Building accessible forms

Forms are one of the most important pieces of any website and web application and you would be surprised to learn how often it is the case that they are implemented incorrectly and/or without accessibility in mind.

Now, admittedly there are quite a few things that go into building an accessible form - and we will look at all of them during this lesson. There are a number of **design decisions** that we must make and **development hurdles** that we must cross, that are maybe not that obvious initially - but they will ultimately lead us to having fully accessible forms on the Web.

Labels & Descriptions

Let's start from the basics - every form has at least one **input**. This can be anything from a simple text input, to a nice looking switch (checkbox) to complicated menus and dropdowns, but all of them have at least one thing in common - **they must have a label**.

We have to label our inputs to let **both users who use and those who don't use assistive technologies** know that essentially: **"This is an input and this is what it's called"**.

The <label> element

The <label> is the only element you need to accomplish this. We can accompany any input element with a label that is associated with it and we can do that a couple of different ways:

Explicit labeling

We can have our label contain the **`for` attribute with its value being the ID of the input that it's associated with**.

```
HTML
<label for="first_name">First name</label>
<input id="first_name" type="text" placeholder="First name here..."></input>
```

If we **can't put the label that close to the input** (on the UI or in the HTML), we might go with this **explicit approach** so that even though they are not close together in the HTML - they are **programmatically connected** through the **`for`** and **`id`** attributes**.**

Implicit labeling

Conversely, we can implicitly label an input by **wrapping it inside of a <label> element**.

```html
<label>
  First name
  <input type="text" placeholder="First name here..."></input>
</label>
```

If **the label and the input are packed closely together**, we might find it simpler to just place the input inside the label and not worry about IDs.
It all depends on the UI that you want to build - just remember that **we always want to label our inputs**.

Invisible labels

The labeling strategies we talked about previously, fall into what's called **visible labeling**.
There are other ways we can label our inputs without providing a visible one. This is called **invisible labeling**.

Invisible labels can be added through the ARIA attributes **`aria-label`** or **`aria-labelledby`**.
If we want to add an **explicit invisible label (which only assistive technologies will be able to read)**, we can use **`aria-label`** as such:

```html
<input type="email" placeholder="Your email here.." aria-label="Email"></input>
```

If we wanted our input to be **labeled according to some other element's content**, we can use **`aria-labelledby` and connect it with the labeling element's ID**:

```html
<p id="email">User email</p>
...
<input type="email" placeholder="User email" aria-labelledby="email"></input>
```

Now the first question you might have at this point could be: **“Why should we have invisible labels in the first place? Shouldn’t the label always be visible?”**

This is a valid question but as with almost everything in life - **it depends on the situation**. There are certain UI scenarios where it is not feasible or even possible to have a visible label next to the input for any number of reasons.

Some examples include:

- **Spreadsheet interfaces (tables)** like Google Sheets or Excel where each cell is an input
- **URL search bars** on browsers
- **Quick filters** where the **context around them explains their purpose**

As you will see in multiple occasions during this lesson, **accessibility requires a heavy context-based approach to solving problems**. There aren’t any silver bullet solutions to a problem - it depends on the situation and the UI that we are aiming to create.

Now, **as developers it is our responsibility to have as many “tools in our toolbelt” and solutions to any given problem as possible**, so that we can tackle these different UI scenarios when they come before us.

Group labeling

There are also situations where we have **multiple inputs that are logically associated with each other** - like a **shipping address** section in a form that has multiple inputs for your street’s name, number, city name, zip code etc.
In this case, we need to have one label (ex. “Shipping Address”) that is associated with that entire section - this is called **group labeling** or **grouping controls**.

We can do such groupings in essentially two ways: using the **<fieldset>** and **<legend>** elements around the inputs we want to group, or **using roles + ARIA attributes**.

