Framer Motion Introduction

It's the open-source animation library that powers the animations within Framer, but it's absolutely intended to be used in real-world production websites and web applications.

In short, using `motion` in React components allow smooth CSS transitions interpolation between state. You can transition `opacity` from `0.5` to `1`, or `width` from `200px` to `1000px`.

Without using motion, you can't transition layout (position) properties.

You cannot animate the DOM tree changes, like an element getting moved from one parent to another, or just reordering them in the DOM. See **FLIP technique** explained below

What are the advantages?

Animate Shared Layouts: One of the standout features of Framer Motion is its ability to animate shared layouts. As you mentioned, animating changes in the DOM tree, like moving an element from one parent to another or reordering elements, can be challenging. Framer Motion provides a `AnimateSharedLayout` component that makes these types of animations straightforward.

Variants: A feature allows you to define a series of animations and transitions as "variants" on a component. This makes it easy to switch between different animation states without boilerplate code.

Orchestration: Framer Motion provides tools to sequence animations, ensuring that they play in the desired order or in response to certain events.

Performance: Framer Motion uses the Web Animations API under the hood, which is optimized for performance. This means animations run smoothly, even on devices with limited resources.

Server-Side Rendering (SSR) Friendly: Framer Motion is compatible with server-side rendering, which is important for many React applications that prioritize SEO or initial load performance.

SSR compatible

No Window or Document Dependencies: Libraries that rely on the `window` or `document` objects directly can break when used with SSR because these objects are not available in a Node.js environment. Framer Motion is built with this in mind and avoids such direct dependencies, ensuring that it doesn't throw errors during server-side rendering.

Initial Static Rendering: When your React component tree is rendered on the server, Framer Motion components will output their initial static state. This means that any animations or transitions won't be applied until the JS is executed in the browser, ensuring initial HTML is still sent to the browser.

Hydration Compatibility: "Hydration" is the process where the static server-HTML content rendered by the server becomes dynamic in the browser, with React attaching additional behaviors. Framer Motion is built to be compatible with this process. Once the client-side React code takes over, Framer Motion components can start their animations and interactions without any issues.

No Flash of nustyled content: One frequent problem with SSR and animations is the "flash of unstyled content" (**FOUC**), where there's a moment when the content appears unstyled or in an unintended state before the client-side JS kicks in. Framer Motion handles this gracefully, ensuring that animations and transitions appear as intended without any jarring visual glitches.

Optimized Bundle Size: While this isn't exclusive to SSR, it's worth noting that Framer Motion is optimized to have a minimal impact on your bundle size. This ensures that the client-side JS that needs to be downloaded and parsed is as small and efficient as possible, which is crucial for performance, especially in SSR scenarios where initial load times are a focus.

Ease of Integration with SSR Frameworks: Framer Motion can be easily integrated with popular SSR frameworks and solutions in the React ecosystem, such as Next.js. This means developers don't have to jump through hoops or apply special configurations to get motion working with their SSR setup.

Getting Started Example

When `isEnabled` is `true`, we want to move the ball down by sixty pixels. The first step is to import the `motion` utility at the top of the component file.

```
import React from 'react';
import { motion } from 'framer-motion';
import './GettingStarted.css';

export default function GettingStarted() {
  const [isEnabled, setIsEnabled] = React.useState(true);

  return (
    <>
      <motion.div
        className='shadow-sm yellow ball '
        initial={false}
        transition={{
          type: 'spring',
          stiffness: 200,
          damping: 25,
        }}
        animate={{
          y: isEnabled ? 60 : 0,
        }}
      />
      <button onClick={() => setIsEnabled(!isEnabled)}>Toggle</button>
    </>
  );
}

```

Then, we need to swap out the element we wish to animate with its `motion` counterpart. Instead of rendering a `<div>`, we'll render a `<motion.div>`:

```
<motion.div className="yellow ball" />
```

`motion.div` is a component that wraps around a primitive `<div>`. It does everything a `<div>` can do, and it'll forward along any attributes, being props you set. `motion` can wrap around any HTML tag. If I render a `<motion.button>`, it'll produce an HTML `<button>`.

Using proxies (like we saw with **Immer**), Framer Motion can intercept the property accessor. It then passes the property along to the `motion` component, and it uses that as the tag. Framer Motion wrapper components give us a couple of new super-powers. We can apply an animation with the `animate` prop:

When `isEnabled` is `false`, the ball moves down by 60px, applying a transform, `translateY(60px)`.

