Search... Ctrl+K

Autocomplete

The autocomplete is a normal text input enhanced by a panel of suggested options.

The widget is useful for setting the value of a single-line textbox in one of two types of scenarios:

1. The value for the textbox must be chosen from a predefined set of allowed values, for example a location field must contain a valid location name: combo box.
2. The textbox may contain any arbitrary value, but it is advantageous to suggest possible values to the user, for example a search field may suggest similar or previous searches to save the user time: free solo.

It's meant to be an improved version of the "react-select" and "downshift" packages.

View as Markdown Feedback Bundle size Source WAI-ARIA Figma Sketch

Combo box

The value must be chosen from a predefined set of allowed values.

Edit in Chat JS TS Collapse code

./ComboBox.tsx ./top100Films.ts

```
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';
import top100Films from './top100Films';

export default function ComboBox() {
  return (
    <Autocomplete
      disablePortal
      options={top100Films}
      sx={{ width: 300 }}
      renderInput={(params) => <TextField {...params} label="Movie" />}
    />
  );
}
```

Options structure

By default, the component accepts the following options structures:

```
interface AutocompleteOption {
  label: string;
}
// or
type AutocompleteOption = string;
```

for instance:

```
const options = [
  { label: 'The Godfather', id: 1 },
  { label: 'Pulp Fiction', id: 2 },
];
// or
const options = ['The Godfather', 'Pulp Fiction'];
```

However, you can use different structures by providing a `getOptionLabel` prop.

If your options are objects, you must provide the `isOptionEqualToValue` prop to ensure correct selection and highlighting. By default, it uses strict equality to compare options with the current value.

⚠ If your options have duplicate labels, you must extract a unique key with the `getOptionKey` prop.

```
const options = [
  { label: 'The Godfather', id: 1 },
  { label: 'The Godfather', id: 2 },
];

return <Autocomplete options={options} getOptionKey={(option) => option.id} />;
```

Playground

Each of the following examples demonstrates one feature of the Autocomplete component.

disableCloseOnSelect

clearOnEscape

disableClearable

includeInputInList

flat

controlled

autoComplete

disableListWrap

openOnFocus

autoHighlight

autoSelect

disabled

disablePortal

blurOnSelect

clearOnBlur

selectOnFocus

readOnly

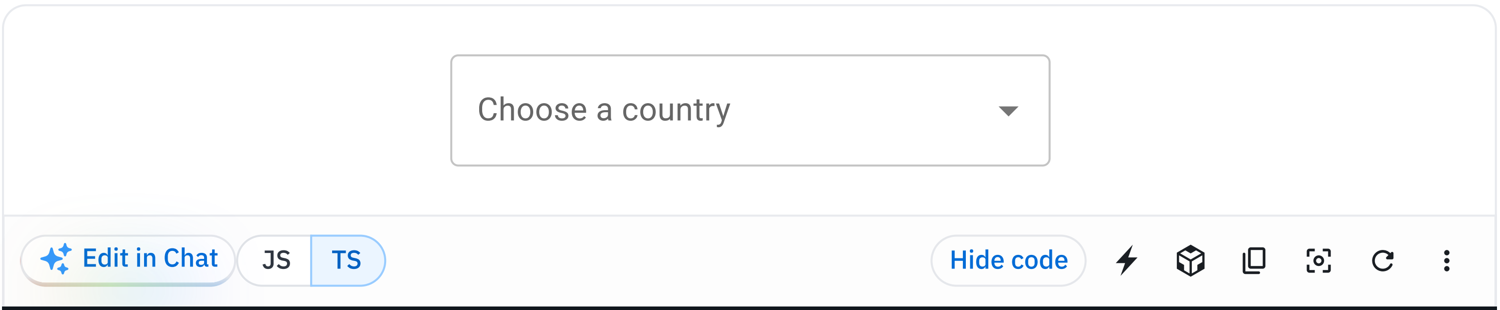
Inception

Edit in Chat JS TS Hide code

```
import * as React from 'react';
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';
import Stack from '@mui/material/Stack';

export default function Playground() {
  const defaultProps = {
    options: top100Films,
    getOptionLabel: (option: FilmOptionType) => option.title,
  };
  const flatProps = {
    options: top100Films.map((option) => option.title),
  };
  const [value, setValue] = React.useState<FilmOptionType | null>(null);

  return (
    <Stack spacing={1} sx={{ width: 300 }}>
      <Autocomplete
        {...defaultProps}
        id="disable-close-on-select"
        disableCloseOnSelect
        renderInput={(params) => (
          <TextField {...params} label="disableCloseOnSelect" variant="standard" />
        )}
      />
      <Autocomplete
        {...defaultProps}
        id="clear-on-escape"
        clearOnEscape
        renderInput={(params) => (
          <TextField {...params} label="clearOnEscape" variant="standard" />
        )}
      />
      <Autocomplete
        {...defaultProps}
        id="disable-clearable"
        disableClearable
        renderInput={(params) => (
```

Country select

Choose one of the 248 countries.



```
import Box from '@mui/material/Box';
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';

export default function CountrySelect() {
  return (
    <Autocomplete
      id="country-select-demo"
      sx={{ width: 300 }}
      options={countries}
      autoHighlight
      getOptionLabel={(option) => option.label}
      renderOption={(props, option) => {
        const { key, ...optionProps } = props;
        return (
          <Box
            key={key}
            component="li"
            sx={{ '& > img': { mr: 2, flexShrink: 0 } }}
            {...optionProps}
          >
            <img
              loading="lazy"
              width="20"
              srcSet={`https://flagcdn.com/w40/${option.code.toLowerCase()}.png 2x`}
              src={`https://flagcdn.com/w20/${option.code.toLowerCase()}.png`}
              alt=""
            />
            {option.label} ({option.code}) +{option.phone}
          </Box>
        );
      }}
      renderInput={(params) => (
        <TextField
          {...params}
          label="Choose a country"
          slotProps={{
            htmlInput: {
```

Controlled states

The component has two states that can be controlled:

1. the "value" state with the `value`/`onChange` props combination. This state represents the value selected by the user, for instance when pressing `Enter`.
2. the "input value" state with the `inputValue`/`onInputChange` props combination. This state represents the value displayed in the textbox.

These two states are isolated, and should be controlled independently.

- A component is **controlled** when it's managed by its parent using props.
- A component is **uncontrolled** when it's managed by its own local state.

Learn more about controlled and uncontrolled components in the React documentation.

value: 'Option 1'
inputValue: 'Option 1'

Controllable
Option 1

Edit in Chat JS TS Hide code

```tsx
import * as React from 'react';
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';

const options = ['Option 1', 'Option 2'];

export default function ControllableStates() {
  const [value, setValue] = React.useState<string | null>(options[0]);
  const [inputValue, setInputValue] = React.useState('');

  return (
    <div>
      <div>{`value: ${value !== null ? `'${value}'` : 'null'}`}</div>
      <div>{`inputValue: '${inputValue}'`}</div>
      <br />
      <Autocomplete
        value={value}
        onChange={(event: any, newValue: string | null) => {
          setValue(newValue);
        }}
        inputValue={inputValue}
        onInputChange={(event, newInputValue) => {
          setInputValue(newInputValue);
        }}
        id="controllable-states-demo"
        options={options}
        sx={{ width: 300 }}
        renderInput={(params) => <TextField {...params} label="Controllable" />}
      />
    </div>
  );
}
```

⚠ If you control the `value`, make sure it's referentially stable between renders. In other words, the reference to the value shouldn't change if the value itself doesn't change.

```jsx
// ⚠ BAD
return <Autocomplete multiple value={allValues.filter((v) => v.selected)} />;
```

```
// 👍 GOOD
const selectedValues = React.useMemo(
  () => allValues.filter((v) => v.selected),
  [allValues],
);
return <Autocomplete multiple value={selectedValues} />;
```

In the first example, `allValues.filter` is called and returns **a new array** every render. The fix includes memoizing the value, so it changes only when needed.

Free solo

Set `freeSolo` to true so the textbox can contain any arbitrary value.

Search input

The prop is designed to cover the primary use case of a **search input** with suggestions, for example Google search or react-autowhatever.