Fieldset & Legend

Using the <fieldset> and <legend> elements to indicate a group of inputs is the traditional grouping strategy where we **wrap an entire section inside our form with the <fieldset> element** and **provide the group label within a <legend> element** like so:

```html
<form>
 ...
  <fieldset>
    <legend>Shipping Address:</legend>

    <label for="shipping_street">Street:</label>
    <input type="text" id="shipping_street"></input>

    <label for="street_number">Number:</label>
    <input type="text" inputmode="numeric" id="street_number"></input>

    <label for="zip_code">ZIP Code:</label>
    <input type="text" inputmode="numeric" id="zip_code"></input>
  </fieldset>
 ...
<form>
```

It is important to note that this approach comes with some **styling limitations**. Depending on how you need to style your UI, it could prove quite difficult to get around the HTML structure and styling constraints to make the UI look like you wish to, just because in this approach the HTML structure is quite rigid in how it needs to be set up.

Group labeling with roles + ARIA attributes

We can use the **ARIA role** of **"group"** on a container element that wraps our form inputs and **connect a group label to is using aria-labelledby** like this:

```html
<div role="group" aria-labelledby="shipping_address">
  <p id="shipping_address">Shipping Address:</p>

  <label for="shipping_street">Street:</label>
  <input type="text" id="shipping_street"></input>

  <label for="street_number">Number:</label>
  <input type="text" inputmode="numeric" id="street_number"></input>

  <label for="zip_code">ZIP Code:</label>
  <input type="text" inputmode="numeric" id="zip_code"></input>
</div>
```

In this case we will no longer be constrained by the default stylings and HTML structure of a fieldset - as long as we can connect that “group” wrapper with the labeling element (the “Shipping Address” paragraph) - we can structure the HTML and style however we wish.

Hints & Descriptions

Some inputs require additional information alongside them, to give users more context on **how they should fill in the required information**.

For example, a password input might need to show a “hint message” saying something like: “Must be at least 12 characters long”. These kinds of “hint messages” are called **descriptions** of an input. We can add these descriptions using the **`aria-describedby`** attribute.

```HTML
<label for="password">Password</label>
<input type="password" id="password" aria-describedby="password_hint"></input>

<span id="password_hint">Must be at least 12 characters long.</span>
```

Screen readers will read this description **after the label**, but we must programmatically connect it (like we did) in order for it to be read **while the user is focused on the password input field**.

Similarly to `aria-labelledby`, `aria-describedby` can take **multiple IDs as values** separated by the “whitespace” character in order to create a **compound description**.

Error messages & Live regions

Okay, so by now we know how to label and additionally describe our inputs.
This is the most crucial part and one that if done correctly, we will have solved most of the accessibility problems in our forms.

However, we now need to consider what happens when **users enter invalid data**, or **the submission of our form fails**. In such cases, we need to be able to tell any kind of user - including ones that use assistive technologies - that something happened during their filling in/submission of the form.

Now, this kind of information is not immediately present on the form when the users start using it. They will only ever appear when an error happens, obviously. So how do we **notify users that use assistive technologies like screen readers** that an error happened? Sighted users can clearly see our error message appear on the screen, but what about screen readers?

Live regions

Live regions are a **mechanism that allows assistive technologies to announce changes on the page**. Whenever we have some dynamic content that can be added or changed, we want that information to be conveyed to users that rely on screen readers as well.

In order to do this, we can **mark a section on our page where we know the content will change** - as a **live region**. We can do that by either **explicitly adding the `aria-live` attribute on a container-like element** or by **implicitly using certain ARIA roles**.

aria-live

Adding this attribute to an element marks it as a live region. What this means is that **any changes to that element's content will be announced by a screen reader**.

Now, there are 3 ways this change can be announced - each of them representing one of the valid values that the `aria-live` attribute can have:

1. **"assertive"** - The content change will be **announced immediately**, even interrupting the screen reader in what it was announcing at the time of the content change
2. **"polite"** - The content change will be **announced after the screen reader is done announcing** the content that it is already reading
3. **"off"** - The content change will only be **announced if the user has already moved the focus indicator on or within the live region**

Generally we only want to use **`aria-live="assertive"`** for **time-critical and very important announcements** that are crucial for the user to know about immediately.
Otherwise we want to use **`aria-live="polite"`** for cases such as error messages on a form, or a filter selection changing the list of movies below it.

Roles as implicit live regions

Roles like **"alert"** and **"status"** automatically make an element a live region.
For example, **`role="alert"`** makes an element a live region with an **assertive politeness**, meaning changes within it will immediately be announced.

