Pro Teams Pricing Documentation

npm

Search packages Search

react-countdown TS

2.3.6 • Public • Published 12 days ago

Readme

Code Beta

1 Dependency

205 Dependents

12 Versions

React <Countdown /> npm v2.3.6 build passing coverage 100%

A customizable countdown component for React.

Getting Started

You can install the module via `npm` or `yarn`:

```
npm install react-countdown --save
```

```
yarn add react-countdown
```

Motivation

As part of a small web app at first, the idea was to separate the countdown component from the main package to combine general aspects of the development with React, testing with Jest and more things that relate to publishing a new Open Source project.

Examples

Here are some examples which you can try directly online. You can also clone this repo and explore some more examples in there by running `yarn start` within the `examples` folder.

Basic Usage

A very simple and minimal example of how to set up a countdown that counts down from 10 seconds.

```
import React from 'react';
import ReactDOM from 'react-dom';
import Countdown from 'react-countdown';

ReactDOM.render(
  <Countdown date={Date.now() + 10000} />,
  document.getElementById('root')
);
```

Live Demo

Custom & Conditional Rendering

In case you want to change the output of the component or want to signal that the countdown's work is done, you can do this by either using the `onComplete` callback, a custom `renderer`, or by specifying a React child within `<Countdown></Countdown>`, which will only be shown once the countdown is complete.

Using a React Child for the Completed State

```
import React from 'react';
import ReactDOM from 'react-dom';
import Countdown from 'react-countdown';

// Random component
const Completionist = () => <span>You are good to go!</span>;

ReactDOM.render(
  (
    <Countdown date={Date.now() + 5000}>
      <Completionist />
    </Countdown>
  ),
  document.getElementById('root')
);
```

Live Demo

Custom Renderer with Completed Condition

```
import React from 'react';
import ReactDOM from 'react-dom';
import Countdown from 'react-countdown';

// Random component
```

```
const Completionist = () => <span>You are good to go!</span>;

// Renderer callback with condition
const renderer = ({ hours, minutes, seconds, completed }) => {
  if (completed) {
    // Render a completed state
    return <Completionist />;
  } else {
    // Render a countdown
    return <span>{hours}:{minutes}:{seconds}</span>;
  }
};

ReactDOM.render(
  <Countdown
    date={Date.now() + 5000}
    renderer={renderer}
  />,
  document.getElementById('root')
);
```

Live Demo

Countdown in Milliseconds

Here is an example with a countdown of 10 seconds that displays the total time difference in milliseconds. In order to display the milliseconds appropriately, the `intervalDelay` value needs to be lower than `1000` ms and a `precision` of `1` to `3` should be used. Last but not least, a simple `renderer` callback needs to be set up.

```
import React from 'react';
import ReactDOM from 'react-dom';
import Countdown from 'react-countdown';

ReactDOM.render(
  <Countdown
    date={Date.now() + 10000}
    intervalDelay={0}
    precision={3}
    renderer={props => <div>{props.total}</div>}
  />,
  document.getElementById('root')
);
```

Live Demo

Props

Name	Type	Default	Description
date	`Date|string|number`	required	`Date` or timestamp in the future
key	`string|number`	`undefined`	React `key`; car be used to restart the countdown
daysInHours	`boolean`	`false`	Days are calculated as hours
zeroPadTime	`number`	`2`	Length of zero-padded output, e.g.: `00:01:02`
zeroPadDays	`number`	`zeroPadTime`	Length of zero-padded days

Name	Type	Default	Description
			output, e.g.: `01`
controlled	`boolean`	`false`	Hands over the control to its parent(s)
intervalDelay	`number`	`1000`	Interval delay ir milliseconds
precision	`number`	`0`	The precision on a millisecond basis
autoStart	`boolean`	`true`	Countdown auto-start option
overtime	`boolean`	`false`	Counts down to infinity
children	`any`	`null`	A React child fo the countdown's completed state
renderer	`function`	`undefined`	Custom renderer callback
now	`function`	`Date.now`	Alternative handler for the current date

Name	Type	Default	Description
onMount	`function`	`undefined`	Callback when component mounts
onStart	`function`	`undefined`	Callback when countdown starts
onPause	`function`	`undefined`	Callback when countdown pauses
onStop	`function`	`undefined`	Callback when countdown stops
onTick	`function`	`undefined`	Callback on every interval tick (`controlled = false`)
onComplete	`function`	`undefined`	Callback when countdown ends

`date`

The `date` prop is the only required one and can be a `Date` object, `string`, or timestamp in the future. By default, this date is compared with the current date, or a custom handler defined via `now`.

