**Apify Platform**

Apify is a [platform](https://apify.com/) built to serve large-scale and high-performance web scraping and automation needs. It provides easy access to [compute instances (Actors)](https://docs.apify.com/sdk/js/docs/guides/apify-platform#what-is-an-actor), convenient [request](https://docs.apify.com/sdk/js/docs/guides/request-storage) and [result](https://docs.apify.com/sdk/js/docs/guides/result-storage) storages, [proxies](https://docs.apify.com/sdk/js/docs/guides/proxy-management), [scheduling](https://docs.apify.com/scheduler), [webhooks](https://docs.apify.com/webhooks) and [more](https://docs.apify.com/), accessible through a [web interface](https://console.apify.com/) or an [API](https://docs.apify.com/api).

While we think that the Apify platform is super cool, and it's definitely worth signing up for a [free account](https://console.apify.com/sign-up), **Crawlee is and will always be open source**, runnable locally or on any cloud infrastructure.

**note**

We do not test Crawlee in other cloud environments such as Lambda or on specific architectures such as Raspberry PI. We strive to make it work, but there are no guarantees.

**Logging into Apify platform from Crawlee[​](https://docs.apify.com/sdk/js/docs/guides/apify-platform" \l "logging-into-apify-platform-from-crawlee" \o "Direct link to heading)**

To access your [Apify account](https://console.apify.com/sign-up" \t "_blank) from Crawlee, you must provide credentials - your [API token](https://console.apify.com/account?tab=integrations). You can do that either by utilizing [Apify CLI](https://github.com/apify/apify-cli" \t "_blank) or with environment variables.

Once you provide credentials to your scraper, you will be able to use all the Apify platform features, such as calling actors, saving to cloud storages, using Apify proxies, setting up webhooks and so on.

**Log in with CLI**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#log-in-with-cli)

Apify CLI allows you to log in to your Apify account on your computer. If you then run your scraper using the CLI, your credentials will automatically be added.

npm install -g apify-cli  
apify login -t YOUR\_API\_TOKEN

**Log in with environment variables**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#log-in-with-environment-variables)

Alternatively, you can always provide credentials to your scraper by setting the [APIFY\_TOKEN](https://docs.apify.com/sdk/js/docs/guides/apify-platform#apify_token) environment variable to your API token.

There's also the [APIFY\_PROXY\_PASSWORD](https://docs.apify.com/sdk/js/docs/guides/apify-platform#apify_proxy_password) environment variable. Actor automatically infers that from your token, but it can be useful when you need to access proxies from a different account than your token represents.

**Log in with Configuration**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#log-in-with-configuration)

Another option is to use the [Configuration](https://sdk.apify.com/api/apify/class/Configuration) instance and set your api token there.

import { Actor } from 'apify';  
  
const sdk = new Actor({ token: 'your\_api\_token' });

**What is an actor**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#what-is-an-actor)

When you deploy your script to the Apify platform, it becomes an [actor](https://apify.com/actors). An actor is a serverless microservice that accepts an input and produces an output. It can run for a few seconds, hours or even infinitely. An actor can perform anything from a simple action such as filling out a web form or sending an email, to complex operations such as crawling an entire website and removing duplicates from a large dataset.

Actors can be shared in the [Apify Store](https://apify.com/store" \t "_blank) so that other people can use them. But don't worry, if you share your actor in the store and somebody uses it, it runs under their account, not yours.

**Related links**

* [Store of existing actors](https://apify.com/store)
* [Documentation](https://docs.apify.com/actors)
* [View actors in Apify Console](https://console.apify.com/actors)
* [API reference](https://apify.com/docs/api/v2#/reference/actors)

**Running an actor locally**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#running-an-actor-locally)

First let's create a boilerplate of the new actor. You could use Apify CLI and just run:

apify create my-hello-world

The CLI will prompt you to select a project boilerplate template - let's pick "Hello world". The tool will create a directory called my-hello-world with a Node.js project files. You can run the actor as follows:

cd my-hello-world  
apify run

**Running Crawlee code as an actor**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#running-crawlee-code-as-an-actor)

For running Crawlee code as an actor on [Apify platform](https://apify.com/actors" \t "_blank) you should either:

* use a combination of [Actor.init()](https://sdk.apify.com/api/apify/class/Actor" \l "init" \t "_blank) and [Actor.exit()](https://sdk.apify.com/api/apify/class/Actor" \l "exit" \t "_blank) functions;
* or wrap it into [Actor.main()](https://sdk.apify.com/api/apify/class/Actor" \l "main" \t "_blank) function.

**NOTE**

* Adding [Actor.init()](https://sdk.apify.com/api/apify/class/Actor" \l "init" \t "_blank) and [Actor.exit()](https://sdk.apify.com/api/apify/class/Actor" \l "exit" \t "_blank) to your code are the only two important things needed to run it on Apify platform as an actor. Actor.init() is needed to initialize your actor (e.g. to set the correct storage implementation), while without Actor.exit() the process will simply never stop.
* [Actor.main()](https://sdk.apify.com/api/apify/class/Actor#main) is an alternative to Actor.init() and Actor.exit() as it calls both behind the scenes.

Let's look at the CheerioCrawler example from the [Quick Start](https://crawlee.dev/docs/quick-start) guide:

* **Using Actor.main()**
* **Using Actor.init() and Actor.exit()**

import { Actor } from 'apify';  
import { CheerioCrawler } from 'crawlee';  
  
await Actor.main(async () => {  
 const crawler = new CheerioCrawler({  
 async requestHandler({ request, $, enqueueLinks }) {  
 const { url } = request;  
  
 *// Extract HTML title of the page.*  
 const title = $('title').text();  
 console.log(`Title of ${url}: ${title}`);  
  
 *// Add URLs that match the provided pattern.*  
 await enqueueLinks({  
 globs: ['https://www.iana.org/\*'],  
 });  
  
 *// Save extracted data to dataset.*  
 await Actor.pushData({ url, title });  
 },  
 });  
  
 *// Enqueue the initial request and run the crawler*  
 await crawler.run(['https://www.iana.org/']);  
});

Note that you could also run your actor (that is using Crawlee) locally with Apify CLI. You could start it via the following command in your project folder:

apify run

**Deploying an actor to Apify platform**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#deploying-an-actor-to-apify-platform)

Now (assuming you are already logged in to your Apify account) you can easily deploy your code to the Apify platform by running:

apify push

Your script will be uploaded to and built on the Apify platform so that it can be run there. For more information, view the [Apify Actor](https://docs.apify.com/cli" \t "_self) documentation.

**Usage on Apify platform**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#usage-on-apify-platform)

You can also develop your actor in an online code editor directly on the platform (you'll need an Apify Account). Let's go to the [Actors](https://console.apify.com/actors) page in the app, click *Create new* and then go to the *Source* tab and start writing the code or paste one of the examples from the [Examples](https://docs.apify.com/sdk/js/docs/examples) section.

**Storages**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#storages)

There are several things worth mentioning here.

