> ***Note:*** *Dynamic imports require Meteor 1.5 or higher.*

The `dynamic-import` package provides an implementation of `Module.prototype.dynamicImport`, an extension of the module runtime which powers the dynamic `import(...)` statement, an up-and-coming (ECMA2020) addition to the ECMAScript standard.

The dynamic `import(...)` statement is a complementary method to the static `import` technique of requiring a module. While a statically `import`-ed module would be bundled into the initial JavaScript bundle, a dynamically `import()`-ed module is fetched from the server at runtime.

Once a module is fetched dynamically from the server, it is cached permanently on the client and additional requests for the same version of the module will not incur the round-trip request to the server. If the module is changed then a fresh copy will always be retrieved from the server.

Usage

The `import(...)` statement returns a `Promise` which is resolved with the `exports` of the module when it has been successfully fetched from the server and is ready to be used.

Because it's a `Promise`, there are a couple methods developers can use to dictate what will happen upon the availability of the dynamically loaded module:

The `.then()` method of the `Promise`

```
import("tool").then(tool => tool.task());
```

By `await`-ing in an asynchronous function

Meteor supports `async` and `await`, which provide a straightforward approach to asynchronously wait for the module to be ready without the need to provide a callback:

```
async function performTask() {
  const tool = await import("tool");
  tool.task();
}
```

{% blockquote %} **Default exports**

The `import(...)` `Promise` is resolved with the `exports` of the module. If it's necessary to use the "default" export from a module, it will be available on the `default` property of the resulting object. In the above examples, this means it will be available as `tool.default`. It can be helpful to use parameter de-structuring to provide additional clarity:

```
import("another-tool").then(({ default: thatTool }) => thatTool.go());
```

{% endblockquote %}

Using `import()` with dynamic expressions

If you try to import using any computed expression, such as:

```
let path = 'example';
const module = await import(`/libs/${path}.js`);
```

You'll get an error like so:

```
Error: Cannot find module '/libs/example.js'
```

Meteor’s build process builds a graph of all files that are imported or required using static analysis. It then creates exact bundles of the referenced files and makes them available to the client for `import()`.

Without a complete import statement (static, dynamic or `require`), Meteor won't make that module available for `import()`.

The solution to make dynamic expressions work is to create a module "whitelist" that can be read by the build process, but does not actually run. For example:

```
if (false) {
  import("/libs/example.js");
  import("/libs/another-example.js");
  import("/libs/yet-another-example.js");
}
```

Make sure the whitelist is imported from both the client and server entry points.

Difference to other bundling systems

In Meteor's implementation, the client has perfect information about which modules were in the initial bundle, which modules are in the local cache, and which modules still need to be fetched. There is never any overlap between requests made by a single client, nor will there be any unneeded modules in the response from the server. You might call this strategy **exact code splitting**, to differentiate it from bundling.

Moreover, the initial bundle includes the hashes of all available dynamic modules, so the client doesn't have to ask the server if it can use a cached version of a module, and the same version of the module never needs to be downloaded again by the same client. This caching system has all the benefits of immutable caching.

Meteor also allows dynamic expressions as long as the dependency is expressed statically somewhere else in your code. This is possible because Meteor's client-side module system understands how to resolve dynamic strings at runtime (which is not true in webpack or browserify, because they replace module identifier strings with numbers). However, the set of available modules is constrained by the string literals that you, the programmer, explicitly decided to allow to be imported (either directly or in a whitelist).