ACCESSIBLE AUTOCOMPLETE

What is Autocomplete?

“Autocomplete” is a software function that **provides relevant suggestions** based on input by the user.

For this presentation, we’re going to focus on accessible autocomplete **associated with search**.

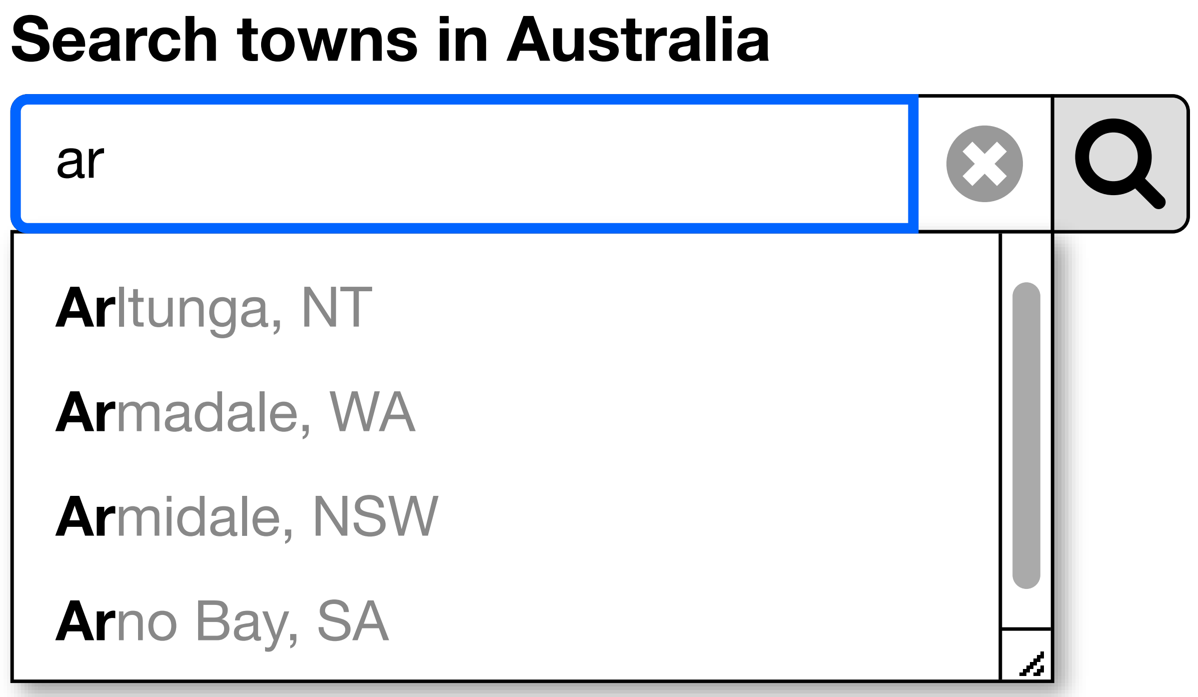


Diagram showing autosuggest search component

The aim is to provide some **information to consider** if you’re thinking about building an accessible autocomplete search.

User Experience

Before diving into accessibility,
we’ll look at some **common UX patterns**
associated with autocomplete search.

1. There should be clear wording or visual indicators to **describe the purpose of the search**.

For example, are users searching across the entire site, an aspect of the site, or is it **a specific search function**?

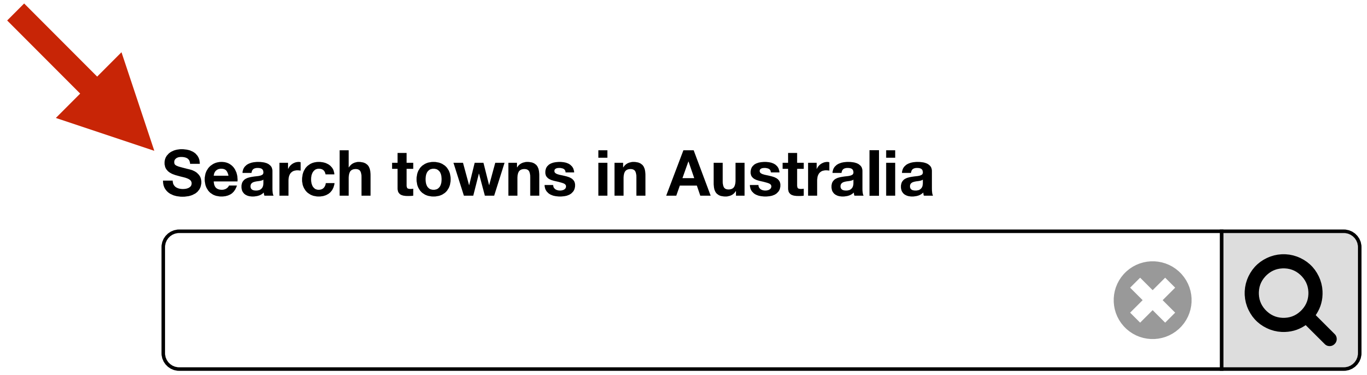


Diagram showing highlighted label “Search towns in Australia”

2. If additional instructions are required, they **should be located in close proximity** to the field.

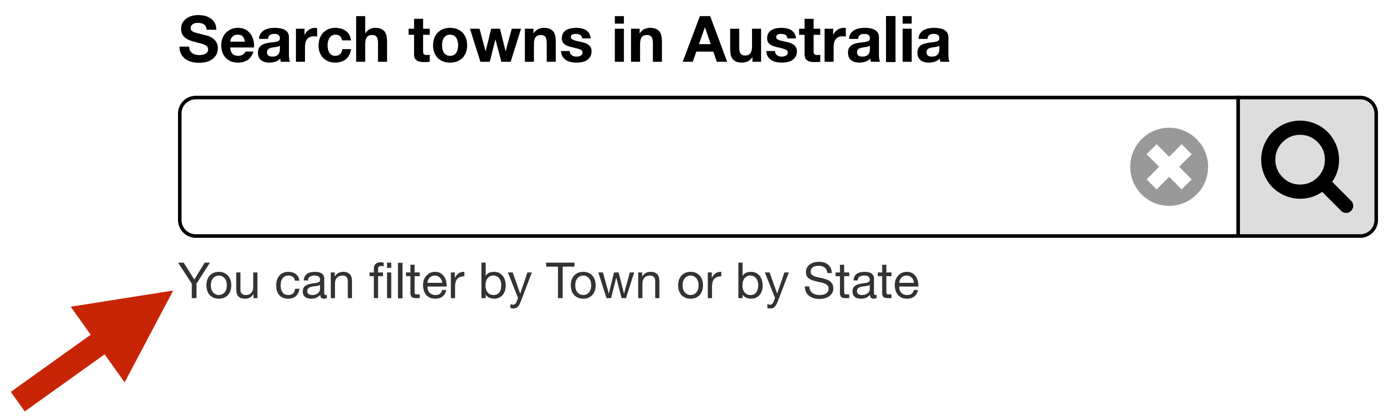


Diagram showing information under the input “You can filter by Town or by State”

3. The `placeholder` attribute should not be used for **complex instructions**.

This attribute it is often **displayed in a faint colour** which fails WCAG colour contrast guidelines.

It is also **wiped as soon as the user begins typing** so instructions become unavailable.

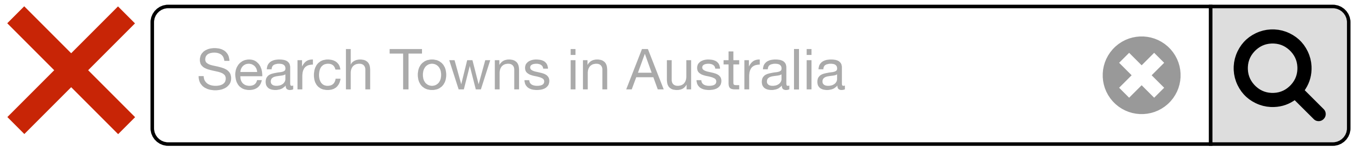


Diagram showing placeholder “Search towns in Australia” with red cross beside placeholder

4. The list of autocomplete suggestions could **highlight the string typed by the user**.

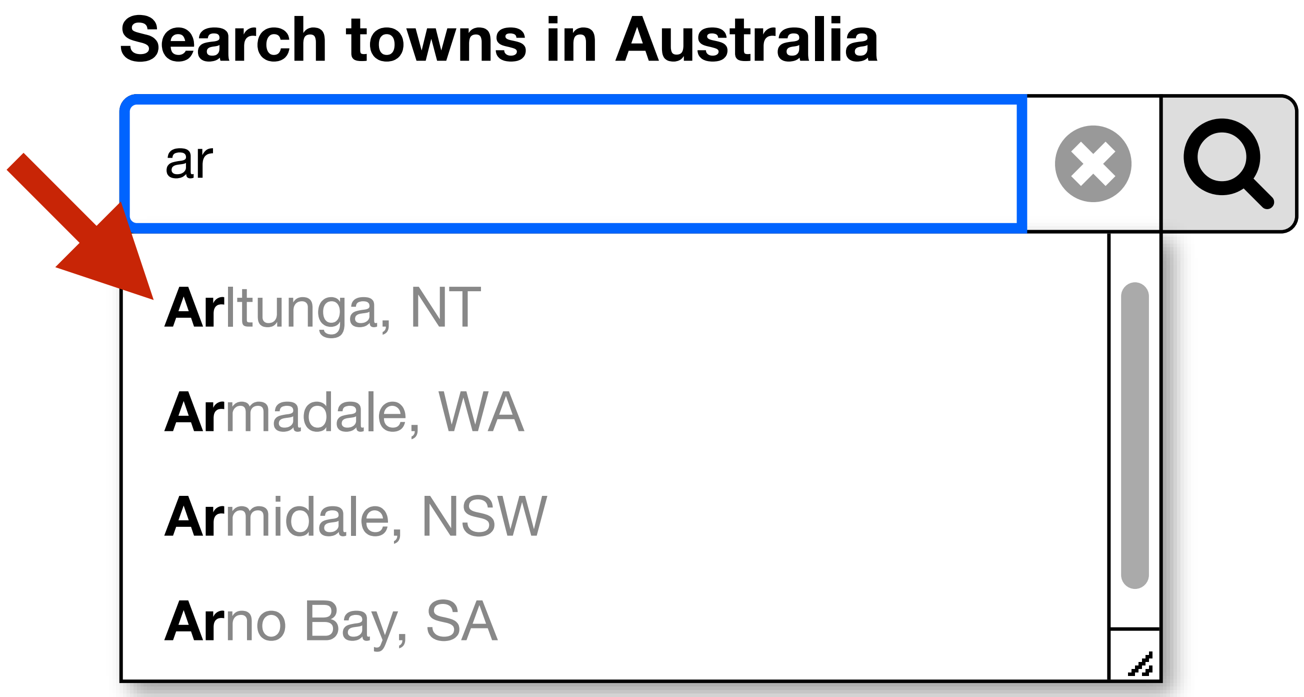
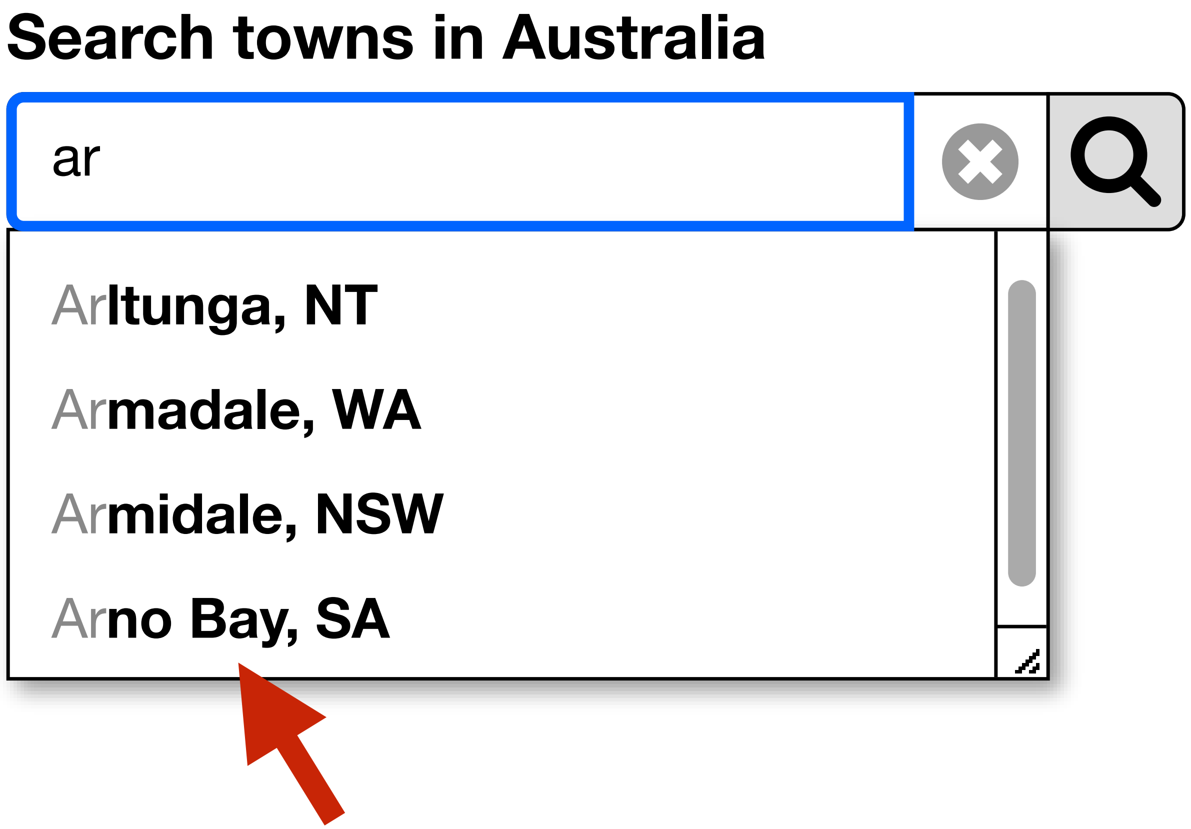