By default, Framer Motion uses *spring physics.* This produces much more fluid, life-like animation compared to the Bézier curves we get with CSS transitions.

We can customize it using the `transition` property. For example, here's a Bézier curve:

```
<motion.div
  className='example'
  transition={{ type: 'tween', ease: 'easeInOut' }}
  animate={{ y: isEnabled ? 0 : 60 }}
/>;
```

Instead, its strongly recommend sticking with spring physics.

Next the `animate` prop includes an **enter** animation out-of-the-box. It animates to the initially specified value. In many situations, this is an experiance wanted, but it won't always be what we need.

We can disable this with the `initial` prop:

```
<motion.div
  initial={false}
  className='example'
  transition={{ type: 'tween', ease: 'easeInOut' }}
  animate={{ y: isEnabled ? 0 : 60 }}
/>;
```

This is the typical flow with Framer Motion!

We import our `motion` utility, swap out the DOM node(s) we want to animate for their `motion.x` counterparts, tweak the transition with the `transition` prop, and customize the animation.

The `animate` prop is used when we have specific values, or we can use the `layout` prop.

Layout Animation

Framer Motion's ability to handle complex animations especially involving `layout` changes is a big advantage, and it is rooted in its underlying design and techniques it employs. Under the hood when animating **layout** changes, why does it appear smoother compared to manual attempts?

When you change the layout, `position`, size like `width` and `height`, or other properties of an element, the browser immediately re-renders the element in its new state. This sudden change is jarring because there's no transition between the initial and final states.

FLIP technique

Framer Motion, using the **FLIP**, creates a smooth `transition` between these states.

Here's a more detailed breakdown of what happens:

Capture Initial State (First): Before any changes are applied, Framer Motion captures the `initial` position and size values of an element. This is the "**snapshot**" of where the element starts.

Apply Changes (Last): A `layout` changes (like a new `position` or `width`) are applied to the element. Framer Motion captures the final and ending state of the element to `animate` toward.

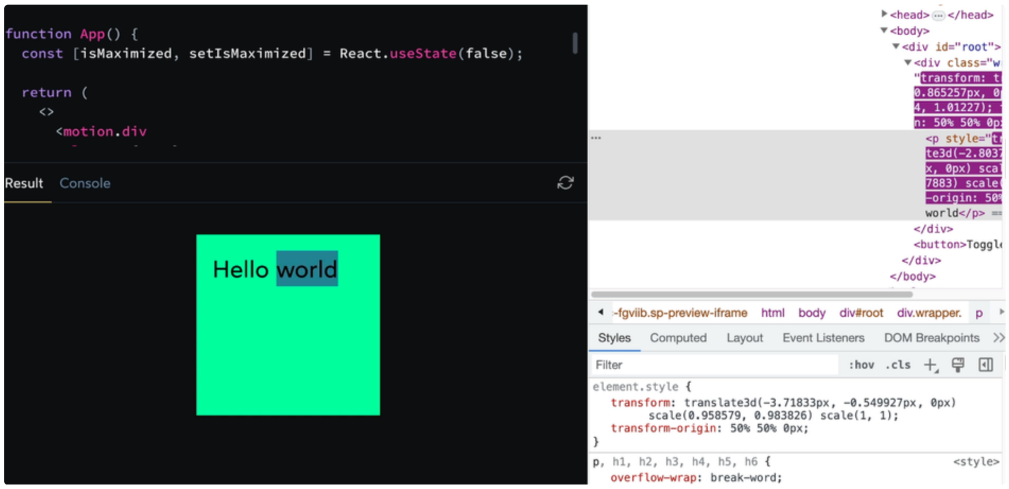
Calculate Differences (Invert): It calculates differences in `position` and size between `initial` and final states to `animate`. A "**delta**" represents how much an element moved & resized.

Override Immediate Changes: Instead of letting the browser immediately re-render the element in its new position and size, Framer Motion uses CSS `transforms` to visually "**override**" the changes. It positions the element back to its `initial` state using the `translate` transform for `position` and the `scale` when it `transforms` for size.

Animate Transition (Play): It animates applying alot of now needed and calcuated `transform` properties from the initial state "**overridden**" to the final ending state.

A smooth transition as the element moves and resizes to its final position and size.