Edit in Chat JS TS Hide code

```
import TextField from '@mui/material/TextField';
import Stack from '@mui/material/Stack';
import Autocomplete from '@mui/material/Autocomplete';

export default function FreeSolo() {
  return (
    <Stack spacing={2} sx={{ width: 300 }}>
      <Autocomplete
        id="free-solo-demo"
        freeSolo
        options={top100Films.map((option) => option.title)}
        renderInput={(params) => <TextField {...params} label="freeSolo" />}
      />
      <Autocomplete
        freeSolo
        id="free-solo-2-demo"
        disableClearable
        options={top100Films.map((option) => option.title)}
        renderInput={(params) => (
          <TextField
            {...params}
            label="Search input"
            slotProps={{
              input: {
                ...params.InputProps,
                type: 'search',
              },
            }}
          />
        )}
      />
    </Stack>
  );
}

// Top 100 films as rated by IMDb users. http://www.imdb.com/chart/top
const top100Films = [
```

Be careful when using the free solo mode with non-string options, as it may cause type mismatch.

The value created by typing into the textbox is always a string, regardless of the type of the options.

Creatable

If you intend to use this mode for a combo box like experience (an enhanced version of a select element) we recommend setting:

- `selectOnFocus` to help the user clear the selected value.
- `clearOnBlur` to help the user enter a new value.
- `handleHomeEndKeys` to move focus inside the popup with the `Home` and `End` keys.
- A last option, for instance: `Add "YOUR SEARCH"`.

Free solo with text demo

Edit in Chat JS TS Hide code

```
import * as React from 'react';
import TextField from '@mui/material/TextField';
import Autocomplete, { createFilterOptions } from '@mui/material/Autocomplete';

const filter = createFilterOptions<FilmOptionType>();

export default function FreeSoloCreateOption() {
  const [value, setValue] = React.useState<FilmOptionType | null>(null);

  return (
    <Autocomplete
      value={value}
      onChange={(event, newValue) => {
        if (typeof newValue === 'string') {
          setValue({
            title: newValue,
          });
        } else if (newValue && newValue.inputValue) {
          // Create a new value from the user input
          setValue({
            title: newValue.inputValue,
          });
        } else {
          setValue(newValue);
        }
      }}
      filterOptions={(options, params) => {
        const filtered = filter(options, params);

        const { inputValue } = params;
        // Suggest the creation of a new value
        const isExisting = options.some((option) => inputValue === option.title);
        if (inputValue !== '' && !isExisting) {
          filtered.push({
            inputValue,
            title: `Add "${inputValue}"`,
          });
        }
```

You could also display a dialog when the user wants to add a new value.

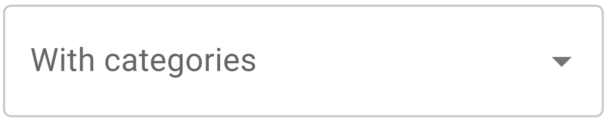
Free solo dialog

Edit in Chat JS TS Hide code

```
import * as React from 'react';
import TextField from '@mui/material/TextField';
import Dialog from '@mui/material/Dialog';
import DialogTitle from '@mui/material/DialogTitle';
import DialogContent from '@mui/material/DialogContent';
import DialogContentText from '@mui/material/DialogContentText';
import DialogActions from '@mui/material/DialogActions';
import Button from '@mui/material/Button';
import Autocomplete, { createFilterOptions } from '@mui/material/Autocomplete';

const filter = createFilterOptions<FilmOptionType>();

export default function FreeSoloCreateOptionDialog() {
  const [value, setValue] = React.useState<FilmOptionType | null>(null);
  const [open, toggleOpen] = React.useState(false);
  const handleClose = () => {
    setDialogValue({
      title: '',
      year: '',
    });
    toggleOpen(false);
  };

  const [dialogValue, setDialogValue] = React.useState({
    title: '',
    year: '',
  });

  const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();
    setValue({
      title: dialogValue.title,
      year: parseInt(dialogValue.year, 10),
    });
    handleClose();
  };
```

Grouped

You can group the options with the `groupBy` prop. If you do so, make sure that the options are also sorted with the same dimension that they are grouped by, otherwise, you will notice duplicate headers.



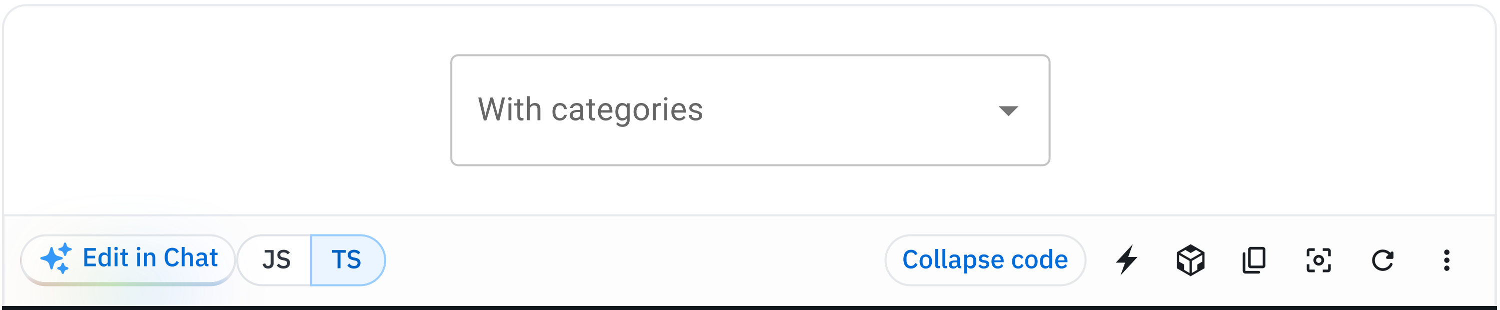
Edit in Chat JS TS Collapse code

```
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';

export default function Grouped() {
  const options = top100Films.map((option) => {
    const firstLetter = option.title[0].toUpperCase();
    return {
      firstLetter: /[0-9]/.test(firstLetter) ? '0-9' : firstLetter,
      ...option,
    };
  });

  return (
    <Autocomplete
      options={options.sort((a, b) => -b.firstLetter.localeCompare(a.firstLetter))}
      groupBy={(option) => option.firstLetter}
      getOptionLabel={(option) => option.title}
      sx={{ width: 300 }}
      renderInput={(params) => <TextField {...params} label="With categories" />}
    />
  );
}

// Top 100 films as rated by IMDb users. http://www.imdb.com/chart/top
const top100Films = [
  { title: 'The Shawshank Redemption', year: 1994 },
  { title: 'The Godfather', year: 1972 },
  { title: 'The Godfather: Part II', year: 1974 },
  { title: 'The Dark Knight', year: 2008 },
  { title: '12 Angry Men', year: 1957 },
  { title: "Schindler's List", year: 1993 },
  { title: 'Pulp Fiction', year: 1994 },
  {
    title: 'The Lord of the Rings: The Return of the King',
    year: 2003,
  },
  { title: 'The Good, the Bad and the Ugly', year: 1966 },
  { title: 'Fight Club', year: 1999 },
```

To control how the groups are rendered, provide a custom `renderGroup` prop. This is a function that accepts an object with two fields:

- `group`—a string representing a group name
- `children`—a collection of list items that belong to the group

The following demo shows how to use this prop to define custom markup and override the styles of the default groups:



```
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';
import { styled, lighten, darken } from '@mui/system';

const GroupHeader = styled('div')(({ theme }) => ({
  position: 'sticky',
  top: '-8px',
  padding: '4px 10px',
  color: theme.palette.primary.main,
  backgroundColor: lighten(theme.palette.primary.light, 0.85),
  ...theme.applyStyles('dark', {
    backgroundColor: darken(theme.palette.primary.main, 0.8),
  }),
}));

const GroupItems = styled('ul')({
  padding: 0,
});

export default function RenderGroup() {
  const options = top100Films.map((option) => {
    const firstLetter = option.title[0].toUpperCase();
    return {
      firstLetter: /[0-9]/.test(firstLetter) ? '0-9' : firstLetter,
      ...option,
    };
  });

  return (
    <Autocomplete
      options={options.sort((a, b) => -b.firstLetter.localeCompare(a.firstLetter))}
      groupBy={(option) => option.firstLetter}
      getOptionLabel={(option) => option.title}
      sx={{ width: 300 }}
      renderInput={(params) => <TextField {...params} label="With categories" />}
      renderGroup={(params) => (
        <li key={params.key}>
          <GroupHeader>{params.group}</GroupHeader>
```

Disabled options

Disabled options

Edit in Chat JS TS Collapse code

```
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';

export default function DisabledOptions() {
  return (
    <Autocomplete
      options={timeSlots}
      getOptionDisabled={(option) =>
        option === timeSlots[0] || option === timeSlots[2]
      }
      sx={{ width: 300 }}
      renderInput={(params) => <TextField {...params} label="Disabled options" />}
    />
  );
}
```

```
// One time slot every 30 minutes.
const timeSlots = Array.from(new Array(24 * 2)).map(
  (_, index) =>
    `${index < 20 ? '0' : ''}${Math.floor(index / 2)}:${
      index % 2 === 0 ? '00' : '30'
    }`,
);
```