Diagram showing autosuggest options highlighting the user string

Or, the list could the **highlight everything apart from** the string typed by the user.


Search towns in Australia

ar

Arltunga, NT

Armadale, WA

Armidale, NSW

Arno Bay, SA

Diagram showing autosuggest options highlighting the non-user string

Both of these methods are beneficial as they help users **understand the relationship** between their string and the resulting options.

5. Users should be able to quickly identify what type of strings will **trigger the autocomplete**?

Does the search work based on the **initial characters** of suggestions…

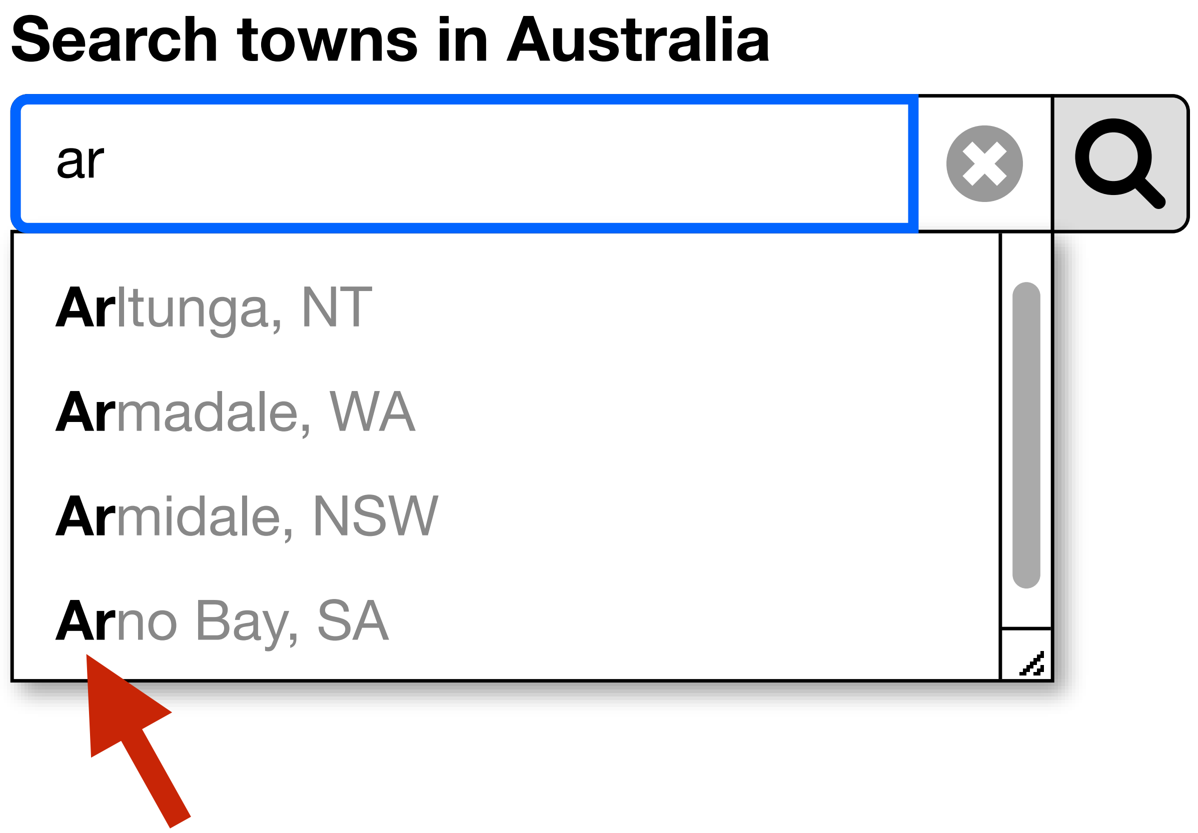


Diagram showing autosuggest options highlighting the user string at the start of each item

Or **any characters** within the suggestion?

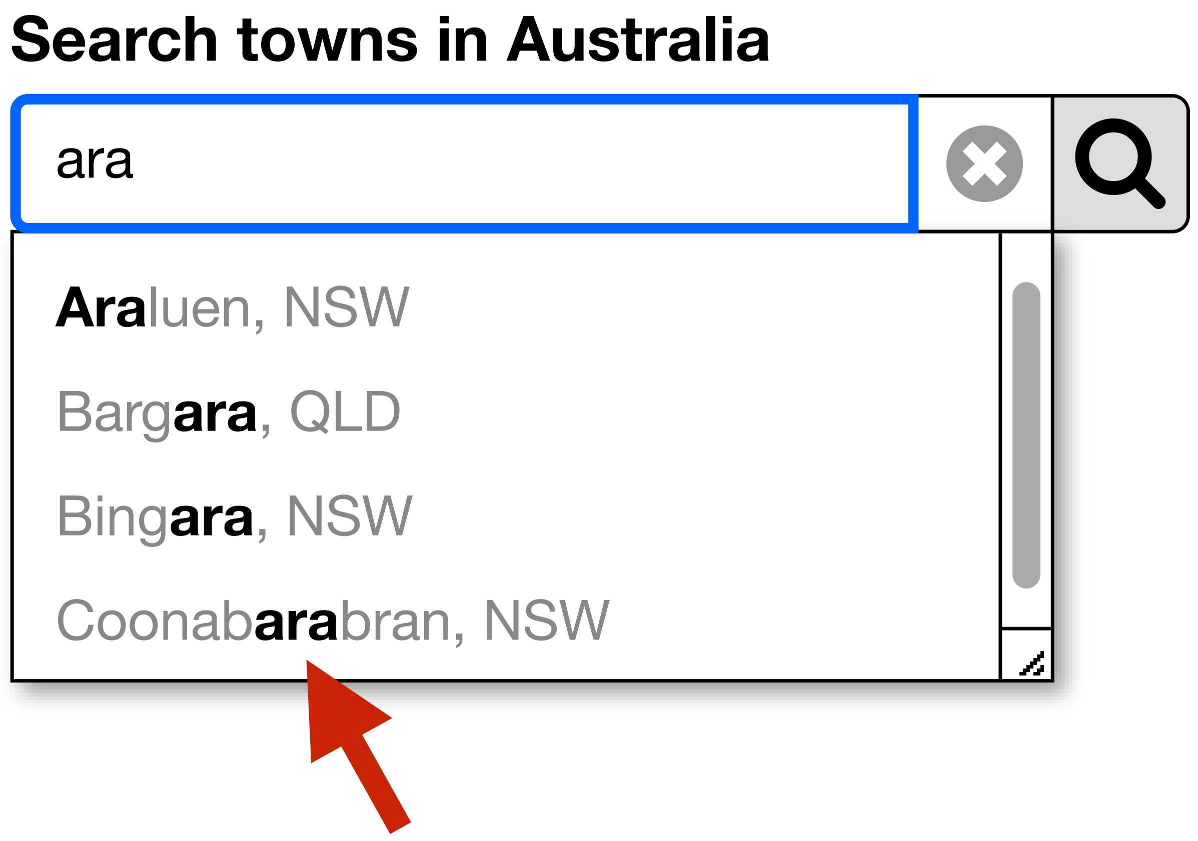


Diagram showing autosuggest options highlighting the user string within each item

6. Any autocomplete **suggestions should be accurate**.

Users should not be presented with suggestions **that do not match their typed strings**.