Inspecting an animating element in DevTools you'll see many `transform` properties being dynamically adjusted. The `translate` values handle the movement, and the `scale` values handle the resizing. These properties are **GPU-accelerated**, which means they're optimized for performance and create smoother animations compared to animating properties like `width`, `height`, or `top`, `left` directly.



```
function App() {
  const [isMaximized, setIsMaximized] = React.useState(false);

  return (
    <>
      <motion.div
```

Result
Console
Hello world

```
▶<head>…</head>
▼<body>
  ▼<div id="root">
    ▼<div class="w
      "transform: t
      0.865257px, 0
      4, 1.01227);
      n: 50% 50% 0p
        <p style="t
        te3d(-2.8037
        x, 0px) sca
        7883) scale
        -origin: 50
        world</p> ==
      </div>
      <button>Toggl
    </div>
  </body>
```

-fgviib.sp-preview-iframe html body div#root div.wrapper. p
Styles Computed Layout Event Listeners DOM Breakpoints
Filter :hov .cls

```
element.style {
  transform: translate3d(-3.71833px, -0.549927px, 0px)
      scale(0.958579, 0.983826) scale(1, 1);
  transform-origin: 50% 50% 0px;
}
p, h1, h2, h3, h4, h5, h6 {                    <style>
  overflow-wrap: break-word;
```

While the actual `layout` properties (like `position`, `width`, `height`, `top`, `left`) have changed immediately, Framer uses the `transform` property to create a visual illusion of a smooth transition between the old and new states of your React element. This approach is more performant visually and much smoother than trying to animate layout properties directly in your CSS.

The smoothness you observe with Framer Motion is primarily due to its intelligent use of the `transform` property on your React components, combined with the **FLIP** technique's mechanics.

Additionally, it also provides the following that complements the FLIP technique

Declarative API: Framer Motion's API is designed to be declarative, which aligns well with the React paradigm. Instead of specifying how an animation should progress step by step, you define the start and end states, and Framer Motion handles the rest. Abstracts complexity in creating animations.

Optimized Performance: It's built on top of Web Animations API, which is optimized for performance. Ensures animations run smoothly, even when multiple elements animate simultaneously.

Intelligent Defaults: Framer Motion comes with intelligent default settings for its animations. These defaults are based on real-world use cases and a good balance in performance and visual appeal.

Complexity Handling: When animating properties like `position` or `transform`, there are many edge cases and complexities to consider. Framer Motion has been designed to handle these complexities, ensuring that animations are consistent and smooth across different scenarios.

State Management: Framer Motion manages the state of animations internally. This means it can handle interruptions, reversals, and other changes in animation state gracefully.

Nested Animations: As you will see below, with the `.wrapper` and `.maximized` classes, certain animations can distort child/nested elements. Framer Motion provides tools to handle nested animations, ensuring that child elements animate correctly in relation to their parents

Here we explore how to use **layout** animations to animate a green box. Here we use `layout` animations to animate the box that gets "`maximized`" by a toggling class.

```css
.wrapper {
  width: 180px;
  height: 180px;
  /* Cosmetic styles omitted for brevity */
}

.maximized {
  position: absolute;
  inset: 0;
  width: revert;
  height: revert;
}

```

By default, switching from one `position` layout to another can be jarring in CSS transitions.

Unfortunately, we can't animate such changes.

We're not merely transitioning properties, we're swapping entire properties! being the `width`, `height`, and `position` of the element to be **absolutely positioned**.

You can address this using Framer Motion's `layout` animation feature.

```jsx
import { motion } from 'framer-motion';

function App() {
  const [isMaximized, setIsMaximized] = React.useState(false);
  return (
    <>
      <motion.div
        layout={true} // 👈 here!
        className={`wrapper ${isMaximized ? 'maximized' : ''}`}
      />
      <button onClick={() => setIsMaximized(!isMaximized)}>
        Toggle
      </button>
    </>
  );
}
```

The `layout` prop instructs Framer Motion to animate any layout changes, which being an elements `position`. It employs a modern take on the **FLIP** technique. This technique calculates the difference between the initial and final states and uses a CSS `transform` to **transition** between them.

We customize the transition in the component's `transition` prop:

```jsx
<motion.div
  layout={true}
  transition={SPRING}
  className={`wrapper ${isMaximized ? 'maximized' : ''}`}
/>
```

Warning: Framer Motion has a steep learning curve.

It's not as simple as adding a `layout` prop and having everything work. If you try to use this in your own projects, you'll run into some weird quirks / things not working the way you'd expect.

```
const SPRING = {
  type: 'spring',
  stiffness: 200,
  damping: 40,
};
```

One common issue arises when the animated element contains content. For instance, adding the text "**Hello world**" can lead to distortions due to how CSS transforms work.