`useAutocomplete`

For advanced customization use cases, a headless `useAutocomplete()` hook is exposed. It accepts almost the same options as the Autocomplete component minus all the props related to the rendering of JSX. The Autocomplete component is built on this hook.

```
import { useAutocomplete } from '@mui/base/useAutocomplete';
```

The `useAutocomplete` hook is also reexported from @mui/material for convenience and backward compatibility.

```
import useAutocomplete from '@mui/material/useAutocomplete';
```

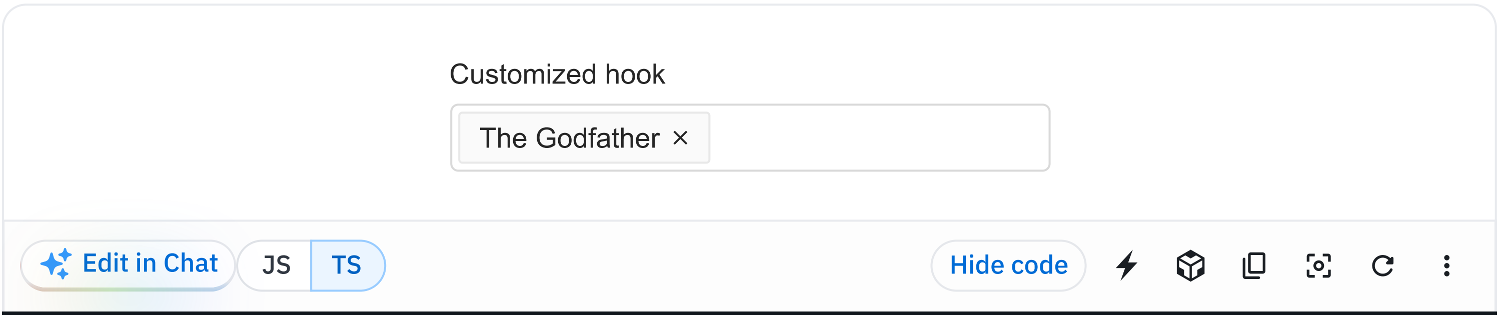
- 📦 4.6 kB gzipped.

useAutocomplete

```
import useAutocomplete from '@mui/material/useAutocomplete';
import { styled } from '@mui/system';

const Label = styled('label')({
  display: 'block',
});

const Input = styled('input')(({ theme }) => ({
  width: 200,
  backgroundColor: '#fff',
  color: '#000',
  ...theme.applyStyles('dark', {
    backgroundColor: '#000',
    color: '#fff',
  }),
}));
```

```
const Listbox = styled('ul')(({ theme }) => ({
  width: 200,
  margin: 0,
  padding: 0,
  zIndex: 1,
  position: 'absolute',
  listStyle: 'none',
  backgroundColor: '#fff',
  overflow: 'auto',
  maxHeight: 200,
  border: '1px solid rgba(0,0,0,.25)',
  '& li.Mui-focused': {
    backgroundColor: '#4a8df6',
    color: 'white',
    cursor: 'pointer',
  },
  '& li:active': {
    backgroundColor: '#2977f5',
    color: 'white',
  },
  ...theme.applyStyles('dark', {
```

Customized hook


Customized hook

The Godfather ×

```
import useAutocomplete, {
  AutocompleteGetItemProps,
  UseAutocompleteProps,
} from '@mui/material/useAutocomplete';
import CheckIcon from '@mui/icons-material/Check';
import CloseIcon from '@mui/icons-material/Close';
import { styled } from '@mui/material/styles';
import { autocompleteClasses } from '@mui/material/Autocomplete';

const Root = styled('div')(({ theme }) => ({
  color: 'rgba(0,0,0,0.85)',
  fontSize: '14px',
  ...theme.applyStyles('dark', {
    color: 'rgba(255,255,255,0.65)',
  }),
}));

const Label = styled('label')`
  padding: 0 0 4px;
  line-height: 1.5;
  display: block;
`;

const InputWrapper = styled('div')(({ theme }) => ({
  width: '300px',
  border: '1px solid #d9d9d9',
  backgroundColor: '#fff',
  borderRadius: '4px',
  padding: '1px',
  display: 'flex',
  flexWrap: 'wrap',
  ...theme.applyStyles('dark', {
    borderColor: '#434343',
    backgroundColor: '#141414',
  }),
  '&:hover': {
    borderColor: '#40a9ff',
```

Head to the customization section for an example with the `Autocomplete` component instead of the hook.

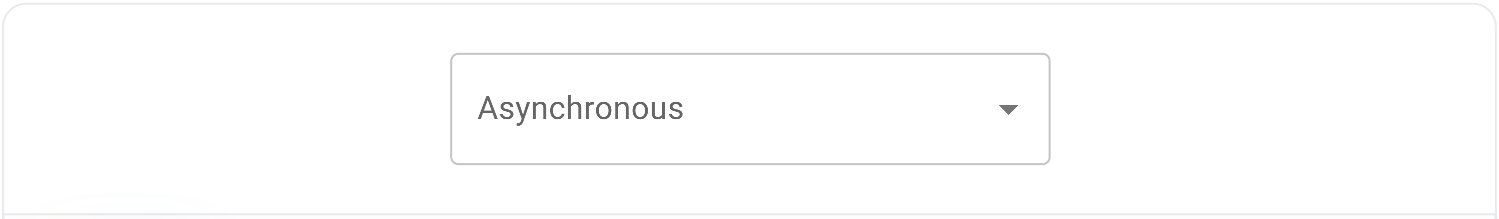
Asynchronous requests

The component supports two different asynchronous use-cases:

- Load on open: it waits for the component to be interacted with to load the options.
- Search as you type: a new request is made for each keystroke.

Load on open

It displays a progress state as long as the network request is pending.



```
import * as React from 'react';
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';
import CircularProgress from '@mui/material/CircularProgress';

interface Film {
  title: string;
  year: number;
}
```

```
function sleep(duration: number): Promise<void> {
  return new Promise<void>((resolve) => {
    setTimeout(() => {
      resolve();
    }, duration);
  });
}

export default function Asynchronous() {
  const [open, setOpen] = React.useState(false);
  const [options, setOptions] = React.useState<readonly Film[]>([]);
  const [loading, setLoading] = React.useState(false);

  const handleOpen = () => {
    setOpen(true);
    (async () => {
      setLoading(true);
      await sleep(1e3); // For demo purposes.
      setLoading(false);

      setOptions([...topFilms]);
    })();
  };

  const handleClose = () => {
    setOpen(false);
    setOptions([]);
  };
```

Search as you type

If your logic is fetching new options on each keystroke and using the current value of the textbox to filter on the server, you may want to consider throttling requests.

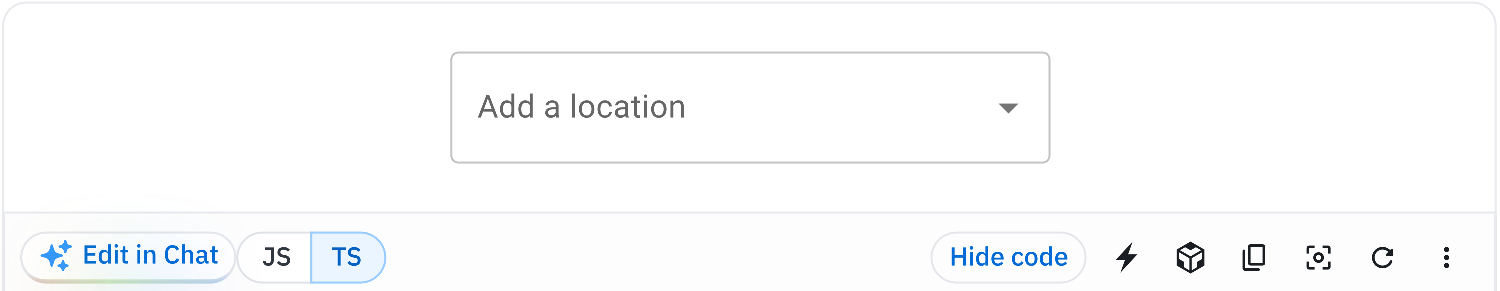
Additionally, you will need to disable the built-in filtering of the `Autocomplete` component by overriding the `filterOptions` prop:

```
<Autocomplete filterOptions={(x) => x} />
```