**Helper functions for default Key-Value Store and Dataset**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#helper-functions-for-default-key-value-store-and-dataset)

To simplify access to the *default* storages, instead of using the helper functions of respective storage classes, you could use:

* [Actor.setValue()](https://sdk.apify.com/api/apify/class/Actor#setValue), [Actor.getValue()](https://sdk.apify.com/api/apify/class/Actor#getValue), [Actor.getInput()](https://sdk.apify.com/api/apify/class/Actor#getInput) for Key-Value Store
* [Actor.pushData()](https://sdk.apify.com/api/apify/class/Actor#pushData) for Dataset

**Using platform storage in a local actor**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#using-platform-storage-in-a-local-actor)

When you plan to use the platform storage while developing and running your actor locally, you should use [Actor.openKeyValueStore()](https://sdk.apify.com/api/apify/class/Actor#openKeyValueStore), [Actor.openDataset()](https://sdk.apify.com/api/apify/class/Actor#openDataset) and [Actor.openRequestQueue()](https://sdk.apify.com/api/apify/class/Actor#openRequestQueue) to open the respective storage.

Using each of these methods allows to pass the [OpenStorageOptions](https://sdk.apify.com/api/apify/interface/OpenStorageOptions" \t "_blank) as a second argument, which has only one optional property: [forceCloud](https://sdk.apify.com/api/apify/interface/OpenStorageOptions" \l "forceCloud" \t "_blank). If set to true - cloud storage will be used instead of the folder on the local disk.

**note**

If you don't plan to force usage of the platform storages when running the actor locally, there is no need to use the [Actor](https://sdk.apify.com/api/apify/class/Actor) class for it. The Crawlee variants [KeyValueStore.open()](https://docs.apify.com/sdk/js/reference/class/KeyValueStore#open), [Dataset.open()](https://docs.apify.com/sdk/js/reference/class/Dataset#open) and [RequestQueue.open()](https://docs.apify.com/sdk/js/reference/class/RequestQueue#open) will work the same.

**Getting public url of an item in the platform storage**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#getting-public-url-of-an-item-in-the-platform-storage)

If you need to share a link to some file stored in a Key-Value Store on Apify Platform, you can use [getPublicUrl()](https://sdk.apify.com/api/apify/class/KeyValueStore" \l "getPublicUrl" \t "_blank) method. It accepts only one parameter: key - the key of the item you want to share.

import { KeyValueStore } from 'apify';  
  
const store = await KeyValueStore.open();  
await store.setValue('your-file', { foo: 'bar' });  
const url = store.getPublicUrl('your-file');  
*// https://api.apify.com/v2/key-value-stores/<your-store-id>/records/your-file*

**Exporting dataset data**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#exporting-dataset-data)

When the [Dataset](https://docs.apify.com/sdk/js/reference/class/Dataset) is stored on the [Apify platform](https://apify.com/actors" \t "_blank), you can export its data to the following formats: HTML, JSON, CSV, Excel, XML and RSS. The datasets are displayed on the actor run details page and in the [Storage](https://console.apify.com/storage) section in the Apify Console. The actual data is exported using the [Get dataset items](https://apify.com/docs/api/v2#/reference/datasets/item-collection/get-items) Apify API endpoint. This way you can easily share the crawling results.

**Related links**

* [Apify platform storage documentation](https://docs.apify.com/storage)
* [View storage in Apify Console](https://console.apify.com/storage)
* [Key-value stores API reference](https://apify.com/docs/api/v2#/reference/key-value-stores)
* [Datasets API reference](https://docs.apify.com/api/v2#/reference/datasets)
* [Request queues API reference](https://docs.apify.com/api/v2#/reference/request-queues)

**Environment variables**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#environment-variables)

The following are some additional environment variables specific to Apify platform. More Crawlee specific environment variables could be found in the [Environment Variables](https://crawlee.dev/docs/guides/configuration#environment-variables) guide.

**note**

It's important to notice that CRAWLEE\_ environment variables don't need to be replaced with equivalent APIFY\_ ones. Likewise, Crawlee understands APIFY\_ environment variables after calling Actor.init() or when using Actor.main().

**APIFY\_TOKEN**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#apify_token)

The API token for your Apify account. It is used to access the Apify API, e.g. to access cloud storage or to run an actor on the Apify platform. You can find your API token on the [Account Settings / Integrations](https://console.apify.com/account?tab=integrations) page.

**Combinations of APIFY\_TOKEN and CRAWLEE\_STORAGE\_DIR**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#combinations-of-apify_token-and-crawlee_storage_dir)

CRAWLEE\_STORAGE\_DIR env variable description could be found in [Environment Variables](https://crawlee.dev/docs/guides/configuration#crawlee_storage_dir) guide.

By combining the env vars in various ways, you can greatly influence the actor's behavior.

| **Env Vars** | **API** | **Storages** |
| --- | --- | --- |
| none OR CRAWLEE\_STORAGE\_DIR | no | local |
| APIFY\_TOKEN | yes | Apify platform |
| APIFY\_TOKEN AND CRAWLEE\_STORAGE\_DIR | yes | local + platform |

When using both APIFY\_TOKEN and CRAWLEE\_STORAGE\_DIR, you can use all the Apify platform features and your data will be stored locally by default. If you want to access platform storages, you can use the { forceCloud: true } option in their respective functions.

import { Actor } from 'apify';  
import { Dataset } from 'crawlee';  
  
*// or Dataset.open('my-local-data')*  
const localDataset = await Actor.openDataset('my-local-data');  
*// but here we need the `Actor` class*  
const remoteDataset = await Actor.openDataset('my-dataset', { forceCloud: true });

**APIFY\_PROXY\_PASSWORD**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#apify_proxy_password)

Optional password to [Apify Proxy](https://docs.apify.com/proxy" \t "_self) for IP address rotation. Assuming Apify Account was already created, you can find the password on the [Proxy page](https://console.apify.com/proxy) in the Apify Console. The password is automatically inferred using the APIFY\_TOKEN env var, so in most cases, you don't need to touch it. You should use it when, for some reason, you need access to Apify Proxy, but not access to Apify API, or when you need access to proxy from a different account than your token represents.

**Proxy management**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#proxy-management)

In addition to your own proxy servers and proxy servers acquired from third-party providers used together with Crawlee, you can also rely on [Apify Proxy](https://apify.com/proxy" \t "_blank) for your scraping needs.

**Apify Proxy**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#apify-proxy)

If you are already subscribed to Apify Proxy, you can start using them immediately in only a few lines of code (for local usage you first should be [logged in](https://docs.apify.com/sdk/js/docs/guides/apify-platform#logging-into-apify-platform-from-crawlee) to your Apify account.

import { Actor } from 'apify';  
  
const proxyConfiguration = await Actor.createProxyConfiguration();  
const proxyUrl = await proxyConfiguration.newUrl();

Note that unlike using your own proxies in Crawlee, you shouldn't use the constructor to create [ProxyConfiguration](https://docs.apify.com/sdk/js/reference/class/ProxyConfiguration) instance. For using Apify Proxy you should create an instance using the [Actor.createProxyConfiguration()](https://sdk.apify.com/api/apify/class/Actor" \l "createProxyConfiguration" \t "_blank) function instead.