Nested and child Layout changes

The solution is to address how the nested elements `layout` transform is adapting to its parent.

```
<motion.div
  layout={true}
  transition={SPRING}
  className={`wrapper ${isMaximized ? 'maximized' : ''}`}
>
  <motion.p layout="position">
    Hello world // 👈🏻 here
  </motion.p>
</motion.div>
```

The `layout` prop can take values that each assist with its own specific behavior:

`layout={true}`:

- This is the most common usage. When set to `true`, Framer Motion will automatically animate both the `position` and `size` of the element if they change between renders. This is useful for elements that might change in both, like items in a responsive grid that reflows based on window size.

`layout="position"`:

- When set to "`position`", motion will only animate the position (`translateX` & `translateY` within its `transform` changes) of the element, if it changes between renders, but not size. Useful for scenarios where the size of the element remains constant, but its `position` might change.
- For example, list items that might change order but not size.

`layout="size"` (**less common**):

- When set to "`size`", motion will only animate the size (`scaleX` and `scaleY` in the `transform`) of the element if it changes between renders, not its position. Useful for elements that might change size but remain in the same position, remain in flow.
- Like a responsive image or a content box that adjusts based on its content.

On the parent, we center the text element within the parent box. We can use `text-align: center`. However, that is not going to really `transform` well and so this can lead to unexpected results.

A better approach is to use positioning like `display: flex`.

```
.wrapper {
  display: flex;
  justify-content: center;
  align-items: flex-start;
}
```

Lastly, on the child!

We ensure that `transition` settings are consistent between parent and child elements that are experiencing this `layout` shift by transforming on `position` change (transform).

Of course, we handle this by using the `layout` prop in both locations:

```
<motion.div
  layout={true} // 👈 here
  transition={SPRING}
  className={`wrapper ${isMaximized ? 'maximized' : ''}`}
>
  <motion.p
    layout="position" // 👈 here
    transition={SPRING}
  >
    Hello world
  </motion.p>
</motion.div>
```

Key takeaways

We use the `layout` prop to enable layout animations for `motion` components.

`layout` can be set to `"position"`, `"size"`, or `true` (for both position and size).

Layout animations use **CSS transforms**, which essentially treat the element as if it was an ***image***. This can cause distortions, if the element contains text or other elements. We fix this by ***nesting*** `motion` components, to "**cancel out**" those transformations.

Framer Motion uses the bounding box for all elements (`box-sizing: border-box`). It doesn't know where the individual characters are within a paragraph.

To avoid issues, we should "shrinkwrap" elements around their characters.

Transition settings aren't inherited to your nested children so be sure to copy the `transition` prop from the parent to the child, so that they both use the same spring parameters.

Inside Framer Motion's Layout Animations - Matt Perry

FLIP Your Animations

Shared Layout

Shared layout animations revolve around the idea of "**sharing**" visual continuity and flow between two components, even if they aren't rendered at the same time.

By assigning the same `layoutId` to two different components, you're telling Framer Motion that these components are visually linked and should transition smoothly between each other.

Does layoutId require elements to be within a specific parent component configuration?

Yes, to make use of the `layoutId` for shared layout animations in Framer Motion, the components involved need to be descendants of an `AnimateSharedLayout` wrapper.

This wrapper informs Framer to watch for changes in components with matching `layoutId` values and to animate between them when changes occur.

```
import { AnimateSharedLayout, motion } from 'framer-motion';

function MyComponent() {
  return (
    <AnimateSharedLayout>
      {condition ? (
        <motion.div layoutId="shared-id">
          {/* Content A */}
        </motion.div>
      ) : (
        <motion.div layoutId="shared-id">
          {/* Content B */}
        </motion.div>
      )}
    </AnimateSharedLayout>
  );
}
```

- The `AnimateSharedLayout` component wraps around the two `motion.div` components.
- Both `motion.div` components have the same `layoutId` (called "`shared-id`").
- Based on the `condition`, one of the `motion.div` components will render.
- When the `condition` changes and causes a different `motion.div` to render as truthly, Framer Motion will animate the transition between them due to the shared `layoutId`.

Important, that the `AnimateSharedLayout` wrapper doesn't need to be the direct parent. It can be an ancestor higher up in the component tree. If the components within your `layoutId` are descendants of `AnimateSharedLayout`, the shared layout animations will work.