Copy

Google Maps place

A customized UI for Google Maps Places Autocomplete. For this demo, we need to load the [Google Maps JavaScript](#) and [Google Places](#) API.



```
import * as React from 'react';
import Box from '@mui/material/Box';
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';
import Paper, { PaperProps } from '@mui/material/Paper';
import LocationOnIcon from '@mui/icons-material/LocationOn';
import Grid from '@mui/material/Grid';
import Typography from '@mui/material/Typography';
import parse from 'autosuggest-highlight/parse';
// For the sake of this demo, we have to use debounce to reduce Google Maps Places API quote use
// But prefer to use throttle in practice
// import throttle from 'lodash/throttle';
import { debounce } from '@mui/material/utils';

// This key was created specifically for the demo in mui.com.
// You need to create a new one for your application.
const GOOGLE_MAPS_API_KEY = 'AIzaSyC3aviU6KHXAjoSnxcw6qbOhjnFctbxPkE';

const useEnhancedEffect =
  typeof window !== 'undefined' ? React.useLayoutEffect : React.useEffect;

function loadScript(src: string, position: HTMLElement) {
  const script = document.createElement('script');
  script.setAttribute('async', '');
  script.src = src;
  position.appendChild(script);
  return script;
}

interface MainTextMatchedSubstrings {
  offset: number;
  length: number;
}
interface StructuredFormatting {
  main_text: string;
  main_text_matched_substrings: readonly MainTextMatchedSubstrings[];
  secondary_text?: string;
}
```

The demo relies on autosuggest-highlight, a small (1 kB) utility for highlighting text in autosuggest and autocomplete components.

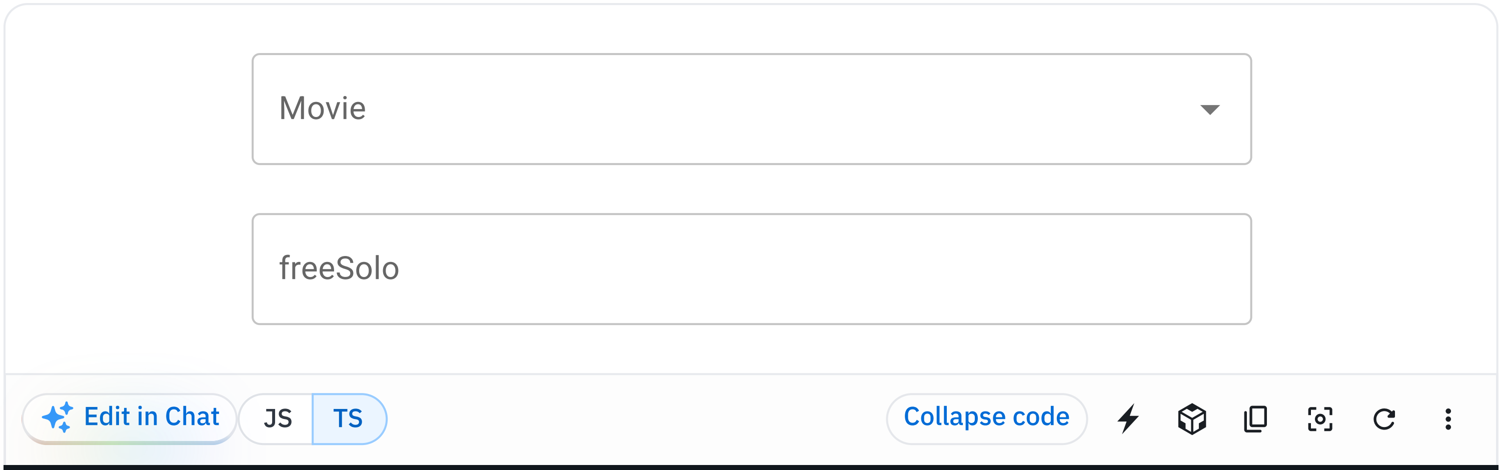
Before you can start using the Google Maps JavaScript API and Places API, you need to get your own API key.

This demo has limited quotas to make API requests. When your quota exceeds, you will see the response for "Paris".

Single value rendering

By default (when `multiple={false}`), the selected option is displayed as plain text inside the input. The `renderValue` prop allows you to customize how the selected value is rendered. This can be useful for adding custom styles, displaying additional information, or formatting the value differently.

- The `getItemProps` getter provides props like `data-item-index`, `disabled`, `tabIndex` and others. These props should be spread onto the rendered component for proper accessibility.
- If using a custom component other than a Material UI Chip, destructure the `onDelete` prop as it's specific to the Material UI Chip.

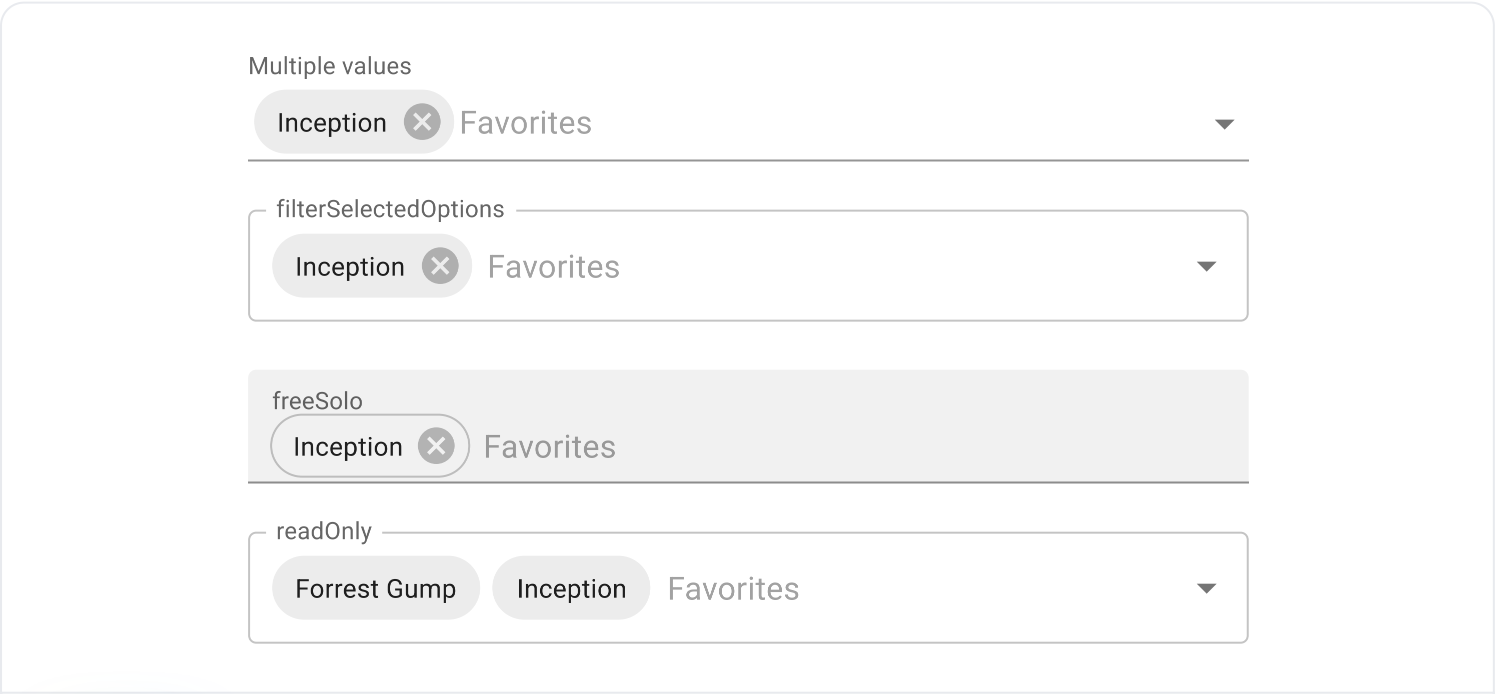


```
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';
import Chip from '@mui/material/Chip';
import Stack from '@mui/material/Stack';

export default function CustomSingleValueRendering() {
  return (
    <Stack spacing={3} sx={{ width: 500 }}>
      <Autocomplete
        options={top100Films}
        getOptionLabel={(option) => option.title}
        renderValue={(value, getItemProps) => (
          <Chip label={value.title} {...getItemProps()} />
        )}
        renderInput={(params) => <TextField {...params} label="Movie" />}
      />
      <Autocomplete
        options={top100Films.map((option) => option.title)}
        freeSolo
        renderValue={(value, getItemProps) => (
          <Chip label={value} {...getItemProps()} />
        )}
        renderInput={(params) => <TextField {...params} label="freeSolo" />}
      />
    </Stack>
  );
}

// Top 100 films as rated by IMDb users. http://www.imdb.com/chart/top
const top100Films = [
  { title: 'The Shawshank Redemption', year: 1994 },
  { title: 'The Godfather', year: 1972 },
  { title: 'The Godfather: Part II', year: 1974 },
  { title: 'The Dark Knight', year: 2008 },
  { title: '12 Angry Men', year: 1957 },
  { title: "Schindler's List", year: 1993 },
  { title: 'Pulp Fiction', year: 1994 },
  {
```

Multiple values

When `multiple={true}`, the user can select multiple values. These selected values, referred to as "items" can be customized using the `renderValue` prop.

