

# Minimizing bundle size

Learn more about the tools you can leverage to reduce the bundle size.

## Bundle size matters

The bundle size of MUI is taken very seriously. Size snapshots are taken on every commit for every package and critical parts of those packages ([view the latest snapshot](#)). Combined with [dangerJS](#) we can inspect [detailed bundle size changes](#) on every Pull Request.

## When and how to use tree-shaking?

Tree-shaking of MUI works out of the box in modern frameworks. MUI exposes its full API on the top-level `@mui` imports. If you're using ES6 modules and a bundler that supports tree-shaking ( [webpack >= 2.x](#), [parcel](#) [with a flag](#)) you can safely use named imports and still get an optimized bundle size automatically:

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

⚠ The following instructions are only needed if you want to optimize your development startup times or if you are using an older bundler that doesn't support tree-shaking.

## Development environment

Development bundles can contain the full library which can lead to **slower startup times**. This is especially noticeable if you import from `@mui/icons-material`. Startup times can be approximately 6x slower than without named imports from the top-level API.

If this is an issue for you, you have various options:

### Option 1

You can use path imports to avoid pulling in unused modules. For instance, use:

```
// 🚀 Fast
import Button from '@mui/material/Button';
import TextField from '@mui/material/TextField';
```

instead of top-level imports (without a Babel plugin):


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

This is the option we document in all the demos since it requires no configuration. It is encouraged for library authors that are extending the components. Head to [Option 2](#) for the approach that yields the best DX and UX.

While importing directly in this manner doesn't use the exports in [the main file of @mui/material](#), this file can serve as a handy reference as to which modules are public.

Be aware that we only support first and second-level imports. Anything deeper is considered private and can cause issues, such as module duplication in your bundle.

```
//  OK
import { Add as AddIcon } from '@mui/icons-material';
import { Tabs } from '@mui/material';
//          ^^^^^^^ 1st or top-level

//  OK
import AddIcon from '@mui/icons-material/Add';
import Tabs from '@mui/material/Tabs';
//          ^^^^ 2nd level

//  NOT OK
import TabIndicator from '@mui/material/Tabs/TabIndicator';
//          ^^^^^^^^^^^^^ 3rd level
```

If you're using `eslint` you can catch problematic imports with the [no-restricted-imports rule](#). The following `.eslintrc` configuration will highlight problematic imports from `@mui` packages:

```
{
  "rules": {
    "no-restricted-imports": [
      "error",
      {
        "patterns": ["@mui/*//*", "!@mui/material/test-utils/*"]
      }
    ]
  }
}
```

## Option 2

This option provides the best User Experience and Developer Experience:

- UX: The Babel plugin enables top-level tree-shaking even if your bundler doesn't support it.
- DX: The Babel plugin makes startup time in dev mode as fast as Option 1.
- DX: This syntax reduces the duplication of code, requiring only a single import for multiple modules. Overall, the code is easier to read, and you are less likely to make a mistake when importing a new module.

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

However, you need to apply the two following steps correctly.

### 1. Configure Babel

Pick one of the following plugins:

- [babel-plugin-import](#) with the following configuration:

```
yarn add -D babel-plugin-import
```

Create a `.babelrc.js` file in the root directory of your project:

```
const plugins = [
  [
    'babel-plugin-import',
    {
      libraryName: '@mui/material',
      libraryDirectory: '',
      camel2DashComponentName: false,
    },
    'core',
  ],
  [
    'babel-plugin-import',
    {
      libraryName: '@mui/icons-material',
      libraryDirectory: '',
      camel2DashComponentName: false,
    },
    'icons',
  ],
];

module.exports = { plugins };
```

- [babel-plugin-direct-import](#) with the following configuration:

```
yarn add -D babel-plugin-direct-import
```

Create a `.babelrc.js` file in the root directory of your project:

```
const plugins = [
  [
    'babel-plugin-direct-import',
    { modules: ['@mui/material', '@mui/icons-material'] },
  ],
];

module.exports = { plugins };
```

If you are using Create React App, you will need to use a couple of projects that let you use `.babelrc` configuration, without ejecting.

```
yarn add -D react-app-rewired customize-cra
```

Create a `config-overrides.js` file in the root directory:

```
/* config-overrides.js */
/* eslint-disable react-hooks/rules-of-hooks */
const { useBabelRc, override } = require('customize-cra');

module.exports = override([useBabelRc]);
```

If you wish, `babel-plugin-import` can be configured through `config-overrides.js` instead of `.babelrc` by using this [configuration](#).

Modify your `package.json` commands:

```
"scripts": {  
  - "start": "react-scripts start",  
  + "start": "react-app-rewired start",  
  - "build": "react-scripts build",  
  + "build": "react-app-rewired build",  
  - "test": "react-scripts test",  
  + "test": "react-app-rewired test",  
  "eject": "react-scripts eject"  
}
```

Enjoy significantly faster start times.

## 2. Convert all your imports

Finally, you can convert your existing codebase to this option with this [top-level-imports codemod](#). It will perform the following diffs:

```
-import Button from '@mui/material/Button';  
-import TextField from '@mui/material/TextField';  
+import { Button, TextField } from '@mui/material';
```

## Available bundles

The package published on npm is **transpiled**, with [Babel](#), to take into account the [supported platforms](#).

⚠ Developers are **strongly discouraged** to import from any of the other bundles directly. Otherwise it's not guaranteed that dependencies used also use legacy or modern bundles. Instead, use these bundles at the bundler level with e.g [Webpack's](#) [resolve.alias](#) :

```
{  
  resolve: {  
    alias: {  
      '@mui/base': '@mui/base/legacy',  
      '@mui/lab': '@mui/lab/legacy',  
      '@mui/material': '@mui/material/legacy',  
      '@mui/styled-engine': '@mui/styled-engine/legacy',  
      '@mui/system': '@mui/system/legacy',  
    }  
  }  
}
```

## Modern bundle

The modern bundle can be found under the [/modern folder](#). It targets the latest released versions of evergreen browsers (Chrome, Firefox, Safari, Edge). This can be used to make separate bundles targeting different browsers.

## Legacy bundle

If you need to support IE 11 you cannot use the default or modern bundle without transpilation. However, you can use the legacy bundle found under the [/legacy folder](#). You don't need any additional polyfills.