On the other hand, **`role="status"`** will make an element be a live region with a "**polite" setting**, announcing the changes only after the screen reader is done reading whatever it was reading at the time of the change.

There are other roles that can cause this implicit live region creation, which we won't cover during this lesson but you are welcome to read more about them on MDN's documentation on ARIA roles.

Error messages

Now, let's utilize live regions to make sure that **screen readers properly announce the error messages** that dynamically appear on our forms' inputs.
We can add an additional element next to our previous password input that will serve as a live-region container for the potential error message associated with it.

```html
<label for="pwd">Password</label>
<input type="password" id="pwd" aria-describedby="error_msg hint_msg"></input>

<span id="hint_msg">Must be at least 12 characters long.</span>

<span aria-live="polite" id="error_msg">
<!-- Dynamically add an error here through JavaScript -->
</span>
```

Notice that we have **combined the error message and the hint message** to describe our password input**.** This is usually something we want to do - **keep both the hint message and error message visible** because the **hint message can instruct users on how to solve the error**.

We also need to emphasize the **order of the IDs in the `aria-describedby` field**. We are listing the error message first and then the description.
This means that screen readers will read the error message first and the description (hint message) second, when users are focused on the password input.

This is a good practice to follow as it's generally a good idea to **announce the problem first, followed by the instructions on how to solve it**.

Inline vs Summary Error messages

There are essentially two types of error messages on a form:

- **Inline error messages** - messages that appear **next to the input** that they are associated with
- **Summary error messages** - global list of error messages that appear (usually) at the start of the form, before all of the inputs

Depending on your UI's needs, you can choose to implement either one or the other, or sometimes even both - again depending on the type of form and type of errors that you need to show.

Here are some guidelines to follow when implementing either of these error messages:

Inline Error messages

Inline error messages are very good for **showing descriptive errors, often offering a potential solution within the message itself**. They are also **located right next to the input** where the error happened so they are easy to understand.

When using this approach, we need to make sure that we **properly connect the error message to the form input** using `aria-describedby` and also make sure that **the error message is announced by a screen reader** using live regions.

Inline error messages can be used in a couple of different scenarios:

- **As the user is typing in the field** - this is generally not very good UX and can be distracting to some users
- **As soon as the user leaves the field (on "blur")** - also not very good UX as focusing on the next field and then reading the potential error message from the previous field can be confusing and distracting
- **After the form has been submitted** - generally good UX as every error message can be read from top to bottom and they are still next to every form input

Summary Error messages

Summary error messages on the other hand, **always appear after the form has been submitted** and as such, they need to be **grouped within a global live region** so that screen readers can read out all of the issues.

```html
<form>
  <div role="alert">
       <p>There was a problem sending your message!</p>

       <ul>
         <li>First and last name cannot be empty</li>
         <li>You must provide a discount code</li>
       </ul>
  </div>

  <label for="first_name">First name</label>
  <input type="text" id="first_name">

  <label for="last_name">Last name</label>
  <input type="text" id="last_name">

  <label for="discount_code">Discount code</label>
  <input type="text" inputmode="numeric" id="discount_code">
</form>
```

Error messages - Best practices

Before we move on - it is important to talk about some good habits when it comes to constructing accessible error messages.

The first and most obvious thing to do is to **make the error message distinguishable from the rest of the form/inputs/descriptions**.
This however doesn't mean just "make it red". In fact, **we cannot and must not rely on color alone when it comes to error messages**. Users who have trouble distinguishing different colors, colorblind users etc. will not be able to distinguish an error message from the rest of the content within a form - **especially if the error message is not clear enough in its actual content**.

To make sure we have all of our bases covered, it's important that we have **an additional indicator next to the error message** that clearly marks it as one. This is usually a **small error SVG icon**, like an exclamation mark or something similar.

Another thing we need to consider is having **clear and understandable error messages**.
We don't want to overwhelm users with a long explanation on why they failed to enter a valid password - we just want to inform them that an error happened and **ideally we want to have**

a way to guide users on how to solve that problem. This “guidance” can be done **using hint messages** like we saw before, or if an input doesn’t have a hint message - **include a hint on how to fix the error within the error message itself**.