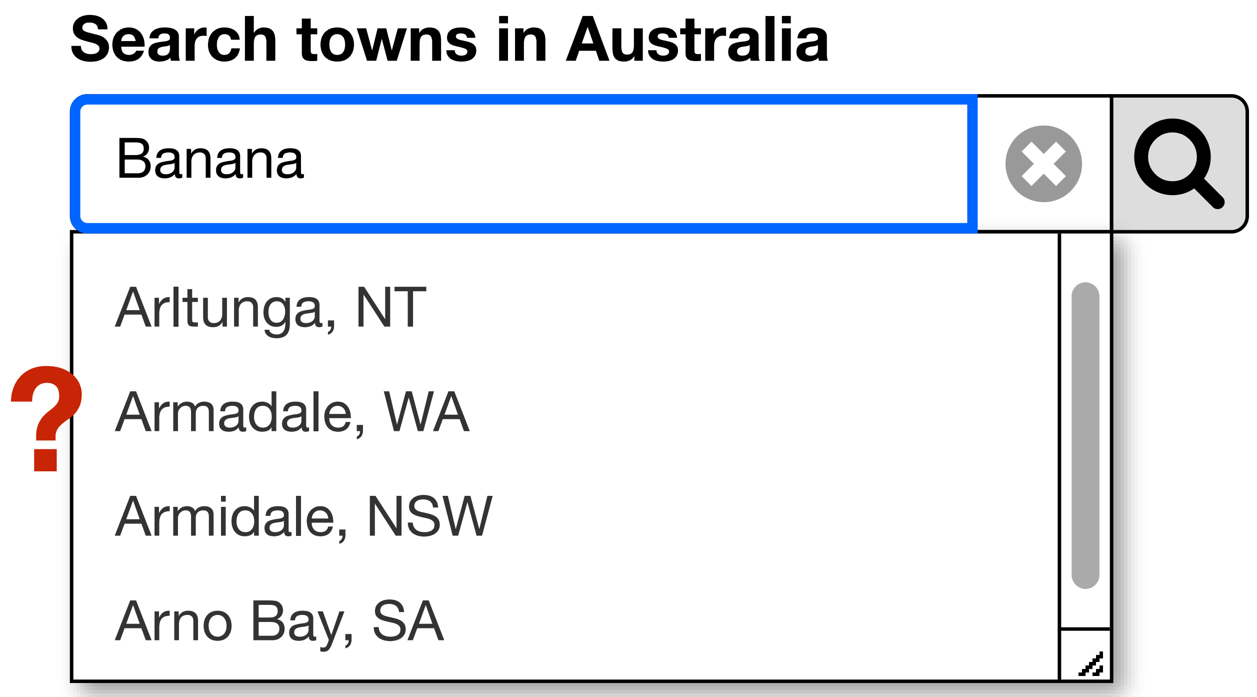


Diagram showing autosuggest options that do not match the user string

7. Users should be able to **easily clear the search form** of previously typed strings.

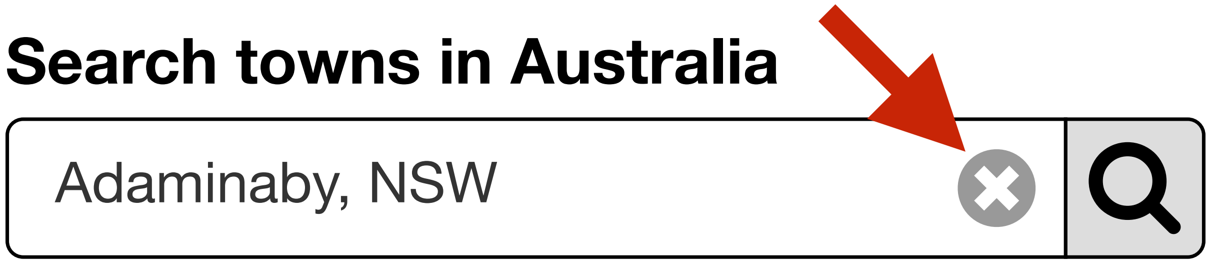


Diagram showing highlighted clear component

8. Ideally, there should be some **clearly defined submit action** associated with the search.

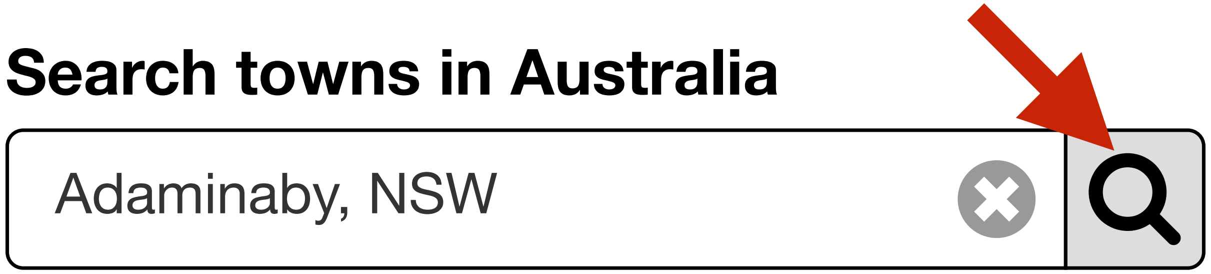


Diagram showing highlighted submit component

Some search functions return “live filtering results”. These are results that **dynamically change** as the user types.

In these cases, a submit button may **seem redundant** as there is nothing to submit.

However, Screen Reader users may not be aware that their typed strings have **already delivered a result** in a different area of the page.

While it's possible to **inform users that changes have occurred**, a submit button is an easy method to get around this problem.

Keyboard-only

Regardless of the method used to build an auto-complete, it should be **accessible to keyboard-only users**.

Focus

Any web page or web app should have **clear visual indicators** to help keyboard-only users determine which element is currently in focus.

This could just be the **default browser focus ring**…

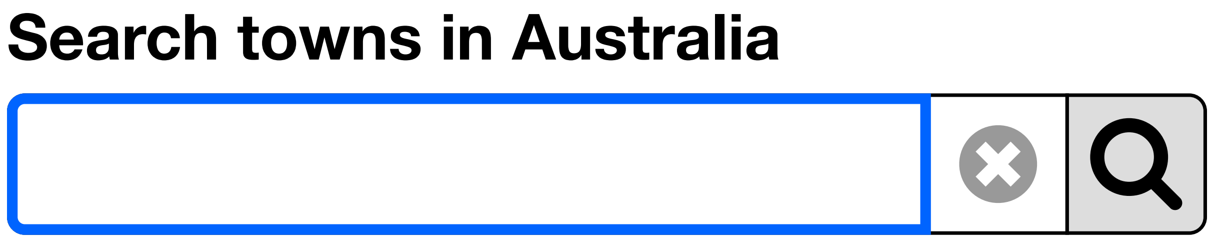


Diagram showing input in focus - indicated with blue focus ring

Or using your own **visual indicator methodology**.

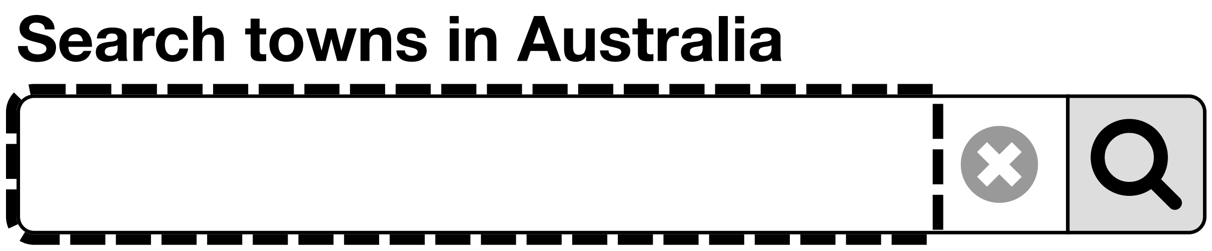


Diagram showing input in focus - indicated with black dotted lines

It should **never be hard or impossible** for keyboard-only users to see what is in focus.

```css
/* Bad practice */
input:focus {
  outline: none;
}
```

Ideally, the visual indicator methodology **should be consistent** across all focusable elements.

Users should not have to learn different visual indicators **just to understand what is currently in focus**.

Keystrokes

Keyboard-only users should be able to **perform any of the following actions**…

1. Use the `TAB` keystroke to move focus into the **search input field** from a previous element with focus.

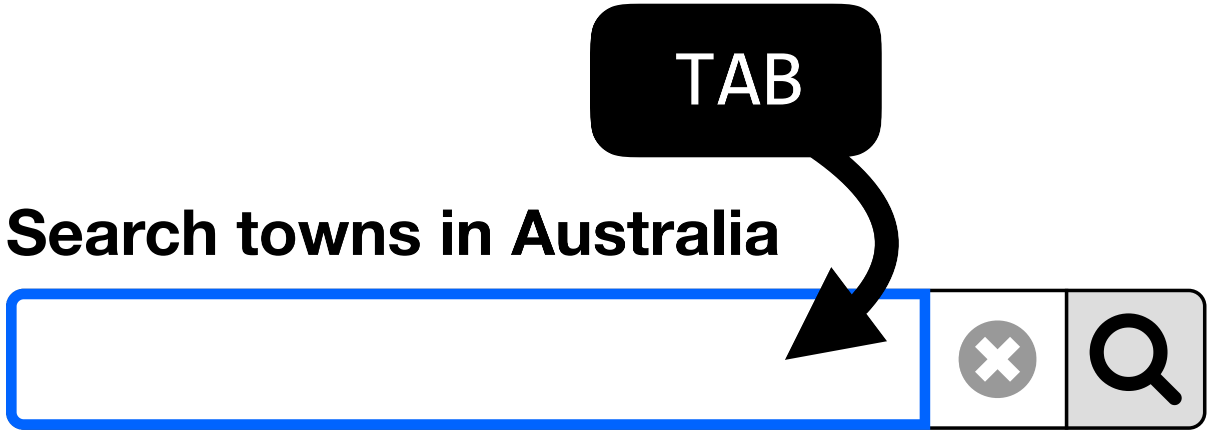


Diagram showing input in focus

2. Use the `TAB` keystroke to move focus from the search input **to the “clear” button**.

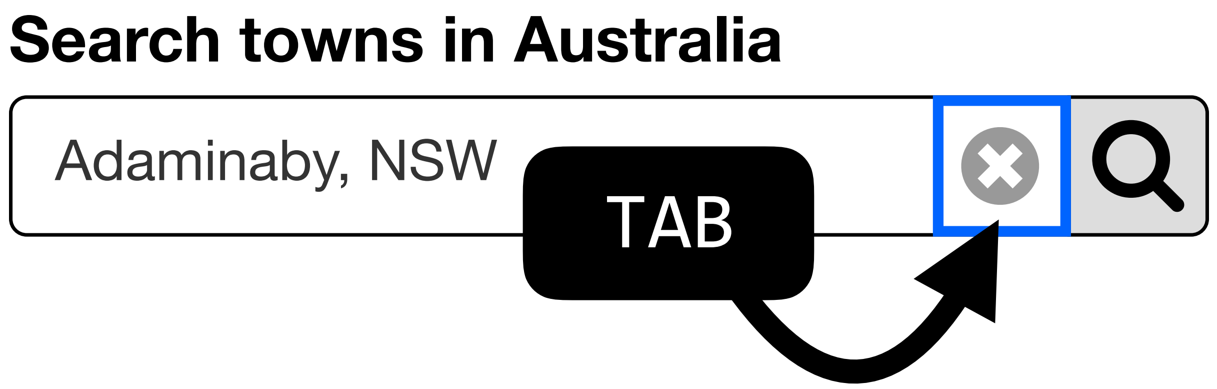


Diagram showing clear button in focus

3. Use the `ENTER` keystroke to **trigger the “clear” button**.

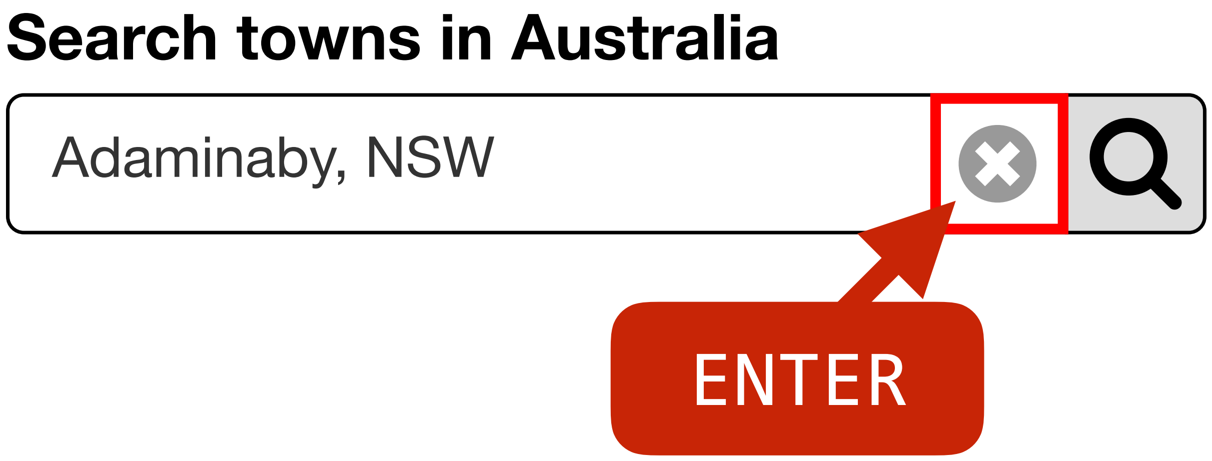


Diagram showing selected clear button

Note: When the “clear” button has been triggered, the search input field should be cleared and focus should **shift to this field** again.