- The `getItemProps` getter supplies essential props like `data-item-index`, `disabled`, `tabIndex` and others. Make sure to spread them on each rendered item.
- If using a custom component other than a Material UI Chip, destructure the `onDelete` prop as it's specific to the Material UI Chip.



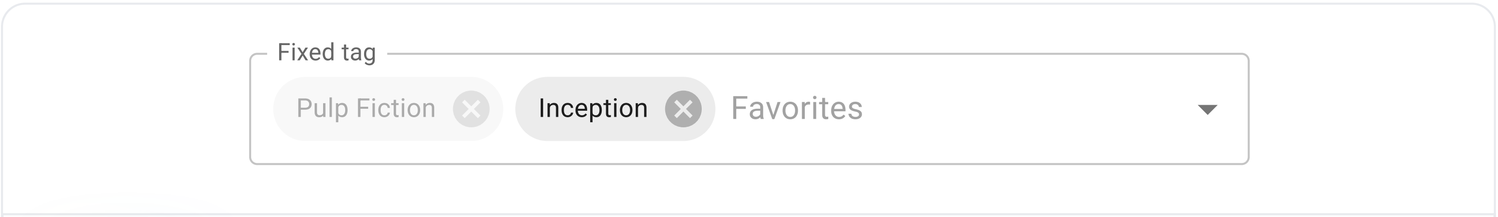
Edit in Chat JS TS Hide code

```
import Chip from '@mui/material/Chip';
import Autocomplete from '@mui/material/Autocomplete';
import TextField from '@mui/material/TextField';
import Stack from '@mui/material/Stack';

export default function Tags() {
  return (
    <Stack spacing={3} sx={{ width: 500 }}>
      <Autocomplete
        multiple
        id="tags-standard"
        options={top100Films}
        getOptionLabel={(option) => option.title}
        defaultValue={[top100Films[13]]}
        renderInput={(params) => (
          <TextField
```

```
            {...params}
            variant="standard"
            label="Multiple values"
            placeholder="Favorites"
          />
        )}
      />
      <Autocomplete
        multiple
        id="tags-outlined"
        options={top100Films}
        getOptionLabel={(option) => option.title}
        defaultValue={[top100Films[13]]}
        filterSelectedOptions
        renderInput={(params) => (
          <TextField
            {...params}
            label="filterSelectedOptions"
            placeholder="Favorites"
          />
        )}
      />
```

Fixed options

In the event that you need to lock certain tags so that they can't be removed, you can set the chips disabled.



Edit in Chat JS TS Hide code

```
import * as React from 'react';
import Chip from '@mui/material/Chip';
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';

export default function FixedTags() {
  const fixedOptions = [top100Films[6]];
  const [value, setValue] = React.useState([...fixedOptions, top100Films[13]]);

  return (
    <Autocomplete
      multiple
      id="fixed-tags-demo"
      value={value}
      onChange={(event, newValue) => {
        setValue([
          ...fixedOptions,
          ...newValue.filter((option) => !fixedOptions.includes(option)),
        ]);
      }}
      options={top100Films}
      getOptionLabel={(option) => option.title}
      renderValue={(values, getItemProps) =>
        values.map((option, index) => {
          const { key, ...itemProps } = getItemProps({ index });
          return (
            <Chip
              key={key}
              label={option.title}
              {...itemProps}
              disabled={fixedOptions.includes(option)}
            />
          );
        })
      }
      style={{ width: 500 }}
      renderInput={(params) => (
        <TextField {...params} label="Fixed tag" placeholder="Favorites" />
```

Checkboxes

Checkboxes

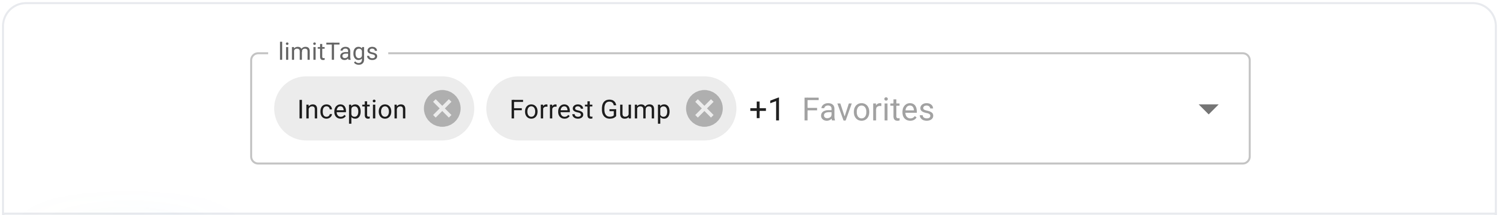
Edit in Chat JS TS Hide code

```
import Checkbox from '@mui/material/Checkbox';
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';
import CheckBoxOutlineBlankIcon from '@mui/icons-material/CheckBoxOutlineBlank';
import CheckBoxIcon from '@mui/icons-material/CheckBox';

const icon = <CheckBoxOutlineBlankIcon fontSize="small" />;
const checkedIcon = <CheckBoxIcon fontSize="small" />;

export default function CheckboxesTags() {
  return (
    <Autocomplete
      multiple
      id="checkboxes-tags-demo"
      options={top100Films}
      disableCloseOnSelect
      getOptionLabel={(option) => option.title}
      renderOption={(props, option, { selected }) => {
        const { key, ...optionProps } = props;
        return (
          <li key={key} {...optionProps}>
            <Checkbox
```

```
              icon={icon}
              checkedIcon={checkedIcon}
              style={{ marginRight: 8 }}
              checked={selected}
            />
            {option.title}
          </li>
        );
      }}
      style={{ width: 500 }}
      renderInput={(params) => (
        <TextField {...params} label="Checkboxes" placeholder="Favorites" />
      )}
    />
  );
```