Valid values can be *(and more)*:

- `'2020-02-01T01:02:03'` // `Date` time string format

- `1580518923000` // Timestamp in milliseconds
- `new Date(1580518923000)` // `Date` object

`key`

This is one of React's internal component props to help identify elements throughout the reconciliation process. It can be used to restart the countdown by passing in a new `string` or `number` value.

Please see **official React docs** for more information about keys.

`daysInHours`

Defines whether the time of day should be calculated as hours rather than separated days.

`controlled`

Can be useful if the countdown's interval and/or date control should be handed over to the parent. In case `controlled` is `true` , the provided **`date`** will be treated as the countdown's actual time difference and not be compared to **`now`** anymore.

`zeroPadTime`

This option defaults to `2` in order to display the common format `00:00:00` instead of `0:0:0` . If the value is higher than `2` , only the hours part *(see **`zeroPadDays`** for days)* will be zero-padded while it stays at `2` for minutes as well as seconds. If the value is lower, the output won't be zero-padded like the example before is showing.

`zeroPadDays`

Defaults to `zeroPadTime` . It works the same way as **`zeroPadTime`** does, just for days.

`intervalDelay`

Since this countdown is based on date comparisons, the default value of `1000` milliseconds is probably enough for most scenarios and doesn't need to be changed.

However, if it needs to be more precise, the `intervalDelay` can be set to something lower - down to `0` , which would, for example, allow showing the milliseconds in a more fancy way (*currently* only possible through a custom **`renderer`**).

`precision`

In certain cases, you might want to base off the calculations on a millisecond basis. The `precision` prop, which defaults to `0`, can be used to refine this calculation. While the default value simply strips the milliseconds part (e.g., `10123` ms => `10000` ms), a precision of `3` leads to `10123` ms.

`autoStart`

Defines whether the countdown should start automatically or not. Defaults to `true`.

`overtime`

Defines whether the countdown can go into overtime by extending its lifetime past the targeted endpoint. Defaults to `false`.

When set to `true`, the countdown timer won't stop when hitting 0, but instead becomes negative and continues to run unless paused/stopped. The `onComplete` callback would still get triggered when the initial countdown phase completes.

> Please note that the `children` prop will be ignored if `overtime` is `true`. Also, when using a custom `renderer`, you'll have to check one of the **render props**, e.g., `total`, or `completed`, to render the overtime output.

`children`

This component also considers the child that may live within the `<Countdown></Countdown>` element, which, in case it's available, replaces the countdown's component state once it's complete. Moreover, an additional prop called `countdown` is set and contains data similar to what the `renderer` callback would receive. Here's an example that showcases its usage.

> Please note that the `children` prop will be ignored if a custom `renderer` is defined.

`renderer`

The component's raw render output is kept very simple.

For more advanced countdown displays, a custom `renderer` callback can be defined to return a new React element. It receives the following render props as the first argument.

Render Props

The render props object consists of the current time delta object, the countdown's `api` , the component `props` , and last but not least, a `formatted` object.

```
{
  total: 0,
  days: 0,
  hours: 0,
  minutes: 0,
  seconds: 0,
  milliseconds: 0,
  completed: true,
  api: { ... },
  props: { ... },
  formatted: { ... }
}
```

> Please note that a defined custom `renderer` will ignore the `children` prop.

`now`

If the current date and time (determined via a reference to `Date.now`) is not the right thing to compare with for you, a reference to a custom function that returns a similar dynamic value could be provided as an alternative.

`onMount`

`onMount` is a callback and triggered when the countdown mounts. It receives a **time delta object** as the first argument.

`onStart`

`onStart` is a callback and triggered whenever the countdown is started (including first-run). It receives a **time delta object** as the first argument.

`onPause`

`onPause` is a callback and triggered every time the countdown is paused. It receives a **time delta object** as the first argument.

`onStop`

`onStop` is a callback and triggered every time the countdown is stopped. It receives a **time delta object** as the first argument.

`onTick`

`onTick` is a callback and triggered every time a new period is started, based on what the `intervalDelay`'s value is. It only gets triggered when the countdown's `controlled` prop is set to `false`, meaning that the countdown has full control over its interval. It receives a **time delta object** as the first argument.

`onComplete`

`onComplete` is a callback and triggered whenever the countdown ends. In contrast to `onTick`, the `onComplete` callback also gets triggered in case `controlled` is set to `true`. It receives a **time delta object** as the first argument and a `boolean` as a second argument, indicating whether the countdown transitioned into the completed state (`false`) or completed on start (`true`).

API Reference

The countdown component exposes a simple API through the `getApi()` function that can be accessed via component `ref`. It is also part (`api`) of the **render props** passed into `renderer` if needed. Here's an **example** of how to use it.

`start()`

Starts the countdown in case it is paused/stopped or needed when `autoStart` is set to `false`.

`pause()`

Pauses the running countdown. This only works as expected if the `controlled` prop is set to `false` because `calcTimeDelta` calculates an offset time internally.