**Apify Proxy Configuration**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#apify-proxy-configuration)

With Apify Proxy, you can select specific proxy groups to use, or countries to connect from. This allows you to get better proxy performance after some initial research.

import { Actor } from 'apify';  
  
const proxyConfiguration = await Actor.createProxyConfiguration({  
 groups: ['RESIDENTIAL'],  
 countryCode: 'US',  
});  
const proxyUrl = await proxyConfiguration.newUrl();

Now your crawlers will use only Residential proxies from the US. Note that you must first get access to a proxy group before you are able to use it. You can check proxy groups available to you in the [proxy dashboard](https://console.apify.com/proxy).

**Apify Proxy vs. Own proxies**[**​**](https://docs.apify.com/sdk/js/docs/guides/apify-platform#apify-proxy-vs-own-proxies)

The ProxyConfiguration class covers both Apify Proxy and custom proxy URLs so that you can easily switch between proxy providers. However, some features of the class are available only to Apify Proxy users, mainly because Apify Proxy is what one would call a super-proxy. It's not a single proxy server, but an API endpoint that allows connection through millions of different IP addresses. So the class essentially has two modes: Apify Proxy or Own (third party) proxy.

The difference is easy to remember.

* If you're using your own proxies - you should create an instance with the ProxyConfiguration [constructor](https://docs.apify.com/sdk/js/reference/class/ProxyConfiguration#constructor) function based on the provided [ProxyConfigurationOptions](https://docs.apify.com/sdk/js/reference/interface/ProxyConfigurationOptions).
* If you are planning to use Apify Proxy - you should create an instance using the [Actor.createProxyConfiguration()](https://sdk.apify.com/api/apify/class/Actor#createProxyConfiguration) function. [ProxyConfigurationOptions.proxyUrls](https://docs.apify.com/sdk/js/reference/interface/ProxyConfigurationOptions#proxyUrls) and [ProxyConfigurationOptions.newUrlFunction](https://docs.apify.com/sdk/js/reference/interface/ProxyConfigurationOptions#newUrlFunction) enable use of your custom proxy URLs, whereas all the other options are there to configure Apify Proxy.

**equest Storage**

The Apify SDK has several request storage types that are useful for specific tasks. The requests are stored either on local disk to a directory defined by the APIFY\_LOCAL\_STORAGE\_DIR environment variable, or on the [Apify platform](https://docs.apify.com/sdk/js/docs/guides/apify-platform) under the user account identified by the API token defined by the APIFY\_TOKEN environment variable. If neither of these variables is defined, by default Apify SDK sets APIFY\_LOCAL\_STORAGE\_DIR to ./storage in the current working directory and prints a warning.

Typically, you will be developing the code on your local computer and thus set the APIFY\_LOCAL\_STORAGE\_DIR environment variable. Once the code is ready, you will deploy it to the Apify platform, where it will automatically set the APIFY\_TOKEN environment variable and thus use cloud storage. No code changes are needed.

**Request queue**[**​**](https://docs.apify.com/sdk/js/docs/guides/request-storage#request-queue)

The request queue is a storage of URLs to crawl. The queue is used for the deep crawling of websites, where you start with several URLs and then recursively follow links to other pages. The data structure supports both breadth-first and depth-first crawling orders.

Each actor run is associated with a **default request queue**, which is created exclusively for the actor run. Typically, it is used to store URLs to crawl in the specific actor run. Its usage is optional.

In Apify SDK, the request queue is represented by the [RequestQueue](https://docs.apify.com/sdk/js/reference/class/RequestQueue) class.

In local configuration, the request queue is emulated by [@apify/storage-local](https://github.com/apify/apify-storage-local-js) NPM package and its data is stored in SQLite database in the directory specified by the APIFY\_LOCAL\_STORAGE\_DIR environment variable as follows:

{APIFY\_LOCAL\_STORAGE\_DIR}/request\_queues/{QUEUE\_ID}/db.sqlite

Note that {QUEUE\_ID} is the name or ID of the request queue. The default queue has ID default, unless you override it by setting the APIFY\_DEFAULT\_REQUEST\_QUEUE\_ID environment variable.

The following code demonstrates basic operations of the request queue:

*// Open the default request queue associated with the actor run*  
const requestQueue = await RequestQueue.open();  
*// Enqueue the initial request*  
await requestQueue.addRequest({ url: 'https://example.com' });  
  
*// The crawler will automatically process requests from the queue*  
const crawler = new CheerioCrawler({  
 requestQueue,  
 handlePageFunction: async ({ $, request }) => {  
 *// Add new request to the queue*  
 await requestQueue.addRequest({ url: 'https://example.com/new-page' });  
 *// Add links found on page to the queue*  
 await Actor.utils.enqueueLinks({ $, requestQueue });  
 },  
});

To see more detailed example of how to use the request queue with a crawler, see the [Puppeteer Crawler](https://docs.apify.com/sdk/js/docs/examples/puppeteer-crawler) example.

**Request list**[**​**](https://docs.apify.com/sdk/js/docs/guides/request-storage#request-list)

The request list is not a storage per se - it represents the list of URLs to crawl that is stored in a run memory (or optionally in default [Key-Value Store](https://docs.apify.com/sdk/js/docs/guides/result-storage#key-value-store) associated with the run, if specified). The list is used for the crawling of a large number of URLs, when you know all the URLs which should be visited by the crawler and no URLs would be added during the run. The URLs can be provided either in code or parsed from a text file hosted on the web.

Request list is created exclusively for the actor run and only if its usage is explicitly specified in the code. Its usage is optional.

In Apify SDK, the request list is represented by the [RequestList](https://crawlee.dev/api/core/class/RequestList" \t "_blank) class.

The following code demonstrates basic operations of the request list:

*// Prepare the sources array with URLs to visit*  
const sources = [  
 { url: 'http://www.example.com/page-1' },  
 { url: 'http://www.example.com/page-2' },  
 { url: 'http://www.example.com/page-3' },  
];  
*// Open the request list.*  
*// List name is used to persist the sources and the list state in the key-value store*  
const requestList = await RequestList.open('my-list', sources);  
  
*// The crawler will automatically process requests from the list*  
const crawler = new PuppeteerCrawler({  
 requestList,  
 handlePageFunction: async ({ page, request }) => {  
 *// Process the page (extract data, take page screenshot, etc).*  
 *// No more requests could be added to the request list here*  
 },  
});

To see more detailed example of how to use the request list with a crawler, see the [Puppeteer with proxy](https://docs.apify.com/sdk/js/docs/examples/puppeteer-with-proxy) example.

**Which one to choose?**[**​**](https://docs.apify.com/sdk/js/docs/guides/request-storage#which-one-to-choose)

When using Request queue - you would normally have several start URLs (e.g. category pages on e-commerce website) and then recursively add more (e.g. individual item pages) programmatically to the queue, it supports dynamic adding and removing of requests. No more URLs can be added to Request list after its initialization as it is immutable, URLs cannot be removed from the list either.

On the other hand, the Request queue is not optimized for adding or removing numerous URLs in a batch. This is technically possible, but requests are added one by one to the queue, and thus it would take significant time with a larger number of requests. Request list however can contain even millions of URLs, and it would take significantly less time to add them to the list, compared to the queue.

Note that Request queue and Request list can be used together by the same crawler. In such cases, each request from the Request list is enqueued into the Request queue first (to the foremost position in the queue, even if Request queue is not empty) and then consumed from the latter. This is necessary to avoid the same URL being processed more than once (from the list first and then possibly from the queue). In practical terms, such a combination can be useful when there are numerous initial URLs, but more URLs would be added dynamically by the crawler.