Limit tags

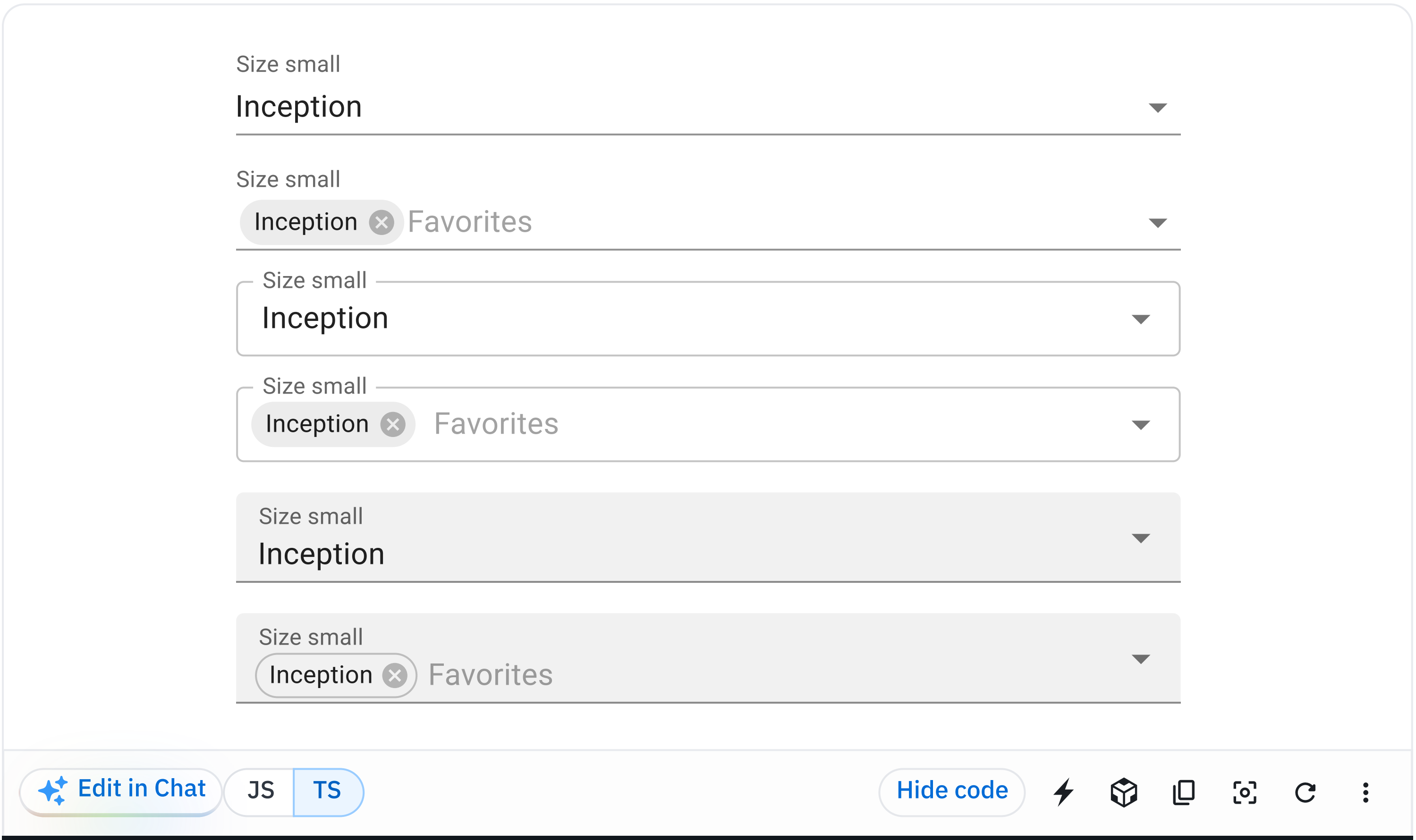
You can use the `limitTags` prop to limit the number of displayed options when not focused.



```
import Autocomplete from '@mui/material/Autocomplete';
import TextField from '@mui/material/TextField';

export default function LimitTags() {
  return (
    <Autocomplete
      multiple
      limitTags={2}
      id="multiple-limit-tags"
      options={top100Films}
      getOptionLabel={(option) => option.title}
      defaultValue={[top100Films[13], top100Films[12], top100Films[11]]}
      renderInput={(params) => (
        <TextField {...params} label="limitTags" placeholder="Favorites" />
      )}
      sx={{ width: '500px' }}
    />
  );
}

// Top 100 films as rated by IMDb users. http://www.imdb.com/chart/top
const top100Films = [
  { title: 'The Shawshank Redemption', year: 1994 },
  { title: 'The Godfather', year: 1972 },
  { title: 'The Godfather: Part II', year: 1974 },
  { title: 'The Dark Knight', year: 2008 },
  { title: '12 Angry Men', year: 1957 },
  { title: "Schindler's List", year: 1993 },
  { title: 'Pulp Fiction', year: 1994 },
  {
    title: 'The Lord of the Rings: The Return of the King',
    year: 2003,
  },
  { title: 'The Good, the Bad and the Ugly', year: 1966 },
  { title: 'Fight Club', year: 1999 },
  {
    title: 'The Lord of the Rings: The Fellowship of the Ring',
    year: 2001,
```

Sizes

Fancy smaller inputs? Use the `size` prop.



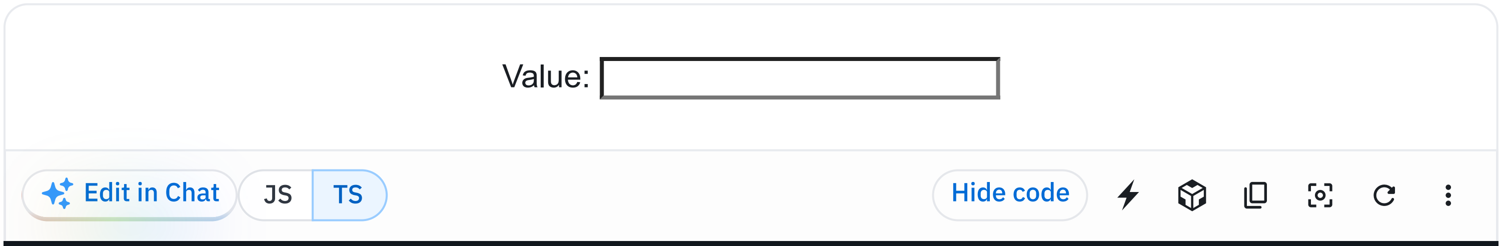
```
import Stack from '@mui/material/Stack';
import Chip from '@mui/material/Chip';
import Autocomplete from '@mui/material/Autocomplete';
import TextField from '@mui/material/TextField';

export default function Sizes() {
  return (
    <Stack spacing={2} sx={{ width: 500 }}>
      <Autocomplete
        id="size-small-standard"
        size="small"
        options={top100Films}
        getOptionLabel={(option) => option.title}
        defaultValue={top100Films[13]}
        renderInput={(params) => (
          <TextField
            {...params}
            variant="standard"
            label="Size small"
            placeholder="Favorites"
          />
        )}
      />
      <Autocomplete
        multiple
        id="size-small-standard-multi"
        size="small"
        options={top100Films}
        getOptionLabel={(option) => option.title}
        defaultValue={[top100Films[13]]}
        renderInput={(params) => (
          <TextField
            {...params}
            variant="standard"
            label="Size small"
            placeholder="Favorites"
          />
        )}
```

Customization

Custom input

The `renderInput` prop allows you to customize the rendered input. The first argument of this render prop contains props that you need to forward. Pay specific attention to the `ref` and `inputProps` keys.

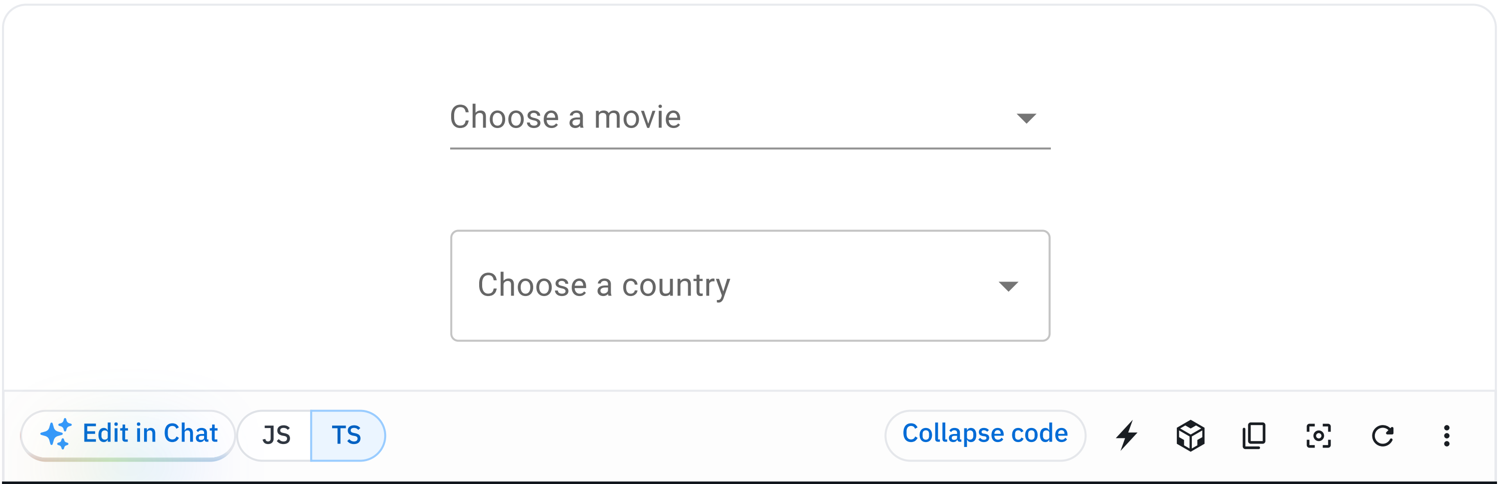
⚠ If you're using a custom input component inside the Autocomplete, make sure that you forward the ref to the underlying DOM element.



```
import Autocomplete from '@mui/material/Autocomplete';

const options = ['Option 1', 'Option 2'];

export default function CustomInputAutocomplete() {
  return (
    <label>
      Value:{' '}
      <Autocomplete
        sx={(theme) => ({
          display: 'inline-block',
          '& input': {
            width: 200,
            bgcolor: 'background.paper',
            color: theme.palette.getContrastText(theme.palette.background.paper),
          },
        })}
        id="custom-input-demo"
        options={options}
        renderInput={(params) => (
          <div ref={params.InputProps.ref}>
            <input type="text" {...params.inputProps} />
          </div>
        )}
      />
    </label>
  );
}
```

Globally customized options

To globally customize the Autocomplete options for all components in your app, you can use the theme default props and set the `renderOption` property in the `defaultProps` key. The `renderOption` property takes the `ownerState` as the fourth parameter, which includes props and internal component state. To display the label, you can use the `getOptionLabel` prop from the `ownerState`. This approach enables different options for each Autocomplete component while keeping the options styling consistent.