How It Works

Matching `layoutId`: The magic starts when two components have the same `layoutId`. This identifier tells Framer Motion that these components are related in terms of layout and should be animated in a way that suggests they're the same element. Considering React components can be instantiated multiple times, consider using the `useId` hook for your parenting container.

Exit & Enter Animations: When a new component (with a `layoutId`) enters the DOM and another component with the same `layoutId` exits, Framer Motion will create a smooth transition between the two. The entering component will seem to "**morph**" from the exiting one, preserving visual flow.

Crossfade: If both components (with the same `layoutId`) are present in the DOM simultaneously, Framer Motion will perform a `crossfade` between them. This means the old component will fade out while the new one fades in, at the same position and size, creating a seamless transition.

How does shared layout animations work under the hood

Identifying Elements:

Framer Motion uses the `layoutId` to identify which components should be linked in terms of layout. When one component with a specific `layoutId` disappears and another with the same `layoutId` appears, Framer Motion recognizes that these two should be animated together.

Capturing Initial State:

Before any animations occur, Framer Motion captures the `initial` layout (position and size) of the disappearing ("`exit`") element. Including position, dimensions, and any relevant layout properties.

Capturing Target State:

Similarly, it captures the target `layout` of the appearing (or "`enter`") element.

Creating a Clone:

Framer Motion then creates a clone of the `exit` element and places it in a portal or an **overlay** layer, ensuring its above other content. This clone is used to perform the animation, allowing the real `exit` and `enter` elements to be hidden or shown without transition.

Applying CSS Transforms:

Using CSS transforms, the clone is raised from the `initial` layout to the target layout. This involves adjusting properties in the `transform` like `translate` (for position) and `scale` (for size) to make the clone appear as if it's **morphing** from the exit element to the enter element.

Cleaning Up:

Once the animation is complete, the clone is removed, and the enter element is visible in its final state.

Handling Nested Elements:

If there are nested elements with `layoutId` props inside the main components, Framer Motion will also animate these. This allows for more complex animations where not only the parent but also the children `animate` between states.

Crossfade:

If both the exit and enter elements are present simultaneously, Framer Motion will **crossfade** between them. This involves fading out the exit element while fading in the enter element, all while performing the layout animation on the clone.
By simply providing a `layoutId` and wrapping components in `AnimateSharedLayout`, you can achieve intricate animations that would be extremely challenging to implement manually.

Use cases for using shared layout animation

Page transitions: When navigating between pages or views in a single-page application, shared layout can help animate the transition, making it feel like elements from the old page are transforming into elements of the new page.

List reordering: If items in a list can be reordered, shared layout can animate items smoothly to their new positions when the order changes.

Modal animations: When opening a modal or a detail view from a smaller element, shared layout can make it appear as if the smaller element is expanding into the larger view.

Gallery or carousel: When transitioning between images or items in a gallery, shared layout can create smooth animations as one item transitions to the next.

Putting it all together

The `layoutId` prop is the key to shared layout animations in Framer Motion.

By giving multiple `motion` components the same `layoutId`, you're telling Framer to treat them as if they were the same element during animations. Creating an illusion of a single element moving around, even though there are multiple elements in the DOM.

```
{hoveredNavItem === slug && (
  <motion.div
    layoutId={id}  // 👈 Shared layout identifier
    className="hovered-backdrop"
  />
)}
```

It's important to ensure that the `layoutId` is unique in the context of the animation. The `React.useId()` hook can be used to generate a unique ID for each instance of the component.

```
const id = React.useId();
```

Lastly, you may want to `animate` additional properties to enhance the shared layout animation like the border radius. Don't do that directly in CSS, otherwise Framer will not be able to control those attributes. Include `borderRadius` in your `animate` to ensure it looks correct during transition.

```
{hoveredNavItem === slug && (
  <motion.div
    layoutId={id}
    className="hovered-backdrop"
    animate={{
      borderRadius: 32, // 👈 Additional animation
    }}
  />
)}
```

Remember, by default framer will `animate` elements when they first mount. This can be disabled using the `initial` prop in your `motion` component.

```
{hoveredNavItem === slug && (
  <motion.div
    layoutId={id}
    className="hovered-backdrop"
    initial={false} // 👈 Disabling default enter animation
    animate={{
      borderRadius: 32,
    }}
  />
)}
```

Shared layout animations can sometimes cause layering issues.