`stop()`

Stops the countdown. This only works as expected if the `controlled` prop is set to `false` because `calcTimeDelta` calculates an offset time internally.

`isPaused()`

Returns a `boolean` for whether the countdown has been paused or not.

`isStopped()`

Returns a `boolean` for whether the countdown has been stopped or not.

`isCompleted()`

Returns a `boolean` for whether the countdown has been completed or not.

> Please note that this will always return `false` if `overtime` is `true`. Nevertheless, an into overtime running countdown's completed state can still be looking at the time delta object's `completed` value.

Helpers

This module also exports three simple helper functions, which can be utilized to build your own countdown custom `renderer`.

```
import Countdown, { zeroPad, calcTimeDelta, formatTimeDelta } from 're
```

`zeroPad(value, [length = 2])`

The `zeroPad` function transforms and returns a given `value` with padded zeros depending on the `length`. The `value` can be a `string` or `number`, while the `length` parameter can be a `number`, defaulting to `2`. Returns the zero-padded `string`, e.g., `zeroPad(5) => 05`.

```
const renderer = ({ hours, minutes, seconds }) => (
  <span>
    {zeroPad(hours)}:{zeroPad(minutes)}:{zeroPad(seconds)}
  </span>
);
```

`calcTimeDelta(date, [options])`

`calcTimeDelta` calculates the time difference between a given end `date` and the current date (`now`). It returns, similar to the `renderer` callback, a custom object (also referred to as **countdown time delta object**) with the following time-related data:

```
{ total, days, hours, minutes, seconds, milliseconds, completed }
```

The `total` value is the absolute time difference in milliseconds, whereas the other time-related values contain their relative portion of the current time difference. The `completed` value signalizes whether the countdown reached its initial end or not.

The `calcTimeDelta` function accepts two arguments in total; only the first one is required.

`date` Date or timestamp representation of the end date. See `date` prop for more details.

`options` The second argument consists of the following optional keys.

- **`now = Date.now`** Alternative function for returning the current date, also see `now` .
- **`precision = 0`** The `precision` on a millisecond basis.
- **`controlled = false`** Defines whether the calculated value is provided in a `controlled` environment as the time difference or not.
- **`offsetTime = 0`** Defines the offset time that gets added to the start time; only considered if controlled is false.
- **`overtime = false`** Defines whether the time delta can go into `overtime` and become negative or not. When set to `true` , the `total` could become negative, at which point `completed` will still be set to `true` .

`formatTimeDelta(timeDelta, [options])`

`formatTimeDelta` formats a given countdown time delta object. It returns the formatted portion of it, equivalent to:

```
{
  days: '00',
  hours: '00',
  minutes: '00',
  seconds: '00',
}
```

This function accepts two arguments in total; only the first one is required.

`timeDelta` Time delta object, e.g., returned by `calcTimeDelta` .

`options` The `options` object consists of the following three component props and is used to customize the time delta object's formatting:

- `daysInHours`
- `zeroPadTime`
- `zeroPadDays`

FAQ

Why does my countdown reset on every re-render?

A common reason for this is that the `date` prop gets passed directly into the component without persisting it in any way.

In order to avoid this from happening, it should be stored in a place that persists throughout lifecycle changes, for example, in the component's local `state` .

Why aren't my values formatted when using the custom `renderer` ?

The `renderer` callback gets called with a **time delta object** that also consists of a `formatted` object which holds these formatted values.

Why do I get this error `"Warning: Text content did not match..."` ?

This could have something to do with server-side rendering and that the countdown already runs on the server-side, resulting in a timestamp discrepancy between the client and the server. In this case, it might be worth checking https://reactjs.org/docs/dom-elements.html#suppresshydrationwarning.

Alternatively, you could try to set `autoStart` to `false` and start the countdown through the API once it's available on the client. Here are some related issues that might help in fixing this problem.

Contributing

Contributions of any kind are very welcome. Read more in our contributing guide about how to report bugs, create pull requests, and other development-related topics.

License

MIT

Keywords

react typescript countdown component

Install

```
npm i react-countdown
```

Repository

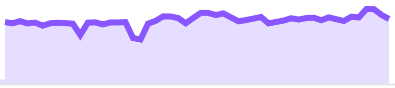
github.com/ndresx/react-countdown

Homepage

github.com/ndresx/react-countdown

Weekly Downloads

172,116



Version

2.3.6

License

MIT

Unpacked Size

65.2 kB

Total Files

9

Issues

3

Pull Requests

1

Last publish

12 days ago

Collaborators

>_Try on RunKit

Report malware

Support

Help

Advisories

Status

Contact npm

Company

About

Blog

Press

Terms & Policies

Policies

Terms of Use

Code of Conduct

Privacy