The following code demonstrates how to use Request queue and Request list in the same crawler:

*// Prepare the sources array with URLs to visit (it can contain millions of URLs)*  
const sources = [  
 { url: 'http://www.example.com/page-1' },  
 { url: 'http://www.example.com/page-2' },  
 { url: 'http://www.example.com/page-3' },  
];  
*// Open the request list*  
const requestList = await RequestList.open('my-list', sources);  
  
*// Open the default request queue. It's not necessary to add any requests to the queue*  
const requestQueue = await RequestQueue.open();  
  
*// The crawler will automatically process requests from the list and the queue*  
const crawler = new PuppeteerCrawler({  
 requestList,  
 requestQueue,  
 *// Each request from the request list is enqueued to the request queue one by one.*  
 *// At this point request with the same URL would exist in the list and the queue*  
 handlePageFunction: async ({ request, page }) => {  
 *// Add new request to the queue*  
 await requestQueue.addRequest({ url: 'http://www.example.com/new-page' });  
  
 *// Add links found on page to the queue*  
 await Actor.utils.enqueueLinks({ page, requestQueue });  
  
 *// The requests above would be added to the queue (but not to the list)*  
 *// and would be processed after the request list is empty.*  
 *// No more requests could be added to the list here*  
 },  
});

**Result Storage**

The Apify SDK has several result storage types that are useful for specific tasks. The data is stored either on local disk to a directory defined by the APIFY\_LOCAL\_STORAGE\_DIR environment variable, or on the [Apify platform](https://docs.apify.com/sdk/js/docs/guides/apify-platform) under the user account identified by the API token defined by the APIFY\_TOKEN environment variable. If neither of these variables is defined, by default Apify SDK sets APIFY\_LOCAL\_STORAGE\_DIR to ./storage in the current working directory and prints a warning.

Typically, you will be developing the code on your local computer and thus set the APIFY\_LOCAL\_STORAGE\_DIR environment variable. Once the code is ready, you will deploy it to the Apify platform, where it will automatically set the APIFY\_TOKEN environment variable and thus use cloud storage. No code changes are needed.

**Key-value store**[**​**](https://docs.apify.com/sdk/js/docs/guides/result-storage#key-value-store)

The key-value store is used for saving and reading data records or files. Each data record is represented by a unique key and associated with a MIME content type. Key-value stores are ideal for saving screenshots of web pages, PDFs or to persist the state of crawlers.

Each actor run is associated with a **default key-value store**, which is created exclusively for the actor run. By convention, the actor run input and output is stored in the default key-value store under the INPUT and OUTPUT key, respectively. Typically the input and output is a JSON file, although it can be any other format.

In the Apify SDK, the key-value store is represented by the [KeyValueStore](https://docs.apify.com/sdk/js/reference/class/KeyValueStore) class. In order to simplify access to the default key-value store, the SDK also provides [Actor.getValue()](https://docs.apify.com/sdk/js/reference/class/Actor" \l "getValue) and [Actor.setValue()](https://docs.apify.com/sdk/js/reference/class/Actor" \l "setValue) functions.

In local configuration, the data is stored in the directory specified by the APIFY\_LOCAL\_STORAGE\_DIR environment variable as follows:

{APIFY\_LOCAL\_STORAGE\_DIR}/key\_value\_stores/{STORE\_ID}/{KEY}.{EXT}

Note that {STORE\_ID} is the name or ID of the key-value store. The default key value store has ID default, unless you override it by setting the APIFY\_DEFAULT\_KEY\_VALUE\_STORE\_ID environment variable. The {KEY} is the key of the record and {EXT} corresponds to the MIME content type of the data value.

The following code demonstrates basic operations of key-value stores:

*// Get actor input from the default key-value store*  
const input = await Actor.getInput();  
  
*// Write actor output to the default key-value store.*  
await Actor.setValue('OUTPUT', { myResult: 123 });  
  
*// Open a named key-value store*  
const store = await Actor.openKeyValueStore('some-name');  
  
*// Write record. JavaScript object is automatically converted to JSON,*  
*// strings and binary buffers are stored as they are*  
await store.setValue('some-key', { foo: 'bar' });  
  
*// Read record. Note that JSON is automatically parsed to a JavaScript object,*  
*// text data returned as a string and other data is returned as binary buffer*  
const value = await store.getValue('some-key');  
  
*// Delete record*  
await store.setValue('some-key', null);

To see a real-world example of how to get the input from the key-value store, see the [Screenshots](https://docs.apify.com/sdk/js/docs/examples/capture-screenshot) example.

**Dataset**[**​**](https://docs.apify.com/sdk/js/docs/guides/result-storage#dataset)

Datasets are used to store structured data where each object stored has the same attributes, such as online store products or real estate offers. You can imagine a dataset as a table, where each object is a row and its attributes are columns. Dataset is an append-only storage - you can only add new records to it but you cannot modify or remove existing records.

When the dataset is stored on the [Apify platform](https://docs.apify.com/sdk/js/docs/guides/apify-platform), you can export its data to the following formats: HTML, JSON, CSV, Excel, XML and RSS. The datasets are displayed on the actor run details page and in the [Storage](https://console.apify.com/storage) section in the Apify Console. The actual data is exported using the [Get dataset items](https://apify.com/docs/api/v2#/reference/datasets/item-collection/get-items) Apify API endpoint. This way you can easily share crawling results.

Each actor run is associated with a **default dataset**, which is created exclusively for the actor run. Typically, it is used to store crawling results specific for the actor run. Its usage is optional.

In the Apify SDK, the dataset is represented by the [Dataset](https://docs.apify.com/sdk/js/reference/class/Dataset) class. In order to simplify writes to the default dataset, the SDK also provides the [Actor.pushData()](https://docs.apify.com/sdk/js/reference/class/Actor" \l "pushData) function.

In local configuration, the data is stored in the directory specified by the APIFY\_LOCAL\_STORAGE\_DIR environment variable as follows:

{APIFY\_LOCAL\_STORAGE\_DIR}/datasets/{DATASET\_ID}/{INDEX}.json

Note that {DATASET\_ID} is the name or ID of the dataset. The default dataset has ID default, unless you override it by setting the APIFY\_DEFAULT\_DATASET\_ID environment variable. Each dataset item is stored as a separate JSON file, where {INDEX} is a zero-based index of the item in the dataset.

The following code demonstrates basic operations of the dataset:

*// Write a single row to the default dataset*  
await Actor.pushData({ col1: 123, col2: 'val2' });  
  
*// Open a named dataset*  
const dataset = await Actor.openDataset('some-name');  
  
*// Write a single row*  
await dataset.pushData({ foo: 'bar' });  
  
*// Write multiple rows*  
await dataset.pushData([{ foo: 'bar2', col2: 'val2' }, { col3: 123 }]);

**Proxy Management**

[IP address blocking](https://en.wikipedia.org/wiki/IP_address_blocking) is one of the oldest and most effective ways of preventing access to a website. It is therefore paramount for a good web scraping library to provide easy to use but powerful tools which can work around IP blocking. The most powerful weapon in your anti IP blocking arsenal is a [proxy server](https://en.wikipedia.org/wiki/Proxy_server).