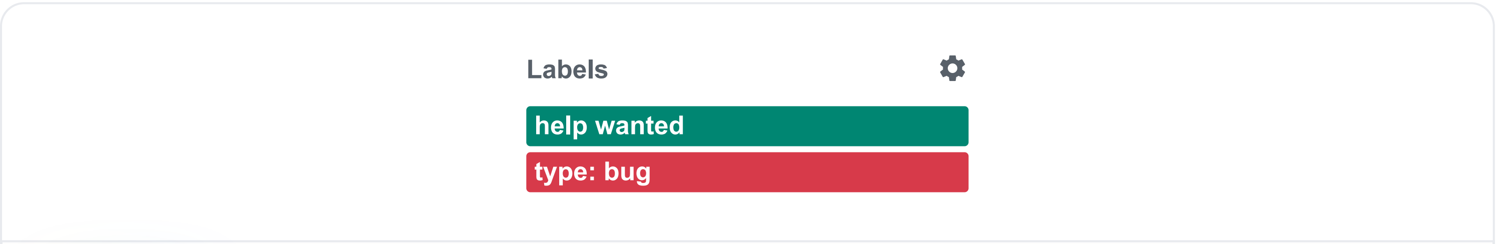


```
import Autocomplete, { autocompleteClasses } from '@mui/material/Autocomplete';
import Box from '@mui/material/Box';
import Stack from '@mui/material/Stack';
import TextField from '@mui/material/TextField';
import { createTheme, useTheme, ThemeProvider, Theme } from '@mui/material/styles';

// Theme.ts
const customTheme = (outerTheme: Theme) =>
```

```
  createTheme({
    cssVariables: {
      colorSchemeSelector: 'class',
    },
    palette: {
      mode: outerTheme.palette.mode,
    },
    components: {
      MuiAutocomplete: {
        defaultProps: {
          renderOption: (props, option, state, ownerState) => {
            const { key, ...optionProps } = props;
            return (
              <Box
                key={key}
                sx={{
                  borderRadius: '8px',
                  margin: '5px',
                  [`&.${autocompleteClasses.option}`]: {
                    padding: '8px',
                  },
                }}
                component="li"
                {...optionProps}
              >
                {ownerState.getOptionLabel(option)}
              </Box>
            );
          },
        },
```

GitHub's picker

This demo reproduces GitHub's label picker:



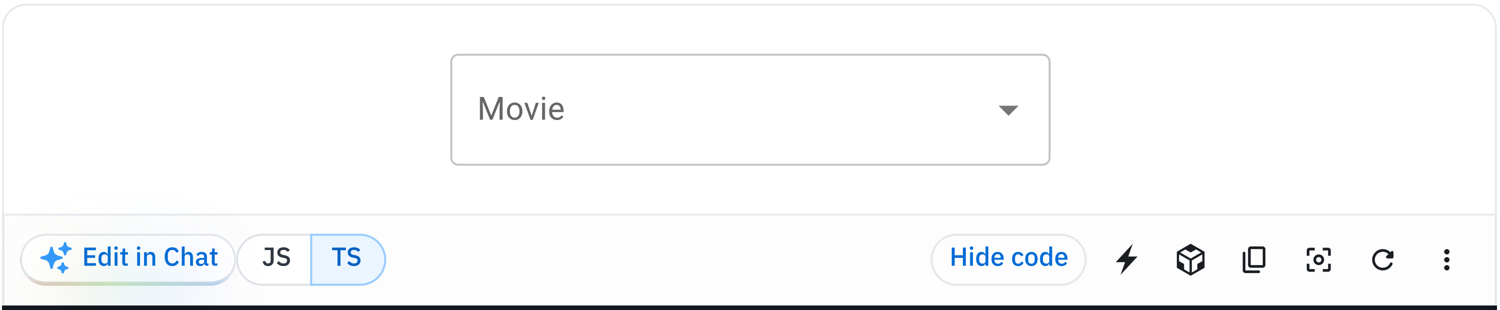
Edit in Chat JS TS Hide code

```
import * as React from 'react';
import { useTheme, styled } from '@mui/material/styles';
import Popper from '@mui/material/Popper';
import ClickAwayListener from '@mui/material/ClickAwayListener';
import SettingsIcon from '@mui/icons-material/Settings';
import CloseIcon from '@mui/icons-material/Close';
import DoneIcon from '@mui/icons-material/Done';
import Autocomplete, {
  AutocompleteCloseReason,
  autocompleteClasses,
} from '@mui/material/Autocomplete';
import ButtonBase from '@mui/material/ButtonBase';
import InputBase from '@mui/material/InputBase';
import Box from '@mui/material/Box';

interface PopperComponentProps {
  anchorEl?: any;
  disablePortal?: boolean;
  open: boolean;
}

const StyledAutocompletePopper = styled('div')(({ theme }) => ({
  [`& .${autocompleteClasses.paper}`]: {
    boxShadow: 'none',
    margin: 0,
    color: 'inherit',
    fontSize: 13,
  },
  [`& .${autocompleteClasses.listbox}`]: {
    padding: 0,
    backgroundColor: '#fff',
    ...theme.applyStyles('dark', {
      backgroundColor: '#1c2128',
    }),
    [`& .${autocompleteClasses.option}`]: {
      minHeight: 'auto',
      alignItems: 'flex-start',
      padding: 8
```

Head to the [Customized hook](#) section for a customization example with the `useAutocomplete` hook instead of the component.

Hint

The following demo shows how to add a hint feature to the Autocomplete:



```
import * as React from 'react';
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';
import Box from '@mui/material/Box';
import Typography from '@mui/material/Typography';

export default function AutocompleteHint() {
  const hint = React.useRef('');
  const [inputValue, setInputValue] = React.useState('');
  return (
    <Autocomplete
      onKeyDown={(event) => {
        if (event.key === 'Tab') {
          if (hint.current) {
            setInputValue(hint.current);
            event.preventDefault();
          }
        }
      }}
      onClose={() => {
        hint.current = '';
      }}
      onChange={(event, newValue) => {
        setInputValue(newValue && newValue.label ? newValue.label : '');
      }}
      disablePortal
      inputValue={inputValue}
      id="combo-box-hint-demo"
      options={top100Films}
      sx={{ width: 300 }}
      renderInput={(params) => {
        return (
          <Box sx={{ position: 'relative' }}>
            <Typography
              sx={{
                position: 'absolute',
                opacity: 0.5,
                left: 14
```

Highlights

The following demo relies on [autosuggest-highlight](), a small (1 kB) utility for highlighting text in autosuggest and autocomplete components.

Highlights

Edit in Chat JS TS Hide code

```
import TextField from '@mui/material/TextField';
import Autocomplete from '@mui/material/Autocomplete';
import parse from 'autosuggest-highlight/parse';
import match from 'autosuggest-highlight/match';

export default function Highlights() {
  return (
    <Autocomplete
```

```
      sx={{ width: 300 }}
      options={top100Films}
      getOptionLabel={(option) => option.title}
      renderInput={(params) => (
        <TextField {...params} label="Highlights" margin="normal" />
      )}
      renderOption={(props, option, { inputValue }) => {
        const { key, ...optionProps } = props;
        const matches = match(option.title, inputValue, { insideWords: true });
        const parts = parse(option.title, matches);

        return (
          <li key={key} {...optionProps}>
            <div>
              {parts.map((part, index) => (
                <span
                  key={index}
                  style={{
                    fontWeight: part.highlight ? 700 : 400,
                  }}
                >
                  {part.text}
                </span>
              ))}
            </div>
          </li>
        );
      }}
    />
```

Custom filter

The component exposes a factory to create a filter method that can be provided to the `filterOptions` prop. You can use it to change the default option filter behavior.

```
import { createFilterOptions } from '@mui/material/Autocomplete';
```

`createFilterOptions(config) => filterOptions`

Arguments

1. `config` (*object* [optional]):

- `config.ignoreAccents` (*bool* [optional]): Defaults to `true`. Remove diacritics.
- `config.ignoreCase` (*bool* [optional]): Defaults to `true`. Lowercase everything.
- `config.limit` (*number* [optional]): Default to null. Limit the number of suggested options to be shown. For example, if `config.limit` is `100`, only the first `100` matching options are shown. It can be useful if a lot of options match and virtualization wasn't set up.