Validation

There are essentially two types of form validation in terms of where that validation happens: **client side validation** and **server side validation**. For security purposes, it is always a good idea for our form validation to **be done on both the client and the server side.**
We will focus on the client side, specifically on the HTML and accessibility side and look at a couple of ways we can validate our forms and also indicate their validity to our users.

Required fields

HTML forms have a built-in validation system, in which we can add all kinds of constraints on our inputs. We can give them **minimum and maximum lengths**, we can make them **adhere to a certain pattern**, but probably the most important constraint is the **`required`** attribute.
This attribute makes sure that a given input **must be filled with valid information**, before it can be submitted.

```html
<form>
  <label for="name">Name</label>
  <input required type="text" id="name" placeholder="Enter your full name...">

  <button type="submit">Submit</button>
</form>
```

If we just tried to submit this form with an empty “name” field - the built-in HTML validation will kick in and tell us that we must enter a value in that field.

Having the **`required`** attribute on an input is the **best way to tell users and assistive technologies** that they cannot skip filling it in. **Screen readers will actually read it out** when the user is focused on a required input. For example, the “name” input above would be announced as: **“Name, required, edit text...”**.

Now, this attribute works amazingly well with **native input elements** such as **input, textarea, select** etc. However, if our form needs a **more complex widget** that acts as an input to some kind of information (ex. A custom dropdown component that works as a **`select`** input) - then setting the `required` attribute on it won't work.

For custom components such as that - we can use the **`aria-required`** attribute. This attribute will also make screen readers read the custom input component as "required", but it **does not trigger any built-in validation logic** (like making sure the input has a valid value before submitting it), which is to be expected since the browser cannot know the logic of our custom input component.

```html
<form>
  <label for="language">Language</label>
  <custom-select aria-required="true" id="language">...</custom-select>

  <button type="submit">Submit</button>
</form>
```

Nevertheless, **we generally don't want to rely on the browser built-in validation anyway** (even with the native `required` attribute) because **it is often very inconsistent across different browsers**.
More often than not, we will be doing our own custom validation logic through our JavaScript, so having the **`required`** attribute on native HTML inputs and the **`aria-required`** attribute on custom input components is more than enough for us to get the correct screen reader announcements out to our users.

In scenarios where we are doing our own custom validation logic, one additional thing we will need to do is to set the **`novalidate`** attribute on the **<form>** element itself, so that we turn off the built-in HTML validation and handle the error messages ourselves.

aria-invalid

This attribute is used to indicate that **an input has been validated and was found to be invalid**. This goes for both native HTML inputs, as well as custom input components.

We need to **add this attribute to every input on our forms** - once they have been validated - and dynamically update the value to either **"true" or "false"** depending on the valid state of the given input.

Submission, Loading & Success

When it comes to submitting a form, we always want to make sure that we have **multiple ways** to do so.
The two common ways are to have a **submit button at the bottom of the form** and also to have **the ability to submit the form by pressing "Enter"** while we are focused on any input inside the form (with the exception of <textarea> - "Enter" means "start a new line").

We can easily enable both at the same time by having our **submit button** have the attribute of **`type="submit"`** and **placing it inside the form's HTML**.

HTML
```html
<form novalidate>
  <label for="email">Email</label>
  <input required type="email" id="email" placeholder="Enter your email...">

  <label for="message">Message</label>
  <textarea type="text" id="message" placeholder="Enter your message...">

  <button type="submit">Submit</button>
</form>
```

There are some situations where our **UI might require that the submit button be outside the form's HTML** and when that happens, we still want the "Enter" key submission to work.
We can do this by **giving the <form> element and ID** and then setting a **`form` attribute on the submit button** with the **value being the ID of the <form> element.**

HTML
```html
<form id="contact-form" novalidate>
  <label for="email">Email</label>
  <input required type="email" id="email" placeholder="Enter your email...">

  <label for="message">Message</label>
  <textarea type="text" id="message" placeholder="Enter your message...">
</form>

<!-- Some other UI here... -->

<button type="submit" form="contact-form">Submit</button>
```

Loading states

Okay, so we submitted our form - what should happen next from an accessibility standpoint? While our form is being submitted, the first thing we actually want to do is to (at least) **disable the submit button** so that we prevent duplicate submissions. Ideally we should also disable all of the inputs in the form as well, but having only the submit button disabled is good enough.