Especially when multiple elements with the same `layoutId` are present. Dynamically adjusting `z-index` can help ensure elements appear in the correct order.

```
<li
  key={slug}
  style={{
    zIndex: hoveredNavItem === slug ? 1 : 2,  // 👈 Dynamic z-index
  }}
>
  {/* ... */}
</li>
```

Do we need to apply the layout prop?

No, as `layout` is only regarding the `transform` of position and size attributes of a parent and nested children elements, within Layout animation. Here is the distinction between the two props.

`layout` prop:

- The `layout` prop on a `motion` component is used to animate that component's layout changes, such as position and size. When React re-renders and the layout of that component has changed, Framer Motion will animate the transition from the old layout to the new one.
- This prop doesn't require the component to be wrapped in `AnimateSharedLayout`. It works on its own to animate changes in layout for that specific component.
- As mentioned, it deals with the `transform` of position and size for the element it's applied to.

Use `layout` prop when you want a single component to animate its own layout changes.

`layoutId` prop with `AnimateSharedLayout`:

- The `layoutId` prop, in conjunction with the `AnimateSharedLayout` wrapper, is used for shared layout animations between two different components.
- It creates the illusion of a single, continuous element even as the actual DOM elements change.
- This requires the components to be descendants of `AnimateSharedLayout` to work.
- It's about creating a visual link between two separate components, making it appear as if one component is morphing into another.

Use `layoutId` prop (and wrap in a `AnimateSharedLayout`) when you want to animate between two separate components, making them appear as a single element.

In summary, shared layout animations in Framer Motion revolve around the `layoutId` prop.

By giving multiple elements the same `layoutId`, you can create the illusion of a single, flowing element even as the actual DOM elements change. This is combined with other animation properties and techniques to ensure the transition looks smooth and correct.

Layout animations | Framer for Developers

Shared Layout Groups

Group animations in Framer Motion allow for the creation of fluid animations between components that might not be direct siblings or even in the same part of the React tree.

This is achieved using `layoutId` prop to link two components, making them appear as if they are the same component transitioning between two states.

Initially, you may attempt to animate using a basic `layout` animation.

This would animate the position and size of the widgets within their respective `div` container that being (`inbox` and `outbox`), but it wouldn't create a smooth transition **between** the two containers.

```
export default function BoxProcessor({ total }) {
  // State to track the number of boxes that remain in the inbox
  // Then we keep track of the number moved to the outbox container
  const [unmovedCount, setUnmovedCount] = React.useState(total);
  const movedCount = total - unmovedCount;

  // Function to decrease the count of unmoved boxes (moving them to the outbox)
  const handleMoveBox = () => {
    if (unmovedCount > 0) setUnmovedCount(unmovedCount - 1);
  };
  // Function to increase the count of unmoved boxes (reverting them back to the inbox)
  const handleRevertBox = () => {
    if (movedCount > 0) setUnmovedCount(unmovedCount + 1);
  };

  return (
    <main className='flex flex-col gap-10'>
      {/* INBOX CONTAINER - Displays boxes that haven't moved yet */}
      <div className='inbox'>
        {range(unmovedCount).map((itemNum) => {
          console.log('inbox', range(unmovedCount));
          return (
            <div key={itemNum} className='box'>
              {itemNum}
            </div>
          );
        })}
      </div>

      {/* CONTROLS - Buttons to move or revert boxes placement */}
      <div className='actions'>
        <button onClick={handleMoveBox}>
          <ChevronDown />
        </button>
        <button onClick={handleRevertBox}>
          <ChevronUp />
        </button>
      </div>

      {/* OUTBOX CONTAINER - Displays boxes that have been moved */}
      <div className='outbox'>
        {range(movedCount).map((itemNum) => {
          console.log('outbox', range(movedCount));
          return (
            <div key={itemNum} className='box'>
              {itemNum}
            </div>
          );
        })}
      </div>
    </main>
  );
}
```

You introduce the `layoutId` prop to link the flex items called (`widget`) within their `inbox` and `outbox`. This tells Framer that a `widget` disappearing from the `inbox` and appearing in the `outbox` is the same element, a single flowing element. And it should animate the transition between the two states.

```
<motion.div
  layoutId={itemNum}
  key={itemNum}
  className="box"
>
```

⚠ **Warning**, using `itemNum` as `layoutId` is a problem as it's not a **globally unique identifier**. Each element that we are tracking will need this to be globally unique. We must make sure that every element that moves has a consistent and unique identifier.