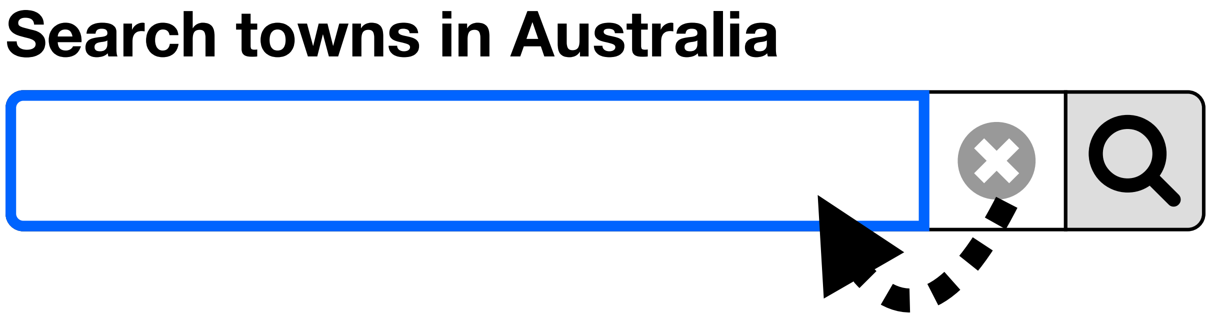


Diagram showing focus move for clear button back to search input

4. Use the `TAB` keystroke to move focus from the clear button **to the submit button**.

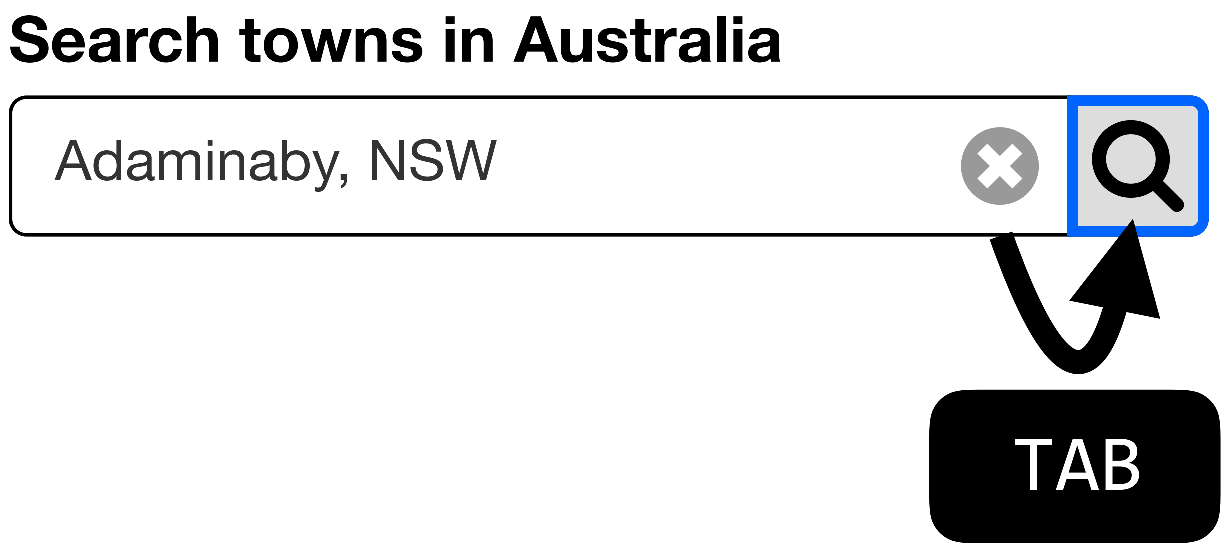


Diagram showing focus move form the clear button to the submit button

5. Use the `ENTER` keystroke to **trigger the submit button**.

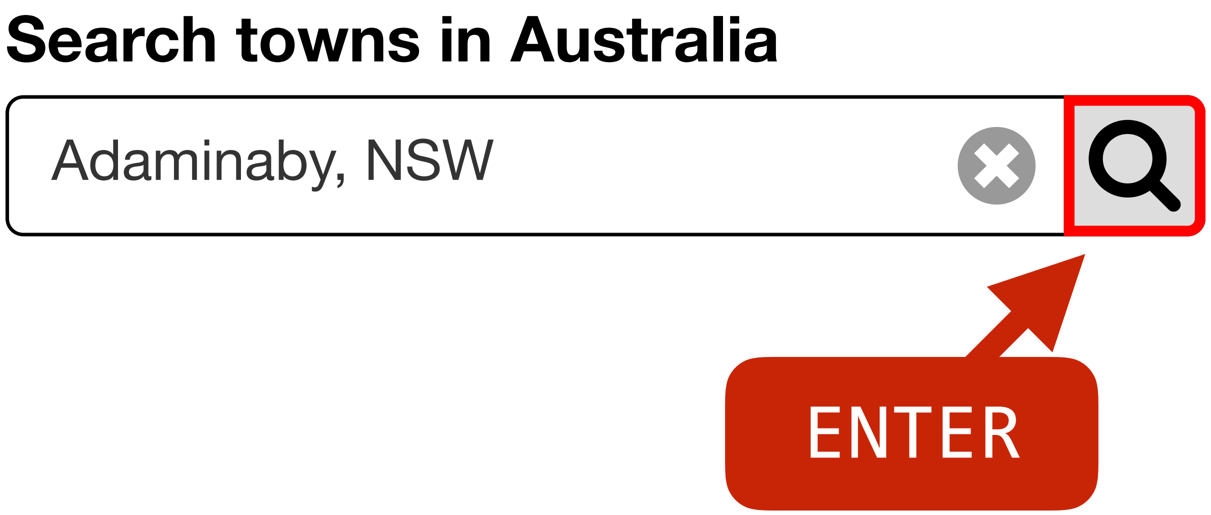
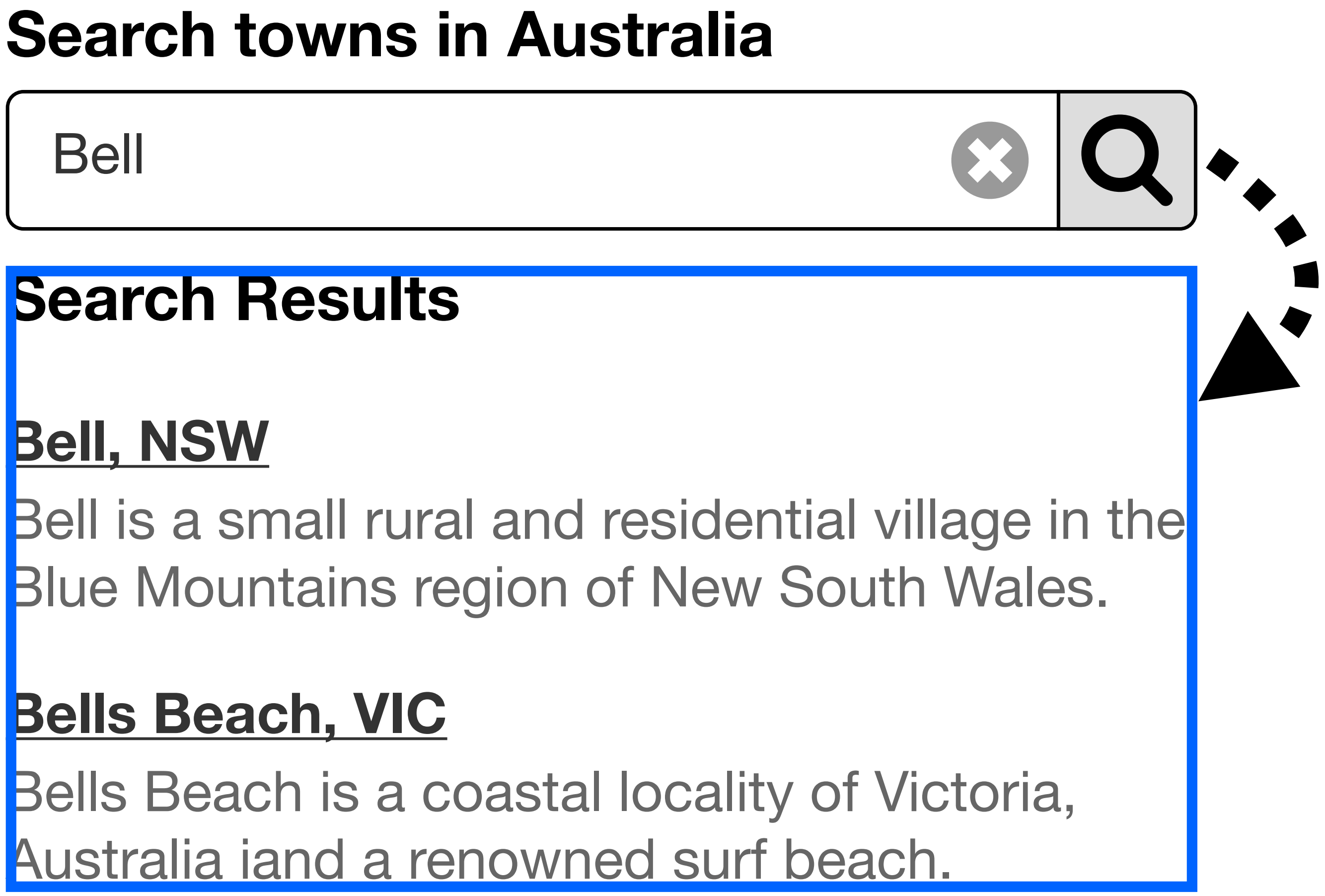


Diagram showing selected submit button

Note: When the submit button has been triggered, focus should **shift to the search result area** below the autocomplete search widget.



Search towns in Australia

Bell

Search Results

Bell, NSW

Bell is a small rural and residential village in the Blue Mountains region of New South Wales.

Bells Beach, VIC

Bells Beach is a coastal locality of Victoria, Australia iand a renowned surf beach.

Diagram showing focus move from submit button to search results

6. Use the `DOWN ARROW` keystroke to move focus from the search input field **to the first item in list of autocomplete suggestions**.