After that, we might want to show a **loading indicator** somewhere inside or next to the form. Now, what we **don't want to do** is just show a generic spinner element and have that be it.

This kind of UI doesn't help users who rely on assistive technologies such as screen readers, because **the loading state is not being announced to them**.

To fix this, we can **have a live region where our spinner will appear** and also add a **visible or visually hidden** "Loading" message inside the live region.

HTML
```html
<form>
<!-- Inputs + Submit button here -->

  <div aria-live="polite">
    <div class="spinner"></div>
    <span>Loading...</span>
  </div>
</form>
```

There are also cases where instead of a spinner, we might need to show a more modern so-called **skeleton animation**. Maybe this is a search form and the content below it needs to be populated by the results of the form submission.

In such cases, we still need to make those skeleton elements accessible by telling assistive technologies that these are basically **placeholder elements** and that content will be filled there shortly.

The way we do that is by setting the **`aria-busy="true"`** attribute on our skeleton element, along with an **`aria-label`** describing what is being loaded (waited on).

aria-busy

The attribute **`aria-busy`** is used to indicate that **an element's content is being modified** and assistive technologies can announce that updated content **once the `aria-busy` attribute's value gets set to "false"**.

HTML
```
<form>
<!-- Inputs + Submit button here -->

  <div aria-busy="true" aria-label="Loading user profile">
    <div class="avatar-skeleton"></div>
    <div class="username-skeleton"></div>
  </div>
</form>
```

Success message

Remember how we made sure that **error messages were announced** by screen readers? Well, we also want to **announce the success message** or if we don't show a success message, **announce the fact that the form was submitted successfully**.

This is probably the most overlooked part of accessible forms, because we put so much attention on the error messages and then forget to announce the success message.

We don't need anything fancy in this case, we can just use yet **another live region that will contain the success message** (whether that message is visible or not) once the form gets submitted successfully.

HTML
```
<form>
<!-- Inputs + Submit button here -->

  <div aria-live="polite">
    <span>Hurray! We got your message.</span>
  </div>
</form>
```

There are admittedly a lot of moving parts when it comes to building forms. If we could condense them into a couple of bullet points that we need to follow, they would look something like this:

- Make sure your **inputs are properly labeled** and always try to have a **visible label**
- Any **descriptions and error messages** need to be **programmatically connected to their associated input**
- Make sure your **error messages, loading states and success messages** are correctly set up within live regions so that screen readers can announce them
- Always go with the **native implementation,** the **native element and attribute** first before implementing your own custom logic, if you can help it

UX Improvements

Now, we can stop here and be sure that our forms are fully accessible, but if we want - we can go a step further. Let's then look at a couple of ways we can **improve the user's experience** while filling in our forms and how we can make the process a whole lot easier for them.

Autocomplete

One of the biggest ways we can help users fill out forms on the Web is by **having browsers understand the meaning of each input** (as much as possible), so that they can use their **autofill** feature to quickly fill out forms on behalf of our users.

The way we do that is by using the mighty **`autocomplete` attribute**.
This attribute conveys a semantic meaning to the input it is assigned on and it has many values that it can contain, each for a different kind of input.

For example, if we have a checkout page where users are supposed to enter their credit card information, it would be very annoying if they needed to enter each field every time they needed to make a purchase.

What we can do instead is add the relevant `autocomplete` attribute to each input field in our form, which will make **browsers suggest previously entered values in those fields** to be autofilled automatically. This autofill functionality can also come from things like **password managers** aside from just the browsers themselves.

```html
HTML
<form>
  <label for="cc_name">Name on card</label>
  <input type="text" id="cc_name" autocomplete="cc-name">

  <label for="cc_num">Card number</label>
  <input type="text" inputmode="numeric" id="cc_num" autocomplete="cc-number">

  <label for="cc_exp">Expiration date</label>
  <input type="text" id="cc_exp" autocomplete="cc-exp">

  <label for="cc_csc">Security code (CVV)</label>
  <input type="text" inputmode="numeric" id="cc_csc" autocomplete="off">

  <button type="submit">Pay now</button>
</form>
```

Apart from just making the entire payment process much faster, this simple addition also makes sure that **users commit fewer errors while submitting our forms**.
Users with cognitive, motor and similar disabilities can more easily fill them out and assistive technologies such as **screen readers have that much more information about the inputs to be able to correctly announce their meaning**.

