# React Built in Components:

* [<Fragment>](https://beta.reactjs.org/reference/react/Fragment), alternatively written as <>...</>, lets you group multiple JSX nodes together.
* [<Profiler>](https://beta.reactjs.org/reference/react/Profiler) lets you measure rendering performance of a React tree programmatically.
* [<Suspense>](https://beta.reactjs.org/reference/react/Suspense) lets you display a fallback while the child components are loading.
* [<StrictMode>](https://beta.reactjs.org/reference/react/StrictMode) enables extra development-only checks that help you find bugs early.

## Fragment:

* Wrap elements together as single entity
* No effects in DOM
* The empty JSX tag <></> is shorthand for <Fragment></Fragment> (ability pass key(unique key of DOM element) as prop ) in most cases.

Usage:

* [Returning multiple elements](https://beta.reactjs.org/reference/react/Fragment#returning-multiple-elements)
* [Assigning multiple elements to a variable](https://beta.reactjs.org/reference/react/Fragment#assigning-multiple-elements-to-a-variable)
* [Grouping elements with text](https://beta.reactjs.org/reference/react/Fragment#grouping-elements-with-text)
* [Rendering a list of Fragments](https://beta.reactjs.org/reference/react/Fragment#rendering-a-list-of-fragments)

// assignig  multiple elements

const allComp = (

  <>

    <Comp1 />

    <Comp2 />

  </>

);

// returning multiple elements

return (

  <>

    <Comp1 />

    <Comp2 />

  </>

);

// grouping element with text

<>

  Icon:

  <IconRender />

</>;

// rendering list of fragments

{

  shopItems.map((item) => (

    <Fragment key={item.id}>

      <TitleRenderer title={item.title} />

      <PictureRenderer img={item.img} />

    </Fragment>

  ));

}

**Props**

* **optional** key: Fragments declared with the explicit <Fragment> syntax may have [keys.](https://beta.reactjs.org/learn/rendering-lists#keeping-list-items-in-order-with-key)

**Caveats**

* If you want to pass key to a Fragment, you can’t use the <>...</> syntax. You have to explicitly import Fragment from 'react' and render <Fragment key={yourKey}>...</Fragment>.
* React does not [reset state](https://beta.reactjs.org/learn/preserving-and-resetting-state) when you go from rendering <><Child /></> to [<Child />] or back, or when you go from rendering <><Child /></> to <Child /> and back. This only works a single level deep: for example, going from <><><Child /></></> to <Child /> resets the state. See the precise semantics [here.](https://gist.github.com/clemmy/b3ef00f9507909429d8aa0d3ee4f986b)

For more {

<https://beta.reactjs.org/learn/preserving-and-resetting-state>

<https://gist.github.com/clemmy/b3ef00f9507909429d8aa0d3ee4f986b>

}

### Unique Keys:

Returning multiple elements in loop need unique keys

* JSX elements directly inside a map() call always need keys!

More about unique keys of react node:

* Each child of array needs unique key
* Uniqueness among its sibilings are fine
* Key type – number or string
* Must not change for each rendering
* Why ? in dynamic list, item can be added, removed or re-ordered, react needs to determine exact components to animate, keys become stable identity.
* Index as key – anit pattern in react, i.e if 2nd & 3rd item are swapped, react assumes like 2nd is removed & 3rd is added and will animate both items. (but logically since its just re-ordering, it don’t need to recreate / reanimate items) – react will use index as key if key not specified.
* must not be created while rendering – defeat its purpose itself since match up keys on renders never going to be equal leads to recreation everytime, affects performance too - do not generate keys on the fly, e.g. with key={Math.random()} -
* most recommended to maintain each for items on own in your constants or map from any unique db keys.
* **Locally generated data:** If your data is generated and persisted locally (e.g. notes in a note-taking app), use an incrementing counter, [crypto.randomUUID()](https://developer.mozilla.org/en-US/docs/Web/API/Crypto/randomUUID) or a package like [uuid](https://www.npmjs.com/package/uuid) when creating items.
* If child component except a prop named key, given key is taken as custom prop or else it will be react key prop itself.