With Apify SDK you can use your own proxy servers, proxy servers acquired from third-party providers, or you can rely on [Apify Proxy](https://apify.com/proxy" \t "_blank) for your scraping needs.

**Quick start**[**​**](https://docs.apify.com/sdk/js/docs/guides/proxy-management#quick-start)

If you already subscribed to Apify Proxy or have proxy URLs of your own, you can start using them immediately in only a few lines of code.

If you want to use Apify Proxy, make sure that your [scraper is logged in](https://docs.apify.com/sdk/js/docs/guides/apify-platform).

const proxyConfiguration = await Actor.createProxyConfiguration();  
const proxyUrl = proxyConfiguration.newUrl();

const proxyConfiguration = await Actor.createProxyConfiguration({  
 proxyUrls: [  
 'http://proxy-1.com',  
 'http://proxy-2.com',  
 ]  
});  
const proxyUrl = proxyConfiguration.newUrl();

**Proxy Configuration**[**​**](https://docs.apify.com/sdk/js/docs/guides/proxy-management#proxy-configuration)

All your proxy needs are managed by the [ProxyConfiguration](https://docs.apify.com/sdk/js/reference/class/ProxyConfiguration) class. You create an instance using the [Actor.createProxyConfiguration()](https://docs.apify.com/sdk/js/reference/class/Actor" \l "createProxyConfiguration) function. See the [ProxyConfigurationOptions](https://docs.apify.com/sdk/js/reference/interface/ProxyConfigurationOptions) for all the possible constructor options.

**Crawler integration**[**​**](https://docs.apify.com/sdk/js/docs/guides/proxy-management#crawler-integration)

ProxyConfiguration integrates seamlessly into [CheerioCrawler](https://crawlee.dev/api/cheerio-crawler/class/CheerioCrawler" \t "_blank) and [PuppeteerCrawler](https://crawlee.dev/api/puppeteer-crawler/class/PuppeteerCrawler" \t "_blank).

const proxyConfiguration = await Actor.createProxyConfiguration({ */\* your proxy opts \*/* });  
const crawler = new CheerioCrawler({  
 proxyConfiguration,  
 *// ...*  
});

const proxyConfiguration = await Actor.createProxyConfiguration({ */\* your proxy opts \*/* });  
const crawler = new PuppeteerCrawler({  
 proxyConfiguration,  
 *// ...*  
});

Your crawlers will now use the selected proxies for all connections.

**IP Rotation and session management**[**​**](https://docs.apify.com/sdk/js/docs/guides/proxy-management#ip-rotation-and-session-management)

[proxyConfiguration.newUrl()](https://docs.apify.com/sdk/js/reference/class/ProxyConfiguration#newUrl) allows you to pass a sessionId parameter. It will then be used to create a sessionId-proxyUrl pair, and subsequent newUrl() calls with the same sessionId will always return the same proxyUrl. This is extremely useful in scraping, because you want to create the impression of a real user. See the [session management guide](https://docs.apify.com/sdk/js/docs/guides/session-management) and [SessionPool](https://crawlee.dev/api/core/class/SessionPool" \t "_blank) class for more information on how keeping a real session helps you avoid blocking.

When no sessionId is provided, your proxy URLs are rotated round-robin, whereas Apify Proxy manages their rotation using black magic to get the best performance.

const proxyConfiguration = await Actor.createProxyConfiguration({ */\* opts \*/* });  
const sessionPool = await SessionPool.open({ */\* opts \*/* });  
const session = await sessionPool.getSession();  
const proxyUrl = proxyConfiguration.newUrl(session.id);

const proxyConfiguration = await Actor.createProxyConfiguration({ */\* opts \*/* });  
const crawler = new PuppeteerCrawler({  
 useSessionPool: true,  
 persistCookiesPerSession: true,  
 proxyConfiguration,  
 *// ...*  
});

**Apify Proxy vs. Your own proxies**[**​**](https://docs.apify.com/sdk/js/docs/guides/proxy-management#apify-proxy-vs-your-own-proxies)

The ProxyConfiguration class covers both Apify Proxy and custom proxy URLs so that you can easily switch between proxy providers, however, some features of the class are available only to Apify Proxy users, mainly because Apify Proxy is what one would call a super-proxy. It's not a single proxy server, but an API endpoint that allows connection through millions of different IP addresses. So the class essentially has two modes: Apify Proxy or Your proxy.

The difference is easy to remember. [ProxyConfigurationOptions.proxyUrls](https://docs.apify.com/sdk/js/reference/interface/ProxyConfigurationOptions#proxyUrls) and [ProxyConfigurationOptions.newUrlFunction](https://docs.apify.com/sdk/js/reference/interface/ProxyConfigurationOptions#newUrlFunction) enable use of your custom proxy URLs, whereas all the other options are there to configure Apify Proxy. Visit the [Apify Proxy docs](https://docs.apify.com/proxy" \t "_self) for more info on how these parameters work.

**Apify Proxy Configuration**[**​**](https://docs.apify.com/sdk/js/docs/guides/proxy-management#apify-proxy-configuration)

With Apify Proxy, you can select specific proxy groups to use, or countries to connect from. This allows you to get better proxy performance after some initial research.

const proxyConfiguration = await Actor.createProxyConfiguration({  
 groups: ['RESIDENTIAL'],  
 countryCode: 'US',  
});  
const proxyUrl = proxyConfiguration.newUrl();

Now your crawlers will use only Residential proxies from the US. Note that you must first get access to a proxy group before you are able to use it. You can find your available proxy groups in the [proxy dashboard](https://console.apify.com/proxy).

**Inspecting current proxy in Crawlers**[**​**](https://docs.apify.com/sdk/js/docs/guides/proxy-management#inspecting-current-proxy-in-crawlers)

CheerioCrawler and PuppeteerCrawler grant access to information about the currently used proxy in their handlePageFunction using a [proxyInfo](https://docs.apify.com/sdk/js/reference/interface/ProxyInfo) object. With the object, you can easily access the proxy URL. If you're using Apify Proxy, the other configuration parameters will also be available in the proxyInfo object.

**Session Management**

[SessionPool](https://crawlee.dev/api/core/class/SessionPool) is a class that allows you to handle the rotation of proxy IP addresses along with cookies and other custom settings in Apify SDK.

The main benefit of a Session pool is that you can filter out blocked or non-working proxies, so your actor does not retry requests over known blocked/non-working proxies. Another benefit of using SessionPool is that you can store information tied tightly to an IP address, such as cookies, auth tokens, and particular headers. Having your cookies and other identificators used only with a specific IP will reduce the chance of being blocked. Last but not least, another benefit is the even rotation of IP addresses - SessionPool picks the session randomly, which should prevent burning out a small pool of available IPs.

Now let's take a look at how to use a Session pool.