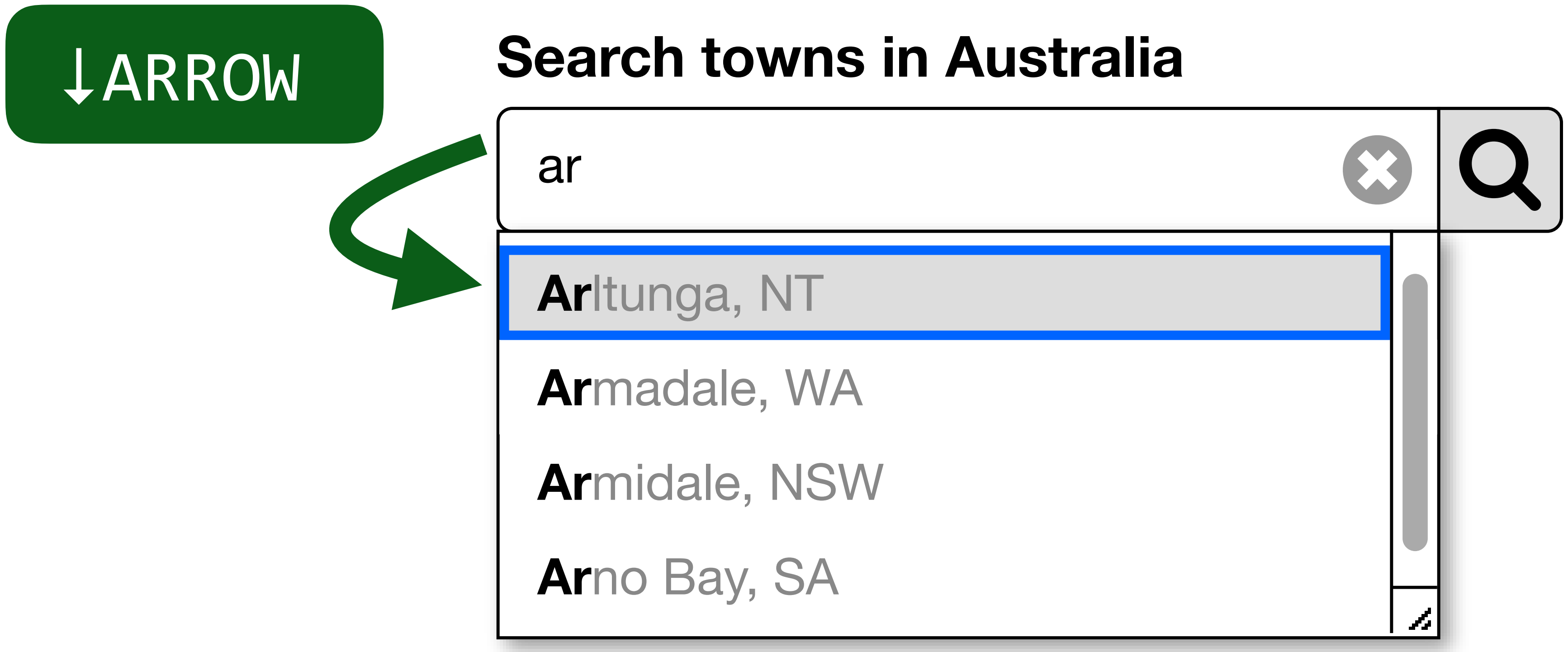


Diagram showing focus move from input to first suggestion

7. Use the `UP ARROW` and `DOWN ARROW` keystrokes **to navigate backwards and forwards** through suggestions.

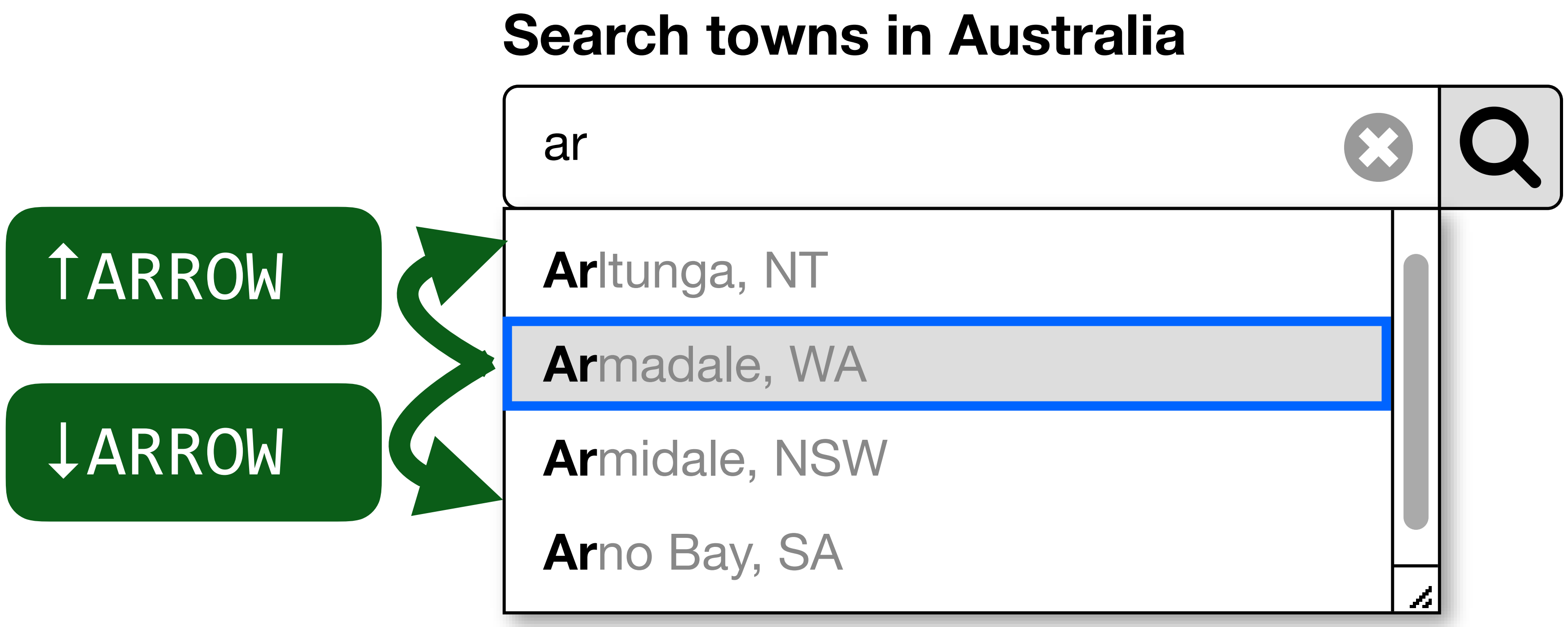


Diagram showing second selection in focus and arrows to indicate focus can move backwards or forwards

Note: The autosuggest item in focus **should always be in view** if there is a scrolling mechanism in place.

8: Users should not be able to `DOWN ARROW` **past the last suggestion option**.

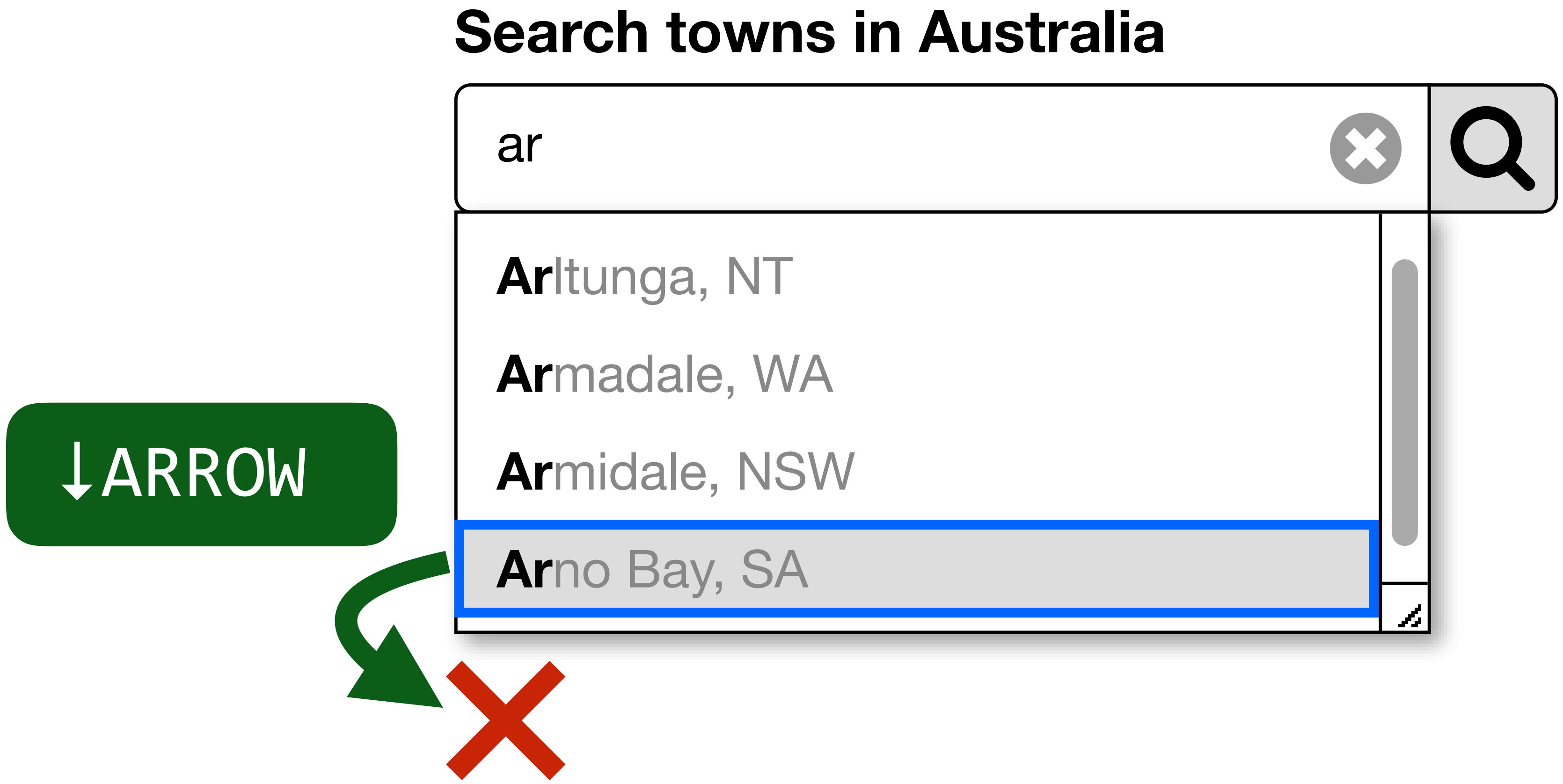


Diagram showing last selection in focus and red cross to indicate focus cannot go forward

9. Some developers allow `DOWN ARROW` keystrokes to loop from the last suggestion **directly back to the initial input box**.

However, I have found that **some users find this confusing**. They may not be aware that they have returning to the input field.