When key used as index – wrong behavior - <https://robinpokorny.com/blog/index-as-a-key-is-an-anti-pattern/>

Algorithm behind it - <https://reactjs.org/docs/reconciliation.html#recursing-on-children>

Referred {

<https://beta.reactjs.org/learn/rendering-lists#keeping-list-items-in-order-with-key>

}

**Profiler**

* measure rendering performance of a React tree programmatically
* although lightweight component , it has to be disabled in production build as it overhead cpu and memory for profiling.  
  still need profiler included prod build check this - <https://gist.github.com/bvaughn/25e6233aeb1b4f0cdb8d8366e54a3977>

import "./styles.css";

import React, { Profiler } from "react";

export default function App() {

  const persons = [

    {

      id: 0, // used as item unqiue key

      name: "Arun",

    },

    {

      id: 1,

      name: "Ramu",

    },

  ];

  const profiling = (

    id,

    phase,

    actualDuration,

    baseDuration,

    startTime,

    commitTime

  ) => {

    const profilingObj = {

      id: id,

      phase,

      actualDuration,

      baseDuration,

      startTime,

      commitTime,

      difference: commitTime - startTime,

    };

    console.table(profilingObj);

  };

  const propName = <span>Name : </span>;

  const listItems = persons.map((person) => (

    <Profiler

      key={`listitem${person.id}`}

      id={`listitem${person.id}`}

      onRender={profiling}

    >

      <div key={person.id}>

        {propName}

        {person.name}

      </div>

    </Profiler>

  ));

  return (

    <div className="App">

      <Profiler id="unorderList" onRender={profiling}>

        <ul>{listItems}</ul>

      </Profiler>

    </div>

  );

}

// output

(index)

Value

id  'listitem0'

phase 'mount'

actualDuration  11.900000002235174

baseDuration  1.6000000014901161

startTime 3373

commitTime  3393.699999999255

difference  20.699999999254942

Object

index.js:27

(index)

Value

id  'listitem1'

phase 'mount'

actualDuration  0.7000000029802322

baseDuration  0.20000000298023224

startTime 3385.199999999255

commitTime  3393.699999999255

difference  8.5

Object

index.js:27

(index)

Value

id  'unorderList'

phase 'mount'

actualDuration  14.900000005960464

baseDuration  3.8000000044703484

startTime 3371

commitTime  3393.699999999255

difference  22.699999999254942

Object

**Props**

* id: A string identifying the part of the UI you are measuring.
* onRender: An [onRender callback](https://beta.reactjs.org/reference/react/Profiler#onrender-callback) that React calls it every time components within the profiled tree update. It receives information about what was rendered and how much time it took.

**Caveats**

* Profiling adds some additional overhead, so **it is disabled in the production build by default.** To opt into production profiling, you need to enable a [special production build with profiling enabled.](https://fb.me/react-profiling)

**Parameters**

* id: The string id prop of the <Profiler> tree that has just committed. This lets you identify which part of the tree was committed if you are using multiple profilers.
* phase: Either "mount" or "update". This lets you know whether the tree has just been mounted for the first time or re-rendered due to a change in props, state, or hooks.
* actualDuration: The number of milliseconds spent rendering the <Profiler> and its descendants for the current update. This indicates how well the subtree makes use of memoization (e.g. [memo](https://beta.reactjs.org/reference/react/memo) and [useMemo](https://beta.reactjs.org/reference/react/useMemo)). Ideally this value should decrease significantly after the initial mount as many of the descendants will only need to re-render if their specific props change.
* baseDuration: The number of milliseconds estimating how much time it would take to re-render the entire <Profiler> subtree without any optimizations. It is calculated by summing up the most recent render durations of each component in the tree. This value estimates a worst-case cost of rendering (e.g. the initial mount or a tree with no memoization). Compare actualDuration against it to see if memoization is working.
* startTime: A numeric timestamp for when React began rendering the current update.
* endTime: A numeric timestamp for when React committed the current update. This value is shared between all profilers in a commit, enabling them to be grouped if desirable.

More profiling {

<https://www.pluralsight.com/guides/profiling-performance-with-react-developer-tools>

react dev extension - <https://blog.openreplay.com/the-definitive-guide-to-profiling-react-applications/>

<https://stackoverflow.com/questions/53342791/react-profiler-what-do-the-timings-mean>

}

Extra {

Log table

<https://www.youtube.com/watch?v=5hddaunJWww>

}

**StrictMode**

* find common bugs in your components early during development