**Example usage in [PuppeteerCrawler](https://crawlee.dev/api/puppeteer-crawler/class/PuppeteerCrawler" \t "_blank)**

const proxyConfiguration = await Actor.createProxyConfiguration();  
  
const crawler = new PuppeteerCrawler({  
 requestQueue,  
 *// To use the proxy IP session rotation logic, you must turn the proxy usage on.*  
 proxyConfiguration,  
 *// Activates the Session pool.*  
 useSessionPool: true,  
 *// Overrides default Session pool configuration*  
 sessionPoolOptions: {  
 maxPoolSize: 100  
 },  
 *// Set to true if you want the crawler to save cookies per session,*  
 *// and set the cookies to page before navigation automatically.*  
 persistCookiesPerSession: true,  
 handlePageFunction: async ({ request, page, session }) => {  
 const title = await page.title();  
  
 if (title === "Blocked") {  
 session.retire()  
 } else if (title === "Not sure if blocked, might also be a connection error") {  
 session.markBad();  
 } else {  
 *// session.markGood() - this step is done automatically in puppeteer pool.*  
 }  
  
 }  
});

**Example usage in [CheerioCrawler](https://crawlee.dev/api/cheerio-crawler/class/CheerioCrawler" \t "_blank)**

const proxyConfiguration = await Actor.createProxyConfiguration();  
  
 const crawler = new CheerioCrawler({  
 requestQueue,  
 *// To use the proxy IP session rotation logic, you must turn the proxy usage on.*  
 proxyConfiguration,  
 *// Activates the Session pool.*  
 useSessionPool: true,  
 *// Overrides default Session pool configuration.*  
 sessionPoolOptions: {  
 maxPoolSize: 100  
 },  
 *// Set to true if you want the crawler to save cookies per session,*  
 *// and set the cookie header to request automatically...*  
 persistCookiesPerSession: true,  
 handlePageFunction: async ({request, $, session}) => {  
 const title = $("title");  
  
 if (title === "Blocked") {  
 session.retire()  
 } else if (title === "Not sure if blocked, might also be a connection error") {  
 session.markBad();  
 } else {  
 *// session.markGood() - this step is done automatically in BasicCrawler.*  
 }  
  
 }  
 });

**Example usage in [BasicCrawler](https://crawlee.dev/api/basic-crawler/class/BasicCrawler" \t "_blank)**

const { gotScraping } = require('got-scraping');  
 const proxyConfiguration = await Actor.createProxyConfiguration();  
  
 const crawler = new BasicCrawler({  
 requestQueue,  
 *// Allows access to proxyInfo object in handleRequestFunction*  
 proxyConfiguration,  
 useSessionPool: true,  
 sessionPoolOptions: {  
 maxPoolSize: 100  
 },  
 handleRequestFunction: async ({request, session, proxyInfo }) => {  
 *// To use the proxy IP session rotation logic, you must turn the proxy usage on.*  
 const proxyUrl = proxyInfo.url;  
 const requestOptions = {  
 url: request.url,  
 proxyUrl,  
 throwHttpErrors: false,  
 headers: {  
 *// If you want to use the cookieJar.*  
 *// This way you get the Cookie headers string from session.*  
 Cookie: session.getCookieString(),  
 }  
 };  
 let response;  
  
 try {  
 response = await gotScraping(requestOptions);  
 } catch (e) {  
 if (e === "SomeNetworkError") {  
 *// If a network error happens, such as timeout, socket hangup etc...*  
 *// There is usually a chance that it was just bad luck and the proxy works.*  
 *// No need to throw it away.*  
 session.markBad();  
 }  
 throw e;  
 }  
  
 *// Automatically retires the session based on response HTTP status code.*  
 session.retireOnBlockedStatusCodes(response.statusCode);  
  
 if (response.body.blocked) {  
 *// You are sure it is blocked.*  
 *// This will throw away the session.*  
 session.retire();  
  
 }  
  
 *// Everything is ok, you can get the data.*  
 *// No need to call session.markGood -> BasicCrawler calls it for you.*  
  
 *// If you want to use the CookieJar in session you need.*  
 session.setCookiesFromResponse(response);  
 }  
});

**Example solo usage**

Actor.main(async () => {  
 const sessionPoolOptions = {  
 maxPoolSize: 100  
 };  
 const sessionPool = await SessionPool.open(sessionPoolOptions);  
  
 *// Get session*  
 const session = sessionPool.getSession();  
  
 *// Increase the errorScore.*  
 session.markBad();  
  
 *// Throw away the session*  
 session.retire();  
  
 *// Lower the errorScore and marks the session good.*  
 session.markGood();  
});

These are the basics of configuring SessionPool. Please, bear in mind that a Session pool needs time to find working IPs and build up the pool, so you will probably see a lot of errors until it becomes stabilized.

**Pay-per-event Monetization**

Apify provides several [pricing models](https://docs.apify.com/platform/actors/publishing/monetize) for monetizing your Actors. The most recent and most flexible one is [pay-per-event](https://docs.apify.com/platform/actors/running/actors-in-store#pay-per-event), which lets you charge your users programmatically directly from your Actor. As the name suggests, you may charge the users each time a specific event occurs, for example a call to an external API or when you return a result.

To use the pay-per-event pricing model, you first need to [set it up](https://docs.apify.com/platform/actors/running/actors-in-store#pay-per-event) for your Actor in the Apify console. After that, you're free to start charging for events.

**Charging for events**[**​**](https://docs.apify.com/sdk/js/docs/guides/pay-per-event#charging-for-events)

After monetization is set in the Apify console, you can add [Actor.charge](https://docs.apify.com/sdk/js/reference/class/Actor" \l "charge) calls to your code and start monetizing!

import { Actor } from 'apify';  
  
await Actor.init();  
  
*// Charge for a single occurence of an event*  
await Actor.charge({ eventName: 'init' });  
  
*// Prepare some mock results*  
const result = [  
 { word: 'Lorem' },   
 { word: 'Ipsum' },   
 { word: 'Dolor' },   
 { word: 'Sit' },   
 { word: 'Amet' },  
];  
  
*// Shortcut for charging for each pushed dataset item*  
await Actor.pushData(result, 'result-item');  
  
*// Or you can charge for a given number of events manually*  
await Actor.charge({  
 eventName: 'result-item',  
 count: result.length,  
})  
  
await Actor.exit();

Then you just push your code to Apify and that's it! The SDK will even keep track of the max total charge setting for you, so you will not provide more value than what the user chose to pay for.

If you need finer control over charging, you can access call [Actor.getChargingManager()](https://docs.apify.com/sdk/js/reference/class/Actor" \l "getChargingManager) to access the [ChargingManager](https://docs.apify.com/sdk/js/reference/class/ChargingManager), which can provide more detailed information - for example how many events of each type can be charged before reaching the configured limit.