You not going to get a good result if the items in the target container share the same `layoutId`.

To fix this, you should ensure that the `layoutId` values are unique across both containers. One way to do this is by using the `range(unmovedBox, total)` for the "`outbox`" container. This ensures the `itemNum` values (and thus `layoutId` values) for the "`outbox`" start where the "`inbox`" left off.

```
{/* OUTBOX CONTAINER - Displays boxes that have been moved */}
<div className='outbox'>
  {range(movedCount).map((itemNum) => {
    console.log('outbox', range(movedCount));
    return (
      <div key={itemNum} className='box'>
        {itemNum}
      </div>
    );
  })}
</div>
```

Using `range(unmovedBox, total)` for "`outbox`" container, we ensure that the boxes have a unique `layoutId` values and don't overlap with the "`inbox`" boxes.

Framer will now correctly animate the boxes between the two containers.

But this is not perfect :(

Because the `layoutId` for the zero index within the array conditions `false` as the number `0` is a false value. Resulting in the first element of the array, without shared animation.

Globally Identifiers

Ensure that the `layoutId` unique across the entire application with React `useId` hook. It generates a unique ID for each element, importantly for each instantiated instance of the component.

We then combine this ID with the `itemNum` to create a globally unique `layoutId`.

```
const id = React.useId();
const layoutId = `${id}-${itemNum}`;
```

It's crucial to note that the `layoutId` and `key` should be the same to avoid unexpected behaviors. This ensures that the React **reconciliation** process and Framer Motion's animation logic align.

Here we can assist with the understanding by performing an array `split` per-render with the required JSX for `inbox` and `outbox` containers based on the `unmovedCount`.

```
export default function BoxProcessor({ total }) {
  const id = React.useId();

  // State to track the number of boxes that remain in the inbox
  // Then we keep track of the number moved to the outbox container
  const [unmovedCount, setUnmovedCount] = React.useState(total);
  const movedCount = total - unmovedCount;

  // ... (rest of the code)

  // Generate a consistent list of boxes, each with a unique layoutId
  const boxes = range(total).map((itemNum) => {
    const layoutId = `${id}-${itemNum}`;
    return (
      <motion.div layoutId={layoutId} key={layoutId} className='box'>
        {itemNum}
      </motion.div>
    );
  });

  // Split the boxes between "inbox" and "outbox" arrays
  const inboxArray = boxes.slice(0, unmovedCount); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ,10 ,11]
  const outboxArray = boxes.slice(unmovedCount); // [] 👈 at first render

  // ... (rest of the code)
}
```

<LayoutGroup /> component

When multiple items move simultaneously, they need to be calculated as a **group** not individually. A `LayoutGroup` component helps manage shared layout animations for its `children`.

This ensures that animations are calculated, coordinated and smooth. However, `children` themselves need to have a `layoutId` for the shared layout animations to work correctly.

A `layoutId` is what Framer Motion uses to determine which components are "the same" across different renders or component trees, and thus should be animated from one state to another.

When you have multiple elements that animate simultaneously due to layout changes, `LayoutGroup` ensures that these animations are coordinated and smooth together as a group.

This tells Framer to treat the animations within this group as interconnected.

```
<LayoutGroup>
  {/* All children that need to be grouped together */}
</LayoutGroup>
```

Why is LayoutGroup beneficial?

Coordinated Animations: When multiple elements animate simultaneously due to layout changes, `LayoutGroup` ensures that these animations are coordinated.

This is especially useful when you have a **group** of elements that need to animate together.

Performance: By grouping elements together, `LayoutGroup` can optimize the animations for performance, ensuring smoother transitions.

Visual Consistency: It ensures that the animations between the children of the `LayoutGroup` are consistent in terms of timing and easing.

Regarding the placement of `LayoutGroup`:

If only the `children` items (boxes) are moving and animating, then wrapping just those elements with `LayoutGroup` would be more appropriate.

If there are other elements outside of it that might also be involved in the layout animation, then wrapping the entire component might be necessary.

Does LayoutGroup require each child to have a unique layoutId?

Yes, for shared layout animations to work, each child that you want to animate should have a unique `layoutId`. This is how Framer Motion knows which component in the previous render corresponds to which component in the next render it is currently calculating.

Does any component need to have a layout prop?

The `layout` prop is used to tell Framer to automatically `animate` the `position` and size being `width` and `height` and `scale` of the component when they change (`transform`). This is useful for simple layout animations where you don't need to coordinate between different components.