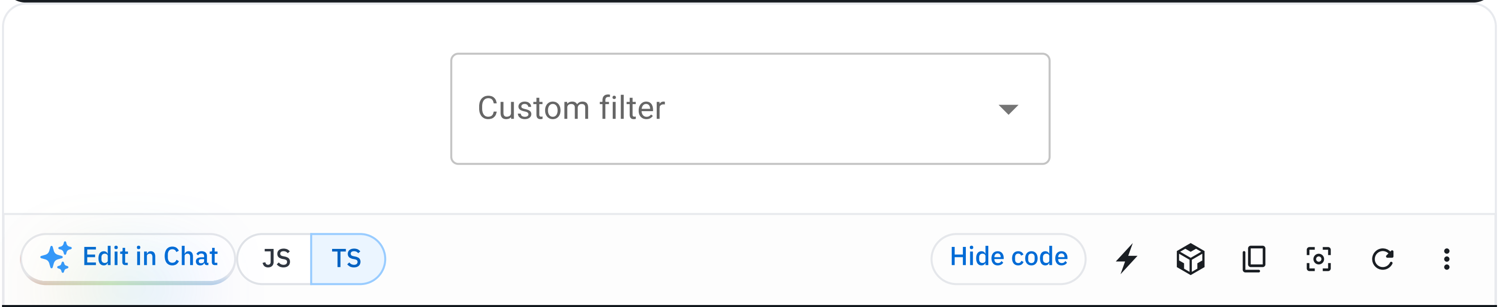
- `config.matchFrom` (*'any'* | *'start'* [optional]): Defaults to `'any'`.
- `config.stringify` (*func* [optional]): Controls how an option is converted into a string so that it can be matched against the input text fragment.
- `config.trim` (*bool* [optional]): Defaults to `false`. Remove trailing spaces.

Returns

`filterOptions`: the returned filter method can be provided directly to the `filterOptions` prop of the `Autocomplete` component, or the parameter of the same name for the hook.

In the following demo, the options need to start with the query prefix:

```
const filterOptions = createFilterOptions({
  matchFrom: 'start',
  stringify: (option) => option.title,
});

<Autocomplete filterOptions={filterOptions} />;
```



```
import TextField from '@mui/material/TextField';
import Autocomplete, { createFilterOptions } from '@mui/material/Autocomplete';

const filterOptions = createFilterOptions({
  matchFrom: 'start',
  stringify: (option: FilmOptionType) => option.title,
});

export default function Filter() {
  return (
    <Autocomplete
      options={top100Films}
      getOptionLabel={(option) => option.title}
      filterOptions={filterOptions}
      sx={{ width: 300 }}
      renderInput={(params) => <TextField {...params} label="Custom filter" />}
    />
  );
}

interface FilmOptionType {
  title: string;
  year: number;
}

// Top 100 films as rated by IMDb users. http://www.imdb.com/chart/top
const top100Films = [
  { title: 'The Shawshank Redemption', year: 1994 },
  { title: 'The Godfather', year: 1972 },
  { title: 'The Godfather: Part II', year: 1974 },
  { title: 'The Dark Knight', year: 2008 },
  { title: '12 Angry Men', year: 1957 },
  { title: "Schindler's List", year: 1993 },
  { title: 'Pulp Fiction', year: 1994 },
  {
    title: 'The Lord of the Rings: The Return of the King',
    year: 2003,
  },
```

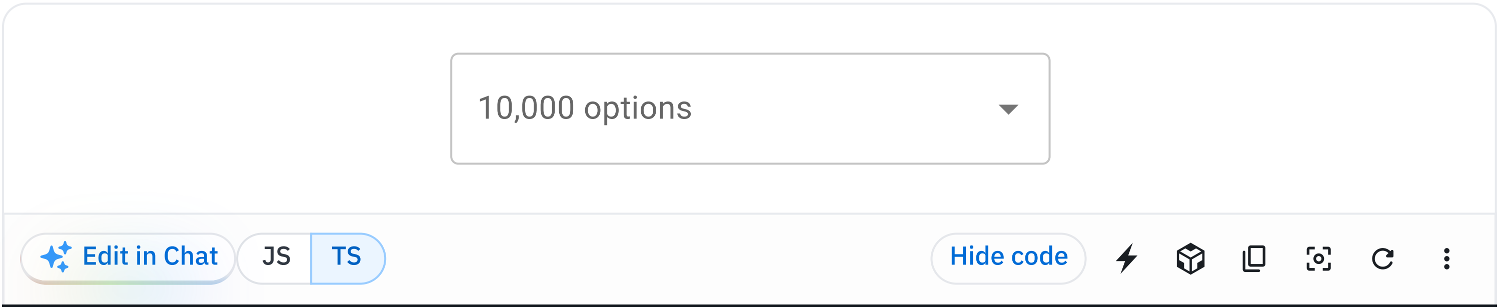
Advanced

For richer filtering mechanisms, like fuzzy matching, it's recommended to look at match-sorter. For instance:

```
import { matchSorter } from 'match-sorter';

const filterOptions = (options, { inputValue }) => matchSorter(options, inputValue);

<Autocomplete filterOptions={filterOptions} />;
```

Virtualization

Search within 10,000 randomly generated options. The list is virtualized thanks to react-window.



```
import * as React from 'react';
import TextField from '@mui/material/TextField';
import Autocomplete, { autocompleteClasses } from '@mui/material/Autocomplete';
import useMediaQuery from '@mui/material/useMediaQuery';
import ListSubheader from '@mui/material/ListSubheader';
import Popper from '@mui/material/Popper';
import { useTheme, styled } from '@mui/material/styles';
import {
  List,
  RowComponentProps,
  useListRef,
  ListImperativeAPI,
} from 'react-window';
import Typography from '@mui/material/Typography';
```

```
const LISTBOX_PADDING = 8; // px

type ItemData = Array<
  | {
      key: number;
      group: string;
      children: React.ReactNode;
    }
  | [React.ReactElement, string, number]
>;

function RowComponent({
  index,
  itemData,
  style,
}: RowComponentProps & {
  itemData: ItemData;
}) {
  const dataSet = itemData[index];
  const inlineStyle = {
    ...style,
    top: ((style.top as number) ?? 0) + LISTBOX_PADDING,
```

Events

If you would like to prevent the default key handler behavior, you can set the event's `defaultMuiPrevented` property to `true`:

```
<Autocomplete
  onKeyDown={(event) => {
    if (event.key === 'Enter') {
      // Prevent's default 'Enter' behavior.
      event.defaultMuiPrevented = true;
      // your handler code
    }
  }}
/>
```

Limitations

autocomplete/autofill

Browsers have heuristics to help the user fill in form inputs. However, this can harm the UX of the component.

By default, the component disables the input **autocomplete** feature (remembering what the user has typed for a given field in a previous session) with the `autoComplete="off"` attribute. Google Chrome does not currently support this attribute setting (Issue 41239842). A possible workaround is to remove the `id` to have the component generate a random one.

In addition to remembering past entered values, the browser might also propose **autofill** suggestions (saved login, address, or payment details). In the event you want the avoid autofill, you can try the following:

- Name the input without leaking any information the browser can use. For example `id="field1"` instead of `id="country"`. If you leave the id empty, the component uses a random id.
- Set `autoComplete="new-password"` (some browsers will suggest a strong password for inputs with this attribute setting):

  ```
  <TextField
    {...params}
    inputProps={{
      ...params.inputProps,
      autoComplete: 'new-password',
    }}
  />
  ```

Read the guide on MDN for more details.

iOS VoiceOver

VoiceOver on iOS Safari doesn't support the `aria-owns` attribute very well. You can work around the issue with the `disablePortal` prop.

ListboxComponent

If you provide a custom `ListboxComponent` prop, you need to make sure that the intended scroll container has the `role` attribute set to `listbox`. This ensures the correct behavior of the scroll, for example when using the keyboard to navigate.

Accessibility

(WAI-ARIA: https://www.w3.org/WAI/ARIA/apg/patterns/combobox/)

We encourage the usage of a label for the textbox. The component implements the WAI-ARIA authoring practices.

API

See the documentation below for a complete reference to all of the props and classes available to the components mentioned here.

- `<Autocomplete />`

- `<Popper />`
- `<TextField />`