**Transitioning from a different pricing model**[**​**](https://docs.apify.com/sdk/js/docs/guides/pay-per-event#transitioning-from-a-different-pricing-model)

When you plan to start using the pay-per-event pricing model for an Actor that is already monetized with a different pricing model, your source code will need support both pricing models during the transition period enforced by the Apify platform. Arguably the most frequent case is the transition from the pay-per-result model which utilizes the ACTOR\_MAX\_PAID\_DATASET\_ITEMS environment variable to prevent returning unpaid dataset items. The following is an example how to handle such scenarios. The key part is the [ChargingManager.getPricingInfo](https://docs.apify.com/sdk/js/reference/class/ChargingManager" \l "getPricingInfo) method which returns information about the current pricing model.

import { Actor } from 'apify';  
  
await Actor.init();  
  
*// Check the dataset because there might already be items if the run migrated or was restarted*  
const defaultDataset = await Actor.openDataset();  
let chargedItems = (await defaultDataset.getInfo())!.itemCount;  
  
if (Actor.getChargingManager().getPricingInfo().isPayPerEvent) {  
 await Actor.pushData({ 'hello': 'world' }, 'dataset-item');  
} else {  
 if (chargedItems < Number(process.env['ACTOR\_MAX\_PAID\_DATASET\_ITEMS'])) {  
 await Actor.pushData({ 'hello': 'world' });  
 chargedItems += 1;  
 }  
}  
  
await Actor.exit();

**Local development**[**​**](https://docs.apify.com/sdk/js/docs/guides/pay-per-event#local-development)

It is encouraged to test your monetization code on your machine before releasing it to the public. To tell your Actor that it should work in pay-per-event mode, pass it the ACTOR\_TEST\_PAY\_PER\_EVENT environment variable:

ACTOR\_TEST\_PAY\_PER\_EVENT=true npm start

If you also wish to see a log of all the events charged throughout the run, you also need to pass the ACTOR\_USE\_CHARGING\_LOG\_DATASET environment variable. Your charging dataset will then be available under the charging\_log name (unless you change your storage settings, this dataset is stored in storage/datasets/charging\_log/). Please note that this log is not available when running the Actor in production on the Apify platform.

Because pricing configuration is stored by the Apify platform, all events will have a default price of $1

**Running in Docker**

Running headless browsers in Docker requires a lot of setup to do it right. But there's no need to worry about that, because we already created base images that you can freely use. We use them every day on the [Apify Platform](https://docs.apify.com/sdk/js/docs/guides/apify-platform).

All images can be found in their [GitHub repo](https://github.com/apify/apify-actor-docker) and in our [DockerHub](https://hub.docker.com/orgs/apify" \t "_blank).

**Overview**[**​**](https://docs.apify.com/sdk/js/docs/guides/docker-images#overview)

Browsers are pretty big, so we try to provide a wide variety of images to suit the specific needs. Here's a full list of our Docker images.

* [apify/actor-node](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node)
* [apify/actor-node-puppeteer-chrome](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node-puppeteer-chrome)
* [apify/actor-node-playwright](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node-playwright)
* [apify/actor-node-playwright-chrome](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node-playwright-chrome)
* [apify/actor-node-playwright-firefox](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node-playwright-firefox)
* [apify/actor-node-playwright-webkit](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node-playwright-webkit)

**Versioning**[**​**](https://docs.apify.com/sdk/js/docs/guides/docker-images#versioning)

Each image is tagged with up to 2 version tags, depending on the type of the image. One for Node.js version and second for pre-installed web automation library version. If you use the image name without a version tag, you'll always get the latest available version.

We recommend always using at least the Node.js version tag in production Dockerfiles. It will ensure that a future update of Node.js will not break our automations.

**Node.js versioning**[**​**](https://docs.apify.com/sdk/js/docs/guides/docker-images#nodejs-versioning)

Our images are built with multiple Node.js versions to ensure backwards compatibility. Currently, Node.js **versions 16 and 18 are supported** (legacy versions still exist, see DockerHub). To select the preferred version, use the appropriate number as the image tag.

*# Use Node.js 16*  
FROM apify/actor-node:16

**Automation library versioning**[**​**](https://docs.apify.com/sdk/js/docs/guides/docker-images#automation-library-versioning)

Images that include a pre-installed automation library, which means all images that include puppeteer or playwright in their name, are also tagged with the pre-installed version of the library. For example, apify/actor-node-puppeteer-chrome:16-13.7.0 comes with Node.js 16 and Puppeteer v13.7.0. If you try to install a different version of Puppeteer into this image, you may run into compatibility issues, because the Chromium version bundled with puppeteer will not match the version of Chromium that's pre-installed.

Similarly apify/actor-node-playwright-firefox:14-1.21.1 runs on Node.js 14 and is pre-installed with the Firefox version that comes with v1.21.1.

Installing apify/actor-node-puppeteer-chrome (without a tag) will install the latest available version of Node.js and puppeteer.

**Pre-release tags**[**​**](https://docs.apify.com/sdk/js/docs/guides/docker-images#pre-release-tags)

We also build pre-release versions of the images to test the changes we make. Those are typically denoted by a beta suffix, but it can vary depending on our needs. If you need to try a pre-release version, you can do it like this:

*# Without library version.*  
FROM apify/actor-node:16-beta

*# With library version.*  
FROM apify/actor-node-playwright-chrome:16-1.10.0-beta

**Best practices**[**​**](https://docs.apify.com/sdk/js/docs/guides/docker-images#best-practices)

* Node.js version tag should **always** be used.
* The automation library version tag should be used for **added security**.
* Asterisk \* should be used as the automation library version in our package.json files.

It makes sure the pre-installed version of Puppeteer or Playwright is not re-installed on build. This is important, because those libraries are only guaranteed to work with specific versions of browsers, and those browsers come pre-installed in the image.

FROM apify/actor-node-playwright-chrome:16

{  
 "dependencies": {  
 "crawlee": "^3.0.0",  
 "playwright": "\*"  
 }  
}

**Warning about image size**[**​**](https://docs.apify.com/sdk/js/docs/guides/docker-images#warning-about-image-size)

Browsers are huge. If you don't need them all in your image, it's better to use a smaller image with only the one browser you need.

You should also be careful when installing new dependencies. Nothing prevents you from installing Playwright into theactor-node-puppeteer-chrome image, but the resulting image will be about 3 times larger and extremely slow to download and build.

When you use only what you need, you'll be rewarded with reasonable build and start times.

**Apify Docker Images**[**​**](https://docs.apify.com/sdk/js/docs/guides/docker-images#apify-docker-images)

**actor-node**[**​**](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node)

This is the smallest image we have based on Alpine Linux. It does not include any browsers, and it's therefore best used with [CheerioCrawler](https://crawlee.dev/api/cheerio-crawler/class/CheerioCrawler" \t "_blank). It benefits from lightning fast builds and container startups.

[PuppeteerCrawler](https://crawlee.dev/api/puppeteer-crawler/class/PuppeteerCrawler), [PlaywrightCrawler](https://crawlee.dev/api/playwright-crawler/class/PlaywrightCrawler" \t "_blank) and other browser based features will **NOT** work with this image.

FROM apify/actor-node:16

**actor-node-puppeteer-chrome**[**​**](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node-puppeteer-chrome)

This image includes Puppeteer (Chromium) and the Chrome browser. It can be used with [CheerioCrawler](https://crawlee.dev/api/cheerio-crawler/class/CheerioCrawler" \t "_blank) and [PuppeteerCrawler](https://crawlee.dev/api/puppeteer-crawler/class/PuppeteerCrawler" \t "_blank), but **NOT** with [PlaywrightCrawler](https://crawlee.dev/api/playwright-crawler/class/PlaywrightCrawler" \t "_blank).

The image supports XVFB by default, so you can run both headless and headful browsers with it.

FROM apify/actor-node-puppeteer-chrome:16

**actor-node-playwright**[**​**](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node-playwright)

A very large and slow image that can run all Playwright browsers: Chromium, Chrome, Firefox, WebKit. Everything is installed. If you need to develop or test with multiple browsers, this is the image to choose, but in most cases, it's better to use the specialized images below.