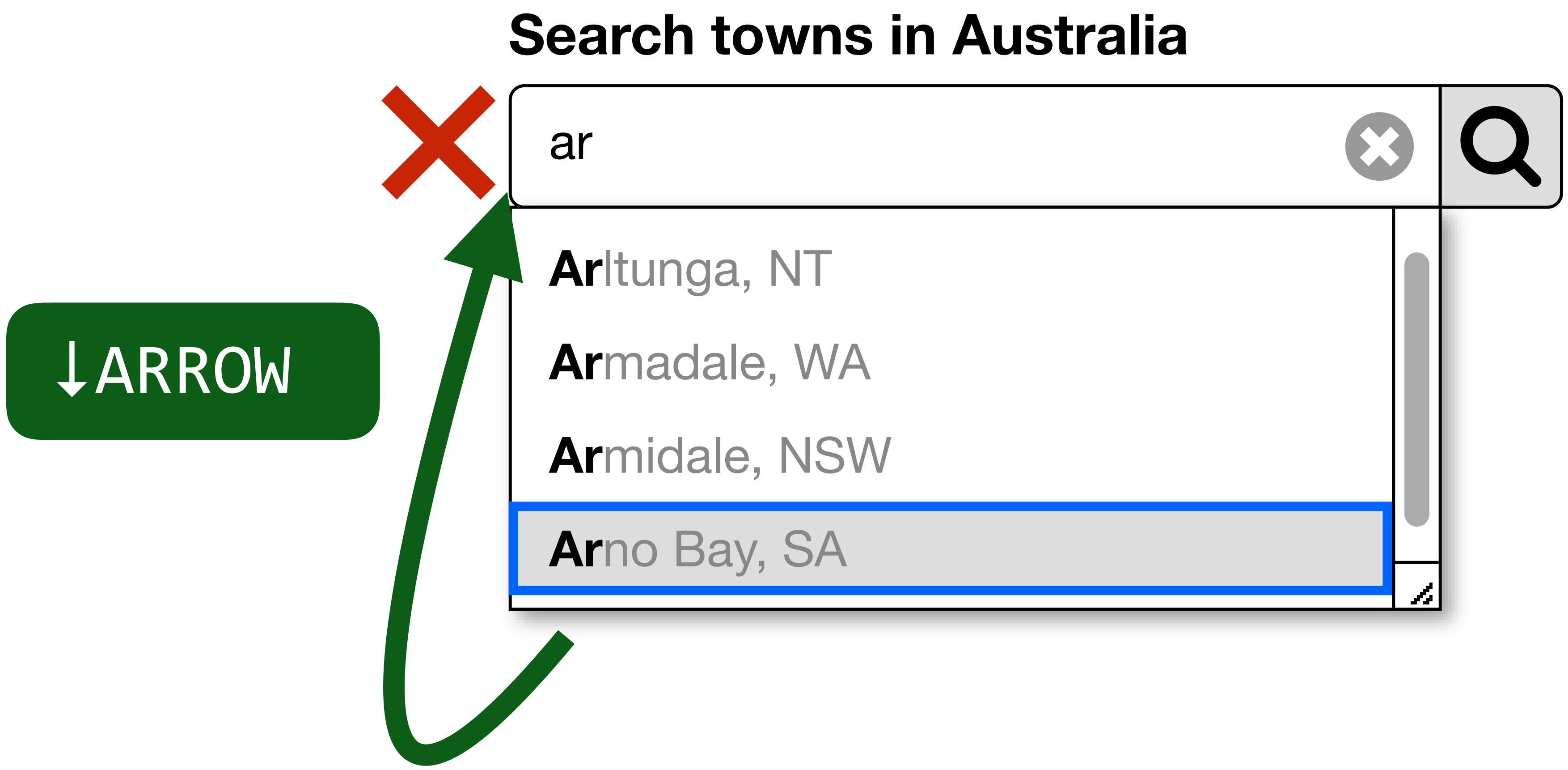


Diagram showing last selection in focus and red cross to indicate focus cannot jump to search input

10. However, users should be able to `UP ARROW` from the first suggestion **back into the search input field**.

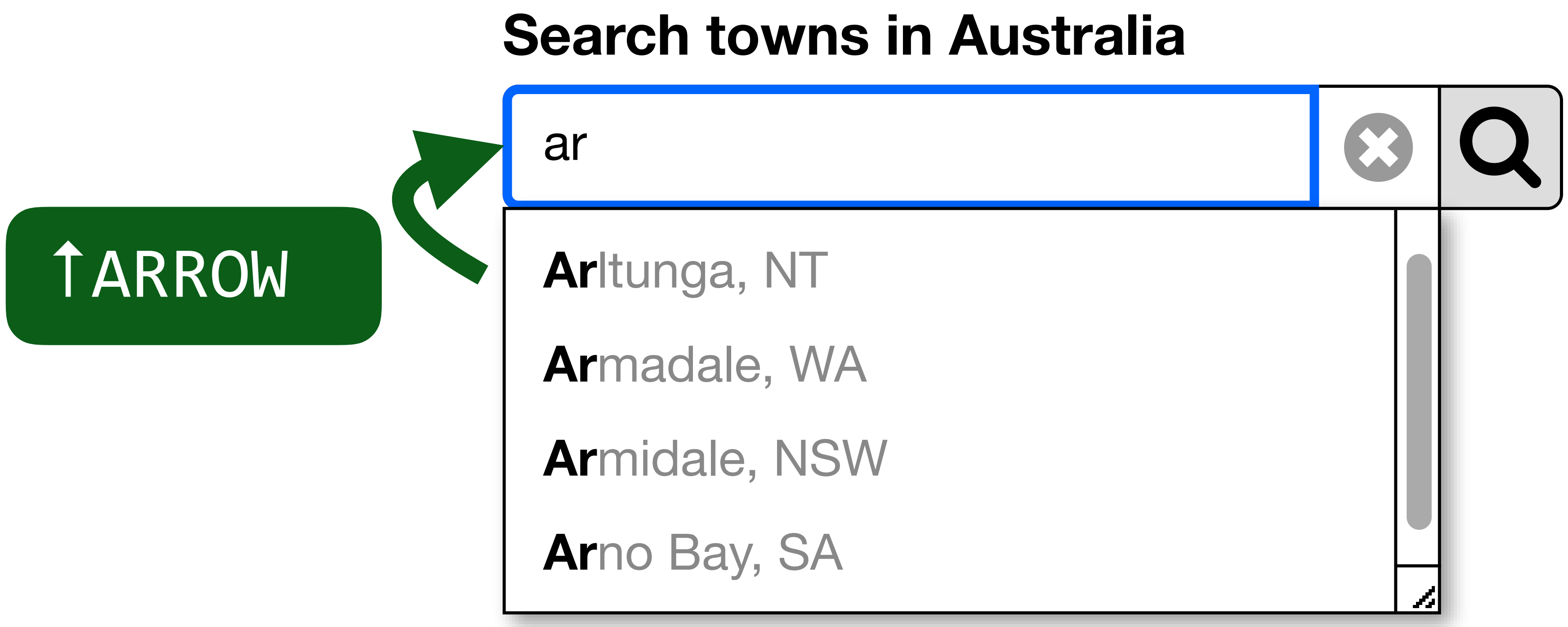


Diagram showing focus move from autosuggest dropdown to search input

11. Use the `ENTER` keystrokes to **select an autocomplete suggestion**.

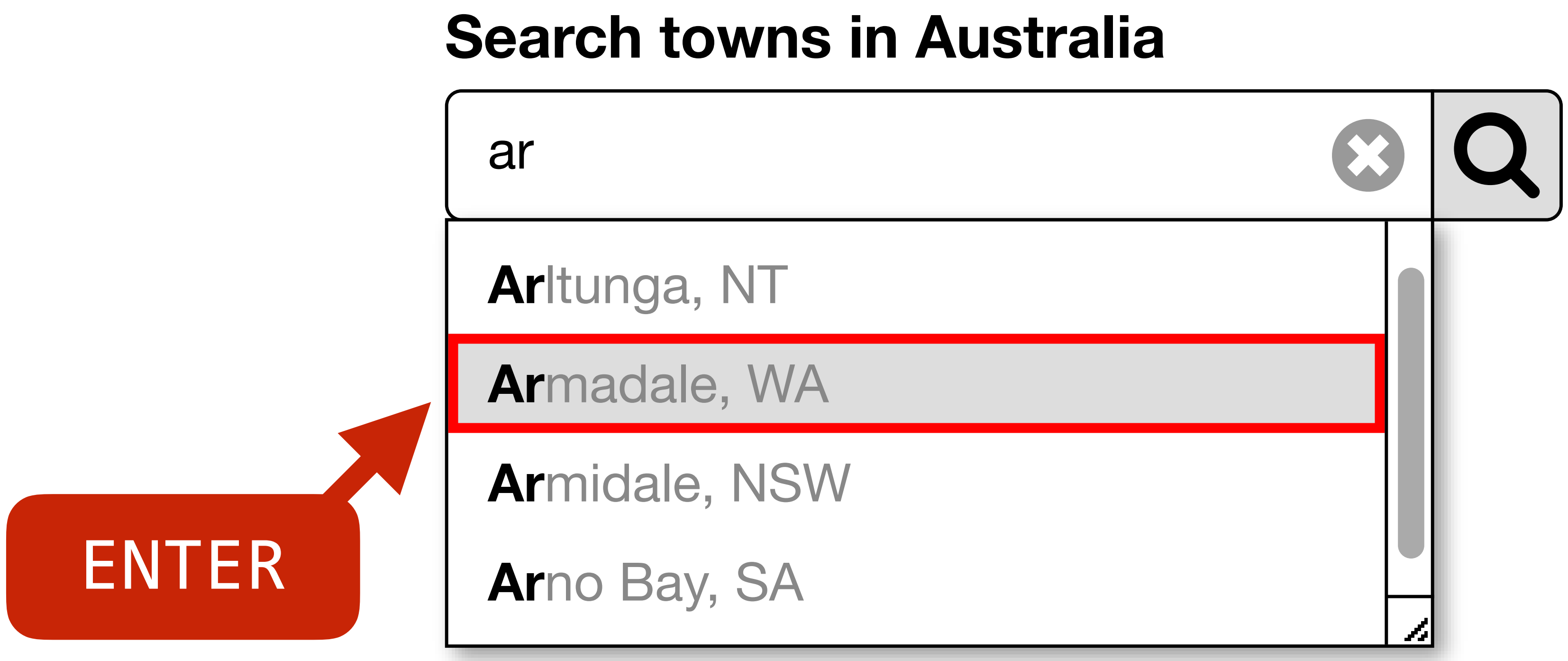


Diagram showing selected suggest option

Note: When the `ENTER` keystroke has been triggered, focus should **shift back to the search input field**.

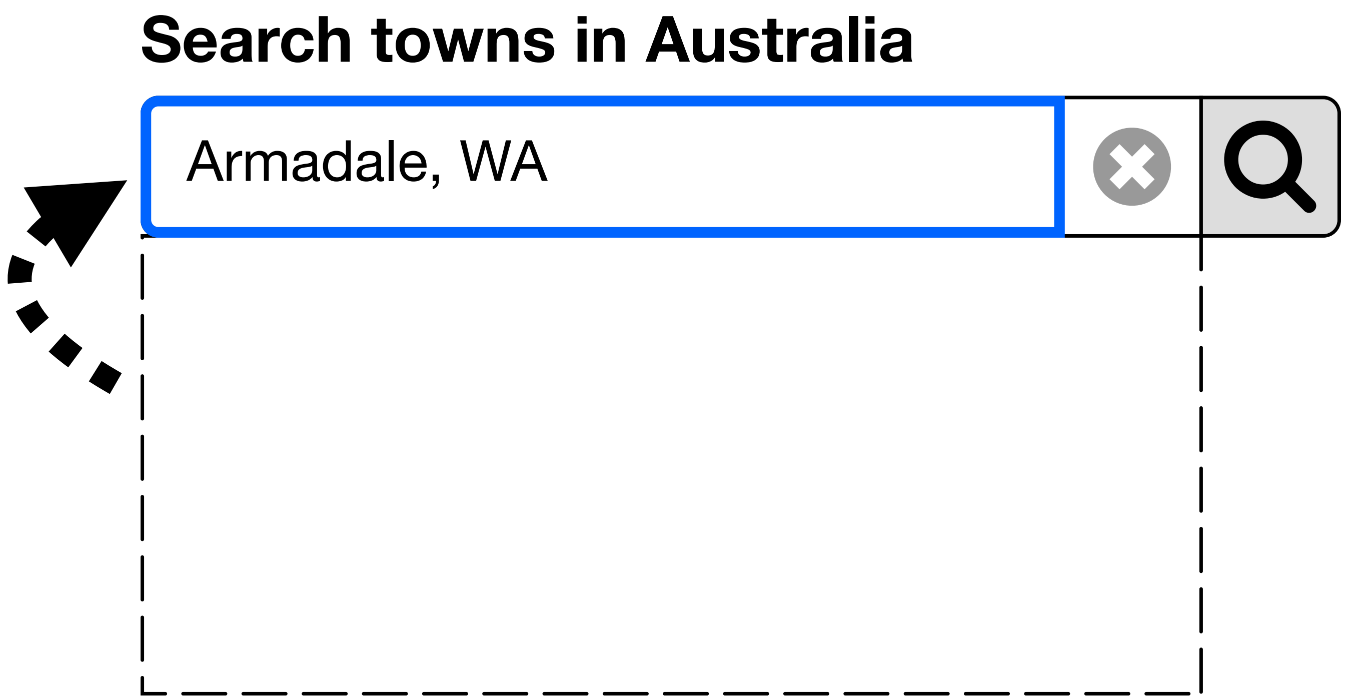


Diagram showing focus move from selected suggestion to search input field

12. Use the `ESC` keystroke **to close the suggestion list** and return focus to the initial input (i.e. if none of the suggestions are relevant).

