# Getting Started

This guide will help you get the most out of the Mindee Node.js client library to easily extract data from your documents.

> 📘 **Info**
>
> The library is written in TypeScript for your coding pleasure and is officially supported on all current LTS versions of Node.js.
> All examples shown in this guide should work in both TypeScript and JavaScript.

## Installation

### Prerequisites

You'll need npm and Node.js.

**Note**: When installing a Node.js instance, nvm will also install a compatible npm version.

## Standard Installation

The easiest way to install the Mindee client library for your project is by using npm:

Shell

```
npm install mindee
```

## Development Installation

If you'll be modifying the source code, you'll need to follow these steps to get started.

1. First clone the repo.

Shell

```
git clone git@github.com:mindee/mindee-api-nodejs.g
```

2. Navigate to the cloned directory and install all required libraries.

Shell

```
cd mindee-api-node.js
npm install
```

# Updating the Library

It is important to always check the version of the Mindee client library you are using, as new and updated features won’t work on older versions.

To get the latest version:

Shell

```
npm update mindee
```

To install a specific version of Mindee:

Shell

```
npm install mindee@<version>
```

## Usage

Using Mindee's APIs can be broken down into the following steps:

1. Import the required classes in your program
2. Initialize a `Client`
3. Load a file
4. Send the file to Mindee's API
5. Retrieve the response
6. Process the response in some way

Let's take a deep dive into how this works.

## Importing Requirements

In most cases, you'll just need to `require` the `mindee` module:

JavaScript

```javascript
const mindee = require("mindee");
```

If you're building your own module, or using TypeScript, the equivalent would be:

ts

```ts
import * as mindee from "mindee";
```

## Initializing the Client

The `Client` centralizes document configurations in a single object.

The `Client` requires your API key.

You can either pass these directly to the constructor or through environment variables.

### Pass the API key directly

ts

```ts
// Init a new client and passing the key directly
```

```
const mindeeClient = new mindee.Client({apiKey: "my
```

### Set the API key in the environment

API keys should be set as environment variables, especially for any production deployment.

The following environment variable will set the global API key:

Shell

```shell
MINDEE_API_KEY="my-api-key"
```

Then in your code:

ts

```ts
// Init a new client without an API key
const mindeeClient = new Client();
```

## Loading a Document File

Before being able to send a document to the API, it must first be loaded.

You don't need to worry about different MIME types, the library will take care of handling
all supported types automatically.

Once a document is loaded, interacting with it is done in exactly the same way, regardless of how it was loaded.

There are a few different ways of loading a document file, depending on your use case:

- Path
- File Object
- Base64
- URL

## Path

Load from a file directly from disk. Requires an absolute path, as a string.

ts

```ts
const inputSource = mindeeClient.docFromPath("/path
```

## Stream Object

Load a standard readable stream object, for example as returned by the `fs.createReadStream()` function.

**Note**: The original filename is required when calling the method.

ts

```ts
const stream = fs.createReadStream("/path/to/the/do
const inputSource = mindeeClient.docFromStream(stre
```

### Base64

Load file contents from a base64-encoded string.

**Note**: The original filename is required when calling the method.

ts

```ts
const b64String = "/9j/4AAQSkZJRgABAQAAAQABAAD/2wBD
const inputSource = mindeeClient.docFromBase64(b64S
```

### URL

Specify a URL to send to the Mindee API.

**Note**: The URL will not be downloaded locally, so checks (i.e. MIME type) and transformations (i.e. remove pages from a PDF) will not be possible.

ts

```ts
const inputSource = mindeeClient.docFromUrl("https:
```

## Sending a Document

To send a file to the API, we need to specify how to process the document.
This will determine which API endpoint is used and how the API return will be handled internally by the library.

More specifically, we need to set a `Product` class as the first parameter of the `parse` method.

This is because the `parse` method is generic, and its return type depends on its first argument.

Each document type available in the library has its corresponding class, which inherit from the base `Document` class.
This is detailed in each document-specific guide.

## Off-the-Shelf Documents

Simply setting the correct class and passing the document is enough:

ts

```ts
const respPromise = mindeeClient.parse(mindee.Invo
```

## Custom Documents

The endpoint to use must also be set in third argument of the `parse` method. This argument is an object:

ts

```ts
const customEndpoint = mindeeClient.createEndpoint(
  "my-endpoint",
  "my-account",
  // "my-version" // Optional: set the version, de
);
```

```
const respPromise = mindeeClient.parse(
  mindee.CustomV1,
  inputSource,
  {
    endpoint: customEndpoint
  }
);
```

This is because the `CustomV1` class is enough to handle the return processing, but the actual endpoint needs to be specified.

## Retrieving the Response

The return of the `parse` method is a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) that resolves to a `Response` object.

More technically, the return object is instantiated depending on the specific `Document` class passed as the first argument to `parse`.

Handling the return is done like any other Promise:

ts

```ts
respPromise.then((resp) => {
  console.log(resp.document);
});
```

Some other styles:

ts

```
// One-liner
mindeeClient.parse(mindee.InvoiceV4, inputSource).t
  console.log(resp.document);
});

// Async function
async function parseInvoice() {
  const inputSource = mindeeClient.docFromPath("/pa
  const response = await mindeeClient.parse(mindee.
}
```

## Processing the Response

The `Response` objects all have the following attributes:

- `document` — [Document level prediction]
- `pages` — [Page level prediction]

### Document Level Prediction

The `document` attribute is an object specific to the type of document being processed.
It is an instance of the `Document` class, to which a generic type is given.

It contains the data extracted from the entire document, all pages combined.
It's possible to have the same field in various pages, but at the document level only the highest confidence field data will be shown (this is all done automatically at the API level).

ts

```ts
// print a summary of the document-level info
console.log(resp.document.toString());
// or
console.log(`${resp.document}`);
```

A `document` 's fields (attributes) can be accessed through it's `prediction` attribute, which have types that can vary from one product to another.
These attributes are detailed in each product's respective guide.

## Page Level Prediction

The `pages` attribute is an array of `Page` objects. `Page` is a wrapper around elements that extend the [`Document` class](). The `prediction` of a `Page` inherits from the product's own `Document` , and adds all page-specific fields to it.

The order of the elements in the array matches the order of the pages in the document.

All response objects have a `pages` property, regardless of the number of pages.
Single-page documents will have a single entry.

Iteration over `pages` is done like with any JavaScript array, for example:

ts

```
resp.pages.forEach((page) => {
  console.log(page.toString());
});
```

## Questions?

Join our Slack

← Node.js OCR SDK

Custom OCR Node.js →