It's essentially a no-brainer to have this attribute set on every input we have... or is it?

Notice that the **CVV field has an `autocomplete` attribute with the value of "off"**. This is due to the PCI DSS (Payment Card Industry Data Security Standard) that strictly prohibits storing sensitive information like the CVV number of a user's credit card on any kind of device (browser or mobile).
There are also other scenarios where we need to actually turn off autocomplete for our inputs:

- **One-time token/password inputs** that expire after a short amount of time
- **Discount/Promo code inputs** that expire after their usage
- **Search inputs** where we list our own custom suggestions

Remember, while the `autocomplete` attribute is a very powerful tool, it is also crucial to understand when and where it can be used so that it is utilized correctly. Check out the MDN documentation on this attribute for more detailed information.

Input types & keyboard layouts

One of the simplest yet most impactful ways to improve our forms' accessibility is by **using the correct input types and keyboard layouts**. This helps browsers understand **what kind of data is expected for a particular input** and allows them to **provide the most appropriate keyboard layout** for users, especially on mobile devices.

The `type` attribute

This attribute tells browsers and assistive technologies **what kind of data an input expects**. The common values this attribute accepts are **`text`, `email`, `password`, `tel`,** and **`number`**. Choosing the right `type` also **triggers the appropriate virtual keyboard on mobile devices** and even enables **browser features like autocomplete and validation**.

```html
<form>
  <label for="email">Email</label>
  <input type="email" id="email" autocomplete="email">

  <label for="pwd">Password</label>
  <input type="password" id="pwd" autocomplete="current-password">

  <button type="submit">Log in</button>
</form>
```

The `inputmode` attribute

In addition to the `type` attribute, the `inputmode` attribute gives us **more fine-grained control over which keyboard layout is displayed**, along with controlling which characters can be typed into a given input element.
This is particularly useful when the `type` attribute doesn't quite match the keyboard layout we want to show our users.

Some common values for the `inputmode` attribute include:

- **`numeric`** - Keyboard layout will be **only digits (0-9)**
- **`decimal`** - Keyboard layout will be **digits plus the decimal point**
- **`tel`** - Keyboard layout will be a **telephone keypad (0-9, *, #)**
- **`email`** - Keyboard layout will be **optimized for emails ("@" and ".com" keys)**
- **`url`** - Keyboard layout will be **optimized for URLs**

A neat trick for numeric inputs is to have an input with **`type="text"`** and **`inputmode="numeric"`** so that we don't show those little "spinner" arrows at the end of the input, but still restrict users from entering any unexpected numbers like "1e4".

We can use **`type="number"`** in situations where we need **inputs that signify some kind of quantity (age, price, count etc.**)

```html
HTML
<form>
  <label for="zip">ZIP Code</label>
  <input type="text" id="zip" inputmode="numeric" autocomplete="postal-code">

  <label for="houses">Max number of houses</label>
  <input type="number" id="houses">

  <button type="submit">Search</button>
</form>
```

Conclusion

As we have seen throughout today's lesson - there are so many things to consider when constructing forms on the Web, especially when we want to make them fully accessible.

However, **you are now equipped with the knowledge you need to tackle these challenges** and direct your worries on the business logic, animation and styling of the forms - because you will have the accessibility part covered.

Try to take these concepts we covered and **apply them to your previous projects** and see if there are some things you may have missed or haven't even thought about when you first developed them - you might be surprised by how easy it is to not have everything covered the first time around.

Further learning

This section is dedicated to additional topics that dive deeper into more nuanced use cases when it comes to accessible forms on the Web. You can check it out on your own time and if you have any questions, don't hesitate to contact me at daniel.shijakovski@testdevlab.com.