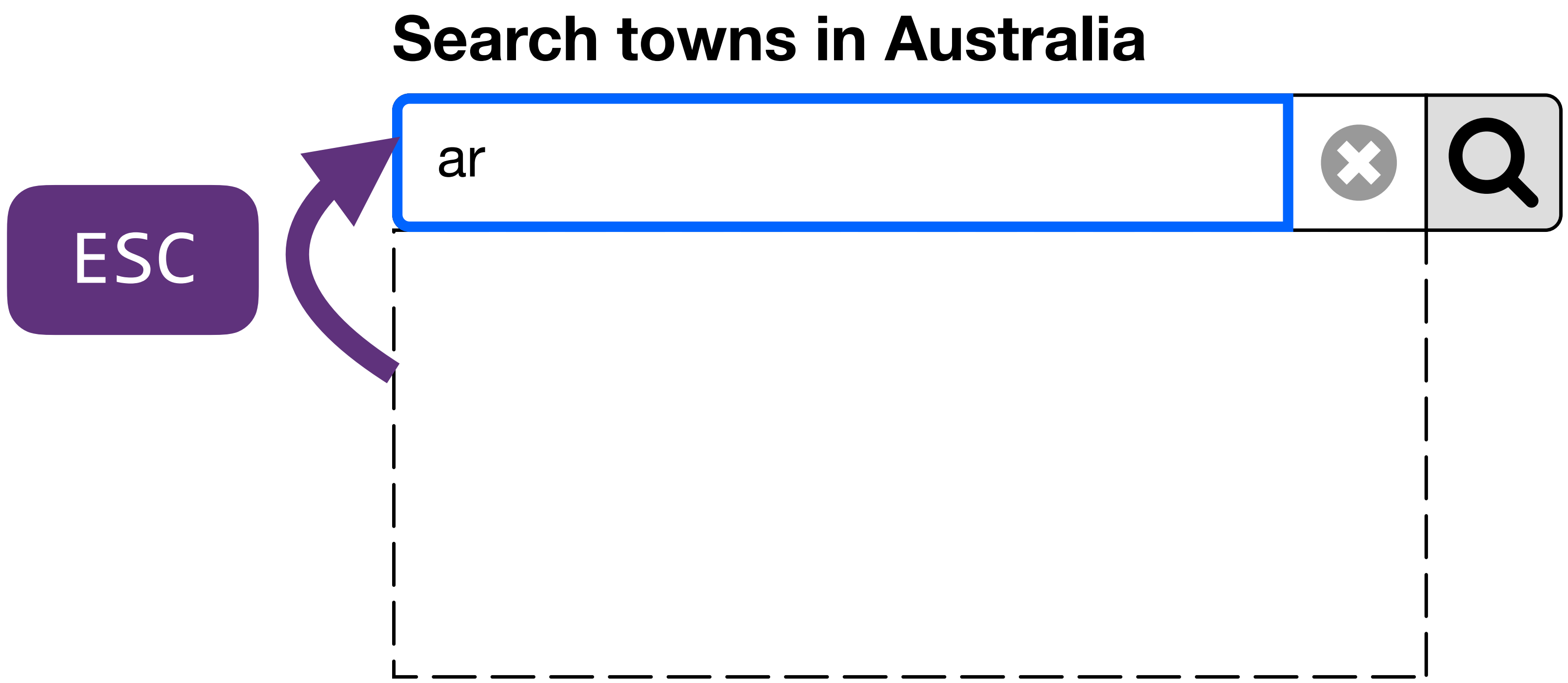


Diagram showing focus returning to input

Markup for screen readers

There are a wide range of different ways to mark up an accessible auto-suggest widget. Here are **some suggestions**.

A quick overall view of the markup components

The widget should be **wrapped inside a `<form>` element**.

```
<form action="#">
</form>
```

The `<label>` and `<input>` elements are the **core of the search button**.

```
<form action="#">
  <label></label>
  <input>
</form>
```

In our example, one `<button>` element will be used to **“clear” user input**.

```html
<form action="#">
  <label></label>
  <input>
  <button></button>
</form>
```

And a second `<button>` element will be used to **submit the form**.

```
<form action="#">
  <label></label>
  <input>
  <button></button>
  <button></button>
</form>
```

The `<ul>` allows us to display the **list of suggestions** when appropriate.

```html
<form action="#">
  <label></label>
  <input>
  <button></button>
  <button></button>
  <ul>
    <li></li>
    <li></li>
    <li></li>
  </ul>
</form>
```

The `<div>` element allows us to provide **hidden instructions** for screen reader users.

```
<form action="#">
  <label></label>
  <input>
  <button></button>
  <button></button>
  <ul>
    <li></li>
    <li></li>
    <li></li>
  </ul>
  <div></div>
</form>
```

FOR and ID attributes

In order to **explicitly associate** the `<label>` element with the `<input>` element, we should use `for` and `id` attributes.

```html
<label for="search">Search towns in Australia</label>
<input
  id="search"
  type="text"
  aria-describedby="instructions"
  aria-owns="results"
  aria-expanded="false"
>
```

TYPE attribute

The `<input>` element’s `type` attribute could be set to a value of `"text"` or `"search"`.

```html
<label for="search">Search towns in Australia</label>
<input
  id="search"
  type="text"
  aria-describedby="instructions"
  aria-owns="results"
  aria-expanded="false"
>
```

```html
<label for="search">Search towns in Australia</label>
<input
  id="search"
  type="search"
  aria-describedby="instructions"
  aria-owns="results"
  aria-expanded="false"
>
```

However, it is important to consider **how the “clear” button will operate**.

Some browsers, like Chrome and Safari will display an `<input>` type of `"search"` with a native “clear” button at the right side of the input. **Other browses like Firefox, do not**.

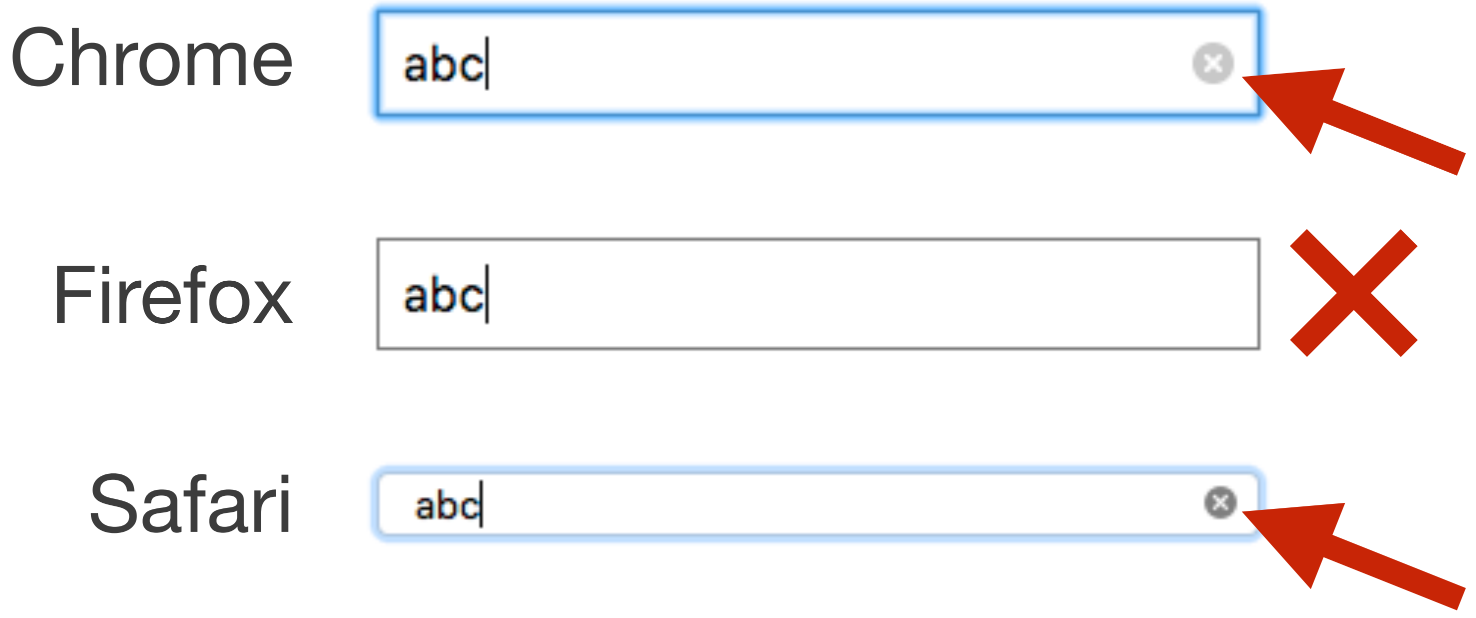


Diagram showing Chrome and Safari’s clear button. Firefox has no clear button.

More importantly, this native “clear” button often **cannot be accessed via the** `TAB` **keystroke**, so it is inaccessible for many Assistive Technology users.

So, if you want to use a robust and accessible “clear” button, **it is better to use a separate `<button>` element**.

Using CSS, you can make the clear button **look like it sits inside** the `<input>` element.

