisie d'un formulaire par exemple.

En revanche, si les panneaux contiennent des textes particulièrement riches, des titres, des paragraphes, des listes, etc., l'utilisation des fonctionnalités de navigation peut être utile.

Dans ce cas, l'implémentation d'un rôle `document` signalera au lecteur d'écran qu'il peut reprendre la main tant que le panneau est actif.

Par exemple :

```
<div id="pan0" aria-hidden="false" aria-expanded="true" aria-labelledby="tab0"
role="tabpanel">
  <div role="document">
    [...]
  </div>
</div>
```

Le raisonnement est identique avec une fenêtre modale dont on souhaite donner le contrôle du contenu à l'utilisateur, pour que ce dernier puisse utiliser les touches de navigation dans le contenu par exemple.

LES « LIVE REGION »

ARIA possède une propriété particulière `aria-live` qui va permettre de faire remonter automatiquement les changements de contenus opérés via AJAX par exemple.

C'est la seule propriété dont le comportement est pris en charge par le navigateur.

Le fonctionnement est le suivant : dès que le navigateur détecte un changement de node (texte, élément ou attribut) dans le contenu spécifié, il remonte la modification au lecteur d'écran qui va vocaliser les contenus selon le paramétrage de la « live region ».

Les paramètres

- `aria-live="polite"` : le contenu est vocalisé dès que l'utilisateur est disponible, c'est à dire dès qu'il ne réalise plus aucune action. `aria-live="assertive"` : le contenu est vocalisé immédiatement.
- `aria-atomic="true"` : toute la zone est vocalisée. `aria-atomic="false"` : seules les modifications sont vocalisées.
- `aria-relevant` sert à préciser les modifications qui seront vocalisées : `addition` pour les ajouts de contenus, `removal` pour les suppressions, `all` pour vocaliser tous les types de contenus, `text` pour ne vocaliser que les contenus de type texte.

À noter qu'ARIA propose également des rôles dont le comportement est similaire, comme les rôles `alert`, `log` ou `progressbar` par exemple.

Bien utilisée, cette propriété est une solution robuste et efficace pour gérer les zones de mise à jour dynamique, par exemple un panier d'achats affiché dans la page.

Mais attention : puisque la zone contrôlée par `aria-live` va être vocalisée, il convient de l'utiliser avec précaution dès que cette zone va contenir beaucoup de contenus.

Par exemple, imaginons un moteur de recherche qui afficherait ses résultats dans la même page via AJAX. L'utilisation de `aria-live` sur la zone mise à jour risque d'être particulièrement lourde et contre-indiquée. Préférez une prise de focus sur la zone mise à jour en résultat de la fonctionnalité de recherche afin de simuler un rechargement de page classique par exemple.

De même, l'utilisation simultanée de plusieurs « live region » dans une même page ou une même application pourrait donner des résultats difficilement interprétables.

Comme tout ce qui concerne ARIA, seuls des tests de restitution en situation réelle peuvent valider l'utilisation cohérente des « live region ».

POUR ALLER PLUS LOIN

RÉFÉRENCES

- [Notes on Using ARIA in HTML - W3C](#)

RESSOURCES

Pour en savoir plus sur les modes d'interaction des lecteurs d'écran, vous pouvez lire l'excellent article de Leonie Watson (TPG) : [Understanding screen reader interaction modes](#).