Labeling

A good use case for labeling that's not immediately obvious within the UI (ex. The label is not directly next to the input) is when we have **tables** which have a **column that contains inputs in the cells**. In that case, we need to **connect that column's header with each input using `aria-labelledby`** so that the inputs can be properly labeled.

```html
<table>
  <thead>
    <tr>
      <th aria-hidden="true"></th>
      <th scope="col" id="col-price">Price</th>
      <th scope="col" id="col-qty">Quantity</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th scope="row" id="row-apples">Apples</th>
      <td><input type="number" aria-labelledby="col-price row-apples"></td>
      <td><input type="number" aria-labelledby="col-qty row-apples"></td>
    </tr>

    <tr>
      <th scope="row" id="row-oranges">Oranges</th>
      <td><input type="number" aria-labelledby="col-price row-oranges"></td>
      <td><input type="number" aria-labelledby="col-qty row-oranges"></td>
    </tr>
  </tbody>
</table>
```

The above table's HTML would visually look something like this:

	Price	**Quantity**
Apples	*<Price input>*	*<Quantity input>*
Oranges	*<Price input>*	*<Quantity input>*

That empty header cell has **`aria-hidden="true"`** set on it, so that assistive technologies can just skip/ignore it.

We are also using the **`scope`** attribute on the vertical and horizontal headers to tell assistive technologies: **“These headers apply to the columns below them (scope=”col”)”** and **“These headers apply to the rows to the right of them (scope=”row”)”**, respectively.

Finally, we are combining the headers from above and to the left of the input cells using the **`aria-labelledby` attribute** which if you didn’t know - can use **multiple elements’ IDs** to create a **compound label**. We do this by just adding the IDs in the order that we want them to be read out by assistive technologies, **delimited by the “whitespace” character**.

Placeholders are NOT labels

An important rule about labeling inputs is that we should **avoid relying on the placeholder attribute to act as the input’s label**.
Placeholders often have **insufficient color contrast** but most importantly - **they disappear once the user starts typing in the input field**. If we don’t have a visible label next to the input, users will get confused and may forget what information they need to fill in, especially if they are dealing with a large form with many inputs.

There are automated accessibility scanners out there that will actually **treat an input that has no visible or invisible label, but has a placeholder - as a validly labeled element** - because the placeholder becomes a fallback label on some browsers. This can make your automation tests lie to you and tell you that all is good in your forms, when in reality you have a major accessibility issue on your hands.

UX improvements

Better summary error messages

Talking about summary error messages, the implementation we covered only makes sure that all of the error messages related to the form’s submission are correctly announced by screen readers. This is good, however the second part of having an error is actually fixing it. We can make this process much easier on users by modifying our implementation in two ways:

The first thing is - **once the form is submitted** and we show the summary error messages - **move focus on the live region that contains the error messages**. This helps keyboard users to navigate to where the error messages are located.
We can of course do this with **`tabindex=“-1”`** by programmatically focusing the summary live region when an error happens upon our form’s submission.

The reason this is helpful is related to the second thing we need to do and that is - modify the actual error message elements to be **enclosed within anchor <a> elements**. Those anchor elements can then be linked to the **ID of the input** that each error message is associated with.

Having this setup, our users can easily both read out the error messages and quickly navigate to each input that an error message is referring to.

HTML
```html
<form>
  <div role="alert" tabindex="-1">
    <p>There was a problem sending your message!</p>

    <ul>
      <li><a href="#first_name">First name cannot be empty</a></li>
      <li><a href="#last_name">Last name cannot be empty</a></li>
      <li><a href="#discount_code">You must provide a discount code</a></li>
    </ul>
  </div>

  <label for="first_name">First name</label>
  <input type="text" id="first_name">

  <label for="last_name">Last name</label>
  <input type="text" id="last_name">

  <label for="discount_code">Discount code</label>
  <input type="text" inputmode="numeric" id="discount_code">
</form>
```

Notice how now we have **one error message per input**. This is better because now we can easily **associate each error message with exactly one input** and make the navigation to it much easier. If however our error messages referred to multiple inputs, this kind of setup might not be ideal for us.
Again, it all depends on the use case that you have and these are just “tools in your toolbelt” to have when such a problem presents itself.