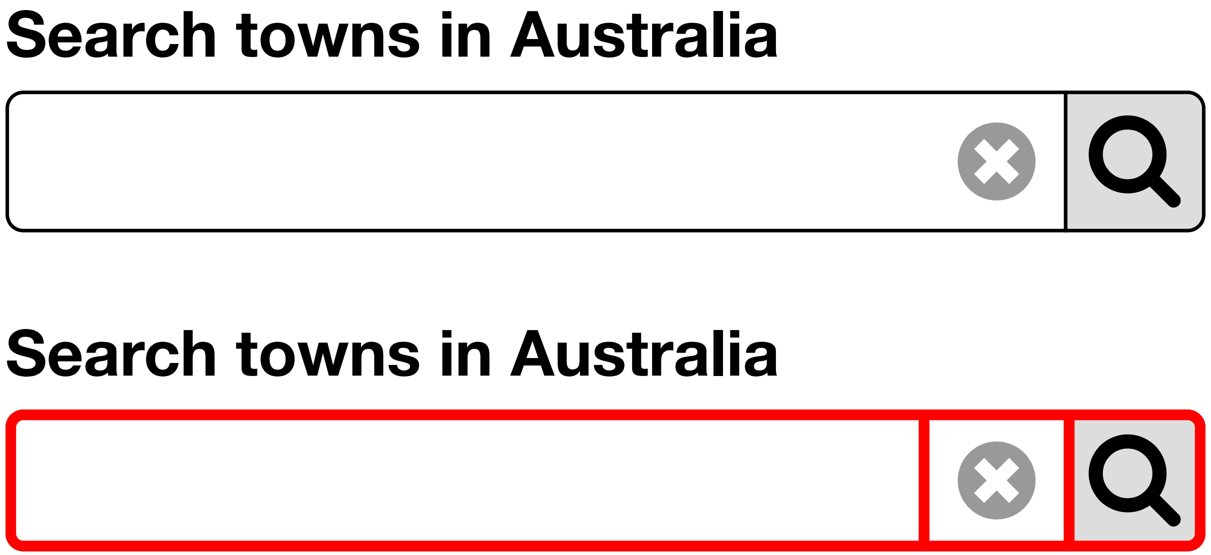


Diagram showing three different elements - the input, the clear and the submit button

Then **make sure** to set the `type` to `"text"` rather than `"search"`.

```html
<label for="search">Search towns in Australia</label>
<input
  id="search"
  type="text"
  aria-describedby="instructions"
  aria-owns="results"
  aria-expanded="false"
>
```

aria-describedby attribute

The `aria-describedby` attribute allows us to **describe the purpose** of the current element. It points the current element to a new element with a matching `ID` value.

```html
<label for="search">Search towns in Australia</label>
<input
  id="search"
  type="text"
  aria-describedby="instructions"
  aria-owns="results"
  aria-expanded="false"
>

<div id="instructions" aria-live="assertive" style="display:
none;">
  ...
</div>
```

This allows us to provide **basic instructions on the use** of the widget for assistive technologies.

```
<div id="instructions" aria-live="assertive" style="display:
none;">
  When autocomplete options are available, use up and down arrows
to review and enter to select.
</div>
```

aria-owns attribute

The `aria-owns` attribute allows us to define “**a parent/child contextual relationship** to assistive technologies that is otherwise impossible to infer from the DOM”.

In other words, we can define the `<input>` element as the **parent**, and the `<ul>` element as the **child** element.

```html
<label for="search">Search towns in Australia</label>
<input
  id="search"
  type="text"
  aria-describedby="instructions"
  aria-owns="results"
  aria-expanded="false"
>

<ul id="results">
  ...
</ul>
```

aria-expanded attribute

The `aria-expanded` attribute allows us to **inform** assistive technologies when the autocomplete dropdown is present. It is initially set to `"false"`.

```
<label for="search">Search towns in Australia</label>
<input
  id="search"
  type="text"
  aria-describedby="instructions"
  aria-owns="results"
  aria-expanded="false"
>
```

This value needs to **dynamically change** to `"true"` as soon as the autocomplete suggestions are present.

```html
<label for="search">Search towns in Australia</label>
<input
  id="search"
  type="text"
  aria-describedby="instructions"
  aria-owns="results"
  aria-expanded="true"
>
```

Buttons

Directly after the input, we need **two** `<button>` elements.

The first `<button>` should be a type of `"button"` and allow users to **clear the input**.

```
<button type="button" aria-label="Clear"></button>
<button type="submit" aria-label="Search"></button>
```

The second `<button>` should be a type of `"submit"` and allow users to **submit the form**.

```
<button type="button" aria-label="Clear"></button>
<button type="submit" aria-label="Search"></button>
```

You may want to use **icons instead of text** for one or both of the buttons. In this case we are using “clear” and “search” icons.

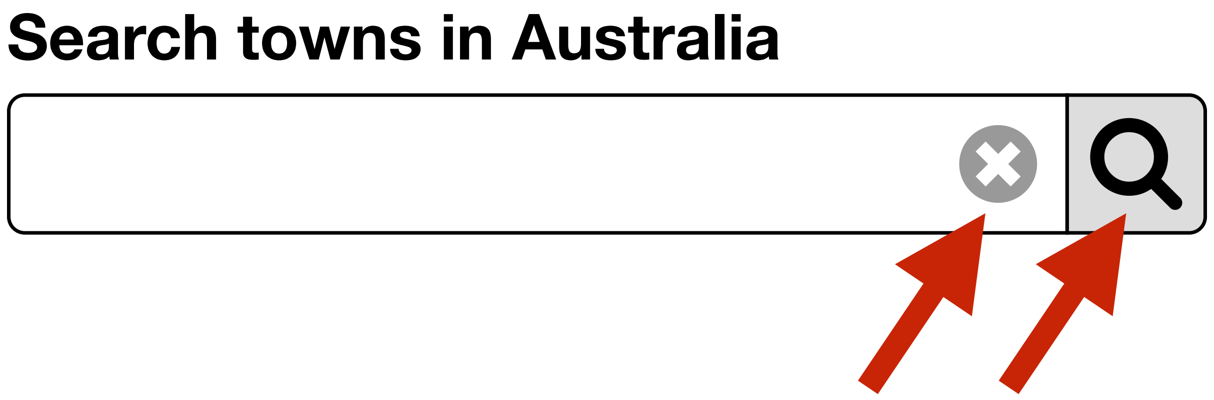


Diagram showing clear and search icons

However, if you user icons instead of text, you will need to **provide additional context** for Assistive Technologies.

In this case, we can use `aria-label` attributes to provide **hidden labels for both buttons**.

```html
<button type="button" aria-label="Clear"></button>
<button type="submit" aria-label="Search"></button>
```

Unordered list

After the two button elements, we need to add the `<ul>` element which will be used to **display the autocomplete suggestions**.

```html
<ul
  id="results"
  role="listbox"
  tabindex="-1"
  style="display: none;"
>
  <li role="option" aria-selected="false">apple</li>
  <li role="option" aria-selected="false">banana</li>
  <li role="option" aria-selected="false">pear</li>
</ul>
```

As mentioned before, the `ID` attribute's value of `"results"` allows us to determine that this element is "owned" by the parent `<input>` element.

```html
<ul
  id="results"
  role="listbox"
  tabindex="-1"
  style="display: none;"
>
  <li role="option" aria-selected="false">apple</li>
  <li role="option" aria-selected="false">banana</li>
  <li role="option" aria-selected="false">pear</li>
</ul>
```

The `role` attribute can be set with a value of `"listbox"`, which informs assistive technologies that the element is a widget that **allows the user to select one or more items** from a list of choices.

```
<ul
  id="results"
  role="listbox"
  tabindex="-1"
  style="display: none;"
>
  <li role="option" aria-selected="false">apple</li>
  <li role="option" aria-selected="false">banana</li>
  <li role="option" aria-selected="false">pear</li>
</ul>
```

To make sure the element **cannot be brought into focus** before it is triggered, we can set the `tabindex` attribute to `"-1"`.

```
<ul
  id="results"
  role="listbox"
  tabindex="-1"
  style="display: none;"
>
  <li role="option" aria-selected="false">apple</li>
  <li role="option" aria-selected="false">banana</li>
  <li role="option" aria-selected="false">pear</li>
</ul>
```

This value needs to **dynamically change** to `"0"` as soon as the autocomplete suggestions are present.

```html
<ul
  id="results"
  role="listbox"
  tabindex="0"
  style="display: block;"
>
  <li role="option" aria-selected="false">apple</li>
  <li role="option" aria-selected="false">banana</li>
  <li role="option" aria-selected="false">pear</li>
</ul>
```

To make sure the element is **initially hidden** we can set the `style` attribute to `"display:none"`.

```html
<ul
  id="results"
  role="listbox"
  tabindex="-1"
  style="display: none;"
>
  <li role="option" aria-selected="false">apple</li>
  <li role="option" aria-selected="false">banana</li>
  <li role="option" aria-selected="false">pear</li>
</ul>
```

This value needs to **dynamically change** to something like `"display:block"` as soon as the autocomplete options are triggered.

```
<ul
  id="results"
  role="listbox"
  tabindex="0"
  style="display: block;"
>
  <li role="option" aria-selected="false">apple</li>
  <li role="option" aria-selected="false">banana</li>
  <li role="option" aria-selected="false">pear</li>
</ul>
```

List items

Each of the `<li>` elements can be given a `role` attribute with a value of `"option"` with informs assistive technologies that they are **selectable items in a select list**.

```html
<ul
  id="results"
  role="listbox"
  tabindex="-1"
  style="display: none;"
>
  <li role="option" aria-selected="false">apple</li>
  <li role="option" aria-selected="false">banana</li>
  <li role="option" aria-selected="false">pear</li>
</ul>
```

Each of the `<li>` elements needs to have an `aria-selected` attribute **initially set** to `"false"`.

```
<ul
  id="results"
  role="listbox"
  tabindex="-1"
  style="display: none;"
>
  <li role="option" aria-selected="false">apple</li>
  <li role="option" aria-selected="false">banana</li>
  <li role="option" aria-selected="false">pear</li>
</ul>
```

This value needs to **dynamically change** to `"true"` if the individual option is selected.

```html
<ul
  id="results"
  role="listbox"
  tabindex="-1"
  style="display: none;"
>
  <li role="option" aria-selected="true">apple</li>
  <li role="option" aria-selected="false">banana</li>
  <li role="option" aria-selected="false">pear</li>
</ul>
```

The hidden DIV

After the `<ul>` element, we have the `<div>` element, which has the **instructions** for assistive technologies.

```
<div
  id="instructions"
  class="off-left"
  aria-live="assertive"
>
  When autocomplete options are available, use up and down arrows
to review and enter to select.
</div>
```

As mentioned before, the `ID` value allows us to **point** the `<input>` element to this `<div>` element via the `aria-describedby` attribute.

```
<div
  id="instructions"
  class="off-left"
  aria-live="assertive"
>
  When autocomplete options are available, use up and down arrows
to review and enter to select.
</div>
```

The `<div>` element needs to be **visually hidden**, but still available to screen readers.

This can be achieved by **setting it “off-left” using CSS**. So, we can give it a pretend “off-left” class here.

```
<div
  id="instructions"
  class="off-left"
  aria-live="assertive"
>
  When autocomplete options are available, use up and down arrows
to review and enter to select.
</div>
```

The `aria-live` attribute is set to `"assertive"`. This informs assistive technologies as soon as anything inside this element is **dynamically changed**.

```
<div
  id="instructions"
  class="off-left"
  aria-live="assertive"
>
  When autocomplete options are available, use up and down arrows
to review and enter to select.
</div>
```

We need this because **the instructions will dynamically change** as soon as the autocomplete options are triggered.

```html
<div
  id="instructions"
  class="off-left"
  aria-live="assertive"
>
  When autocomplete results are available use up and down arrows
to review and enter to select.
</div>
```

```
<div
  id="instructions"
  class="off-left"
  aria-live="assertive"
>
  6 options available. Use up and down arrows to review and enter
to select.
</div>
```

The instructions should also **immediately change as users type** if the number of suggestions changes.

```html
<div
  id="instructions"
  class="off-left"
  aria-live="assertive"
>
  3 options available. Use up and down arrows to review and enter
to select.
</div>
```

The ideal method?

As mentioned before, this is just **one method** that could be used.

Before deciding, make sure you **check out different methods and test them** - with real users, and across different assistive technologies.

Good examples

One of my favourite accessible autocomplete search widgets is the **haltersweb version**:

https://haltersweb.github.io/Accessibility/autocomplete.html

However, there are a **wide range of different solutions** available, such as:

http://www.visionaustralia.org/digital-access-autocomplete

https://a11y.nicolas-hoffmann.net/autocomplete-list/

https://alphagov.github.io/accessible-autocomplete/examples/

http://oaa-accessibility.org/examplep/combobox2/