Shared layout animations (where you are animating between different components or across sections of a component tree), use `layoutId` instead (or as an addition to `layout`).

If a component has a layoutId but not a layout prop?

It will still participate in the shared layout grp animations because of the `layoutId` prop, but it won't animate its **own** position or size (`transform`) handled by the `layout` prop for that actual element within its individual layout changes outside of those shared animations.

In summary, shared layout animations in Framer Motion creates fluid transitions between components. By using the `layoutId` prop and ensuring globally unique identifiers, you can create animations that give the illusion of a single, continuous element even as the actual DOM elements change.

`LayoutGroup` further enhances this by allowing for interconnected animations in a **group** of components. But requires each child to have a `layoutId` assigned to it.

A component with a `layoutId` will participate in shared layout animations. This means that if another component with the same `layoutId` is present in a different part of the component tree (or replaces the original component), Framer will animate the transition between the two components, making it appear as if the original component is morphing into the new one.

The `layout` prop, on the other hand, is for automatic animations of a component's position and size when those change due to reasons other than **shared** 'grouped' layout animations. See the Layout animation section, as explained if a component's parent resizes or if that component moves due to a sibling being added or removed, the `layout` prop will make sure this change is animated.

So, if a component has a `layoutId` but not a `layout` prop:

- It will animate when there's a matching `layoutId` elsewhere that it needs to animate to/from.
- It won't animate its own individual position or size changes due to `layout` shifts in its local space.

This distinction allows for fine-grained control over animations. You can choose to have a component participate in shared layout animations, local layout animations, both, or neither.

Your content is well-structured and provides a comprehensive overview of common issues and solutions when working with Framer Motion's layout animations. I've made some formatting adjustments for clarity and added a bit more context where needed:

Troubleshooting

When working with layout animations in Framer Motion, you might encounter unexpected behaviors. This guide provides solutions to common issues you might face.

Preventing text stretching or warping

To prevent text from stretching or warping during animations, wrap nested elements in their own `motion` component and set the `layout` prop to "`position`".

```
<motion.div layout={true}>
  <motion.p layout="position">
    Hello world!
  </motion.p>
</motion.div>
```

Preventing text "snapping"

If the text appears to "**snap**" to a new `position` at the start of the `transition`, it might be due to the characters shifting within their DOM rectangle.

Use CSS to ensure the DOM node tightly wraps around the characters:

```
<div style={{ display: 'flex', justifyContent: 'center' }}>
  <motion.p>
    Centered text
  </motion.p>
</div>
```

> **Note:** The above example uses inline styles for brevity. In practice, consider using CSS Modules or your preferred styling method.

Preventing element "jiggling"

If elements seem to "**jiggle**" during animations, ensure that nested `motion` components have consistent `transition` settings with the parent animating parent.

```
const CUSTOM_SPRING = {
  type: 'spring',
  stiffness: 300,
  damping: 45,
};

<motion.div transition={CUSTOM_SPRING}>
  <motion.p transition={CUSTOM_SPRING}>
    Centered text
  </motion.p>
</motion.div>
```

Addressing corner twitching

If the corners of an animated element appear to **twitch**, explicitly `borderRadius` in the `initial` prop:

```
<motion.div initial={{ borderRadius: 32 }} />
```

Similarly, for `boxShadow`, ensure you specify it explicitly, but note that it only works with a single shadow.

Handling shared layout animations

In shared layout animations where elements **teleport**, try wrapping all related elements in `LayoutGroup`:

```
<LayoutGroup>
  <div>
    {someCondition && <motion.div layoutId="some-val" />}
  </div>
  {!someCondition && <motion.div layoutId="some-val" />}
</LayoutGroup>
```

Preventing element "blinking"

If elements disappear during a shared layout animation, ensure `layoutId` & `key` props have same value:

```
<motion.div layoutId={layoutId} key={layoutId} />
```

Ensuring truthy layout identifier values

When setting the `layoutId` prop, ensure it's a **truthy** value, especially when combining strings:

```
<motion.div layoutId={`${uniqueId}-${index}`} />
```

Additional Tips:

- If an element doesn't animate, ensure it's not set to `display: inline`.
- For inline elements like `<span>` or `<a>`, set `display: block` to enable transformations.
- If unwanted layout animations occur, use `layoutRoot` to ignore specific changes.
- For a comprehensive list of issues and solutions, refer to the Framer Motion Troubleshooting Guide.