FROM apify/actor-node-playwright:16

**actor-node-playwright-chrome**[**​**](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node-playwright-chrome)

Similar to [actor-node-puppeteer-chrome](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node-puppeteer-chrome), but for Playwright. You can run [CheerioCrawler](https://crawlee.dev/api/cheerio-crawler/class/CheerioCrawler" \t "_blank) and [PlaywrightCrawler](https://crawlee.dev/api/playwright-crawler/class/PlaywrightCrawler" \t "_blank), but **NOT** [PuppeteerCrawler](https://crawlee.dev/api/puppeteer-crawler/class/PuppeteerCrawler" \t "_blank).

It uses the [PLAYWRIGHT\_SKIP\_BROWSER\_DOWNLOAD](https://playwright.dev/docs/api/environment-variables/) environment variable to block installation of more browsers into the image to keep it small. If you want more browsers, either use the [actor-node-playwright](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node-playwright) image override this env var.

The image supports XVFB by default, so we can run both headless and headful browsers with it.

FROM apify/actor-node-playwright-chrome:16

**actor-node-playwright-firefox[​](https://docs.apify.com/sdk/js/docs/guides/docker-images" \l "actor-node-playwright-firefox" \o "Direct link to heading)**

Same idea as [actor-node-playwright-chrome](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node-playwright-chrome), but with Firefox pre-installed.

FROM apify/actor-node-playwright-firefox:16

**actor-node-playwright-webkit[​](https://docs.apify.com/sdk/js/docs/guides/docker-images" \l "actor-node-playwright-webkit" \o "Direct link to heading)**

Same idea as [actor-node-playwright-chrome](https://docs.apify.com/sdk/js/docs/guides/docker-images#actor-node-playwright-chrome), but with WebKit pre-installed.

FROM apify/actor-node-playwright-webkit:16

**Example Dockerfile[​](https://docs.apify.com/sdk/js/docs/guides/docker-images" \l "example-dockerfile" \o "Direct link to heading)**

To use the above images, it's necessary to have a [Dockerfile](https://docs.docker.com/engine/reference/builder/" \t "_blank). You can either use this example, or bootstrap your projects with the [Crawlee CLI](https://crawlee.dev/docs/introduction/setting-up" \t "_blank) which automatically adds the correct Dockerfile into our project folder.

* **Node+JavaScript**
* **Node+TypeScript**
* **Browser+JavaScript**
* **Browser+TypeScript**

*# Specify the base Docker image. You can read more about*  
*# the available images at https://crawlee.dev/docs/guides/docker-images*  
*# You can also use any other image from Docker Hub.*  
FROM apify/actor-node:16  
  
*# Copy just package.json and package-lock.json*  
*# to speed up the build using Docker layer cache.*  
COPY package\*.json ./  
  
*# Install NPM packages, skip optional and development dependencies to*  
*# keep the image small. Avoid logging too much and print the dependency*  
*# tree for debugging*  
RUN npm --quiet set progress=false \  
 && npm install --omit=dev --omit=optional \  
 && echo "Installed NPM packages:" \  
 && (npm list --omit=dev --all || true) \  
 && echo "Node.js version:" \  
 && node --version \  
 && echo "NPM version:" \  
 && npm --version  
  
*# Next, copy the remaining files and directories with the source code.*  
*# Since we do this after NPM install, quick build will be really fast*  
*# for most source file changes.*  
COPY . ./  
  
  
*# Run the image.*  
CMD npm start --silent

**ccept user input**

This example accepts and logs user input:

import { Actor } from 'apify';  
  
await Actor.init();  
  
const input = await Actor.getInput();  
console.log(input);  
  
await Actor.exit();

To provide the actor with input, create a INPUT.json file inside the "default" key-value store:

{PROJECT\_FOLDER}/storage/key\_value\_stores/default/INPUT.json

Anything in this file will be available to the actor when it runs.

**Add data to dataset**

This example saves data to the default dataset. If the dataset doesn't exist, it will be created. You can save data to custom datasets by using [Actor.openDataset()](https://docs.apify.com/sdk/js/reference/class/Dataset" \l "open)

[**Run on**](https://console.apify.com/actors/kk67IcZkKSSBTslXI?runConfig=eyJ1IjoiRWdQdHczb2VqNlRhRHQ1cW4iLCJ2IjoxfQ..fhpAfqCjjEMd7THx-jtJurjuRe7si1RztaBrOcDRcQ8&asrc=run_on_apify)

import { Actor } from 'apify';  
import { CheerioCrawler } from 'crawlee';  
  
await Actor.init();  
  
*// Create a dataset where we will store the results.*  
const crawler = new CheerioCrawler({  
 *// Function called for each URL*  
 async requestHandler({ request, body }) {  
 *// Save data to default dataset*  
 await Actor.pushData({  
 url: request.url,  
 html: body,  
 });  
 },  
});  
  
*// Run the crawler*  
await crawler.run([  
 { url: 'http://www.example.com/page-1' },  
 { url: 'http://www.example.com/page-2' },  
 { url: 'http://www.example.com/page-3' },  
]);  
  
await Actor.exit();

Each item in this dataset will be saved to its own file in the following directory:

{PROJECT\_FOLDER}/storage/datasets/default/

**Basic crawler**

This is the most bare-bones example of the Apify SDK, which demonstrates some of its building blocks such as the [BasicCrawler](https://crawlee.dev/api/basic-crawler/class/BasicCrawler" \t "_blank). You probably don't need to go this deep though, and it would be better to start with one of the full-featured crawlers like [CheerioCrawler](https://crawlee.dev/api/cheerio-crawler/class/CheerioCrawler" \t "_blank) or [PlaywrightCrawler](https://crawlee.dev/api/playwright-crawler/class/PlaywrightCrawler" \t "_blank).

The script simply downloads several web pages with plain HTTP requests using the [got-scraping](https://github.com/apify/got-scraping) npm package and stores their raw HTML and URL in the default dataset. In local configuration, the data will be stored as JSON files in ./storage/datasets/default.

[**Run on**](https://console.apify.com/actors/kk67IcZkKSSBTslXI?runConfig=eyJ1IjoiRWdQdHczb2VqNlRhRHQ1cW4iLCJ2IjoxfQ..-TdH8qt-fjSHPGoP8mJHr2LqYkhq6aWUhY9IdesMFrM&asrc=run_on_apify)

import { Actor } from 'apify';  
import { BasicCrawler } from 'crawlee';  
import { gotScraping } from 'got-scraping';  
  
await Actor.init();  
  
*// Create a dataset where we will store the results.*  
*// Create a BasicCrawler - the simplest crawler that enables*  
*// users to implement the crawling logic themselves.*  
const crawler = new BasicCrawler({  
 *// This function will be called for each URL to crawl.*  
 async requestHandler({ request }) {  
 const { url } = request;  
 console.log(`Processing ${url}...`);  
  
 *// Fetch the page HTML via Apify utils gotScraping*  
 const { body } = await gotScraping({ url });  
  
 *// Store the HTML and URL to the default dataset.*  
 await Actor.pushData({  
 url: request.url,  
 html: body,  
 });  
 },  
});  
  
*// The initial list of URLs to crawl. Here we use just a few hard-coded URLs.*  
await crawler.run([  
 { url: 'http://www.google.com/' },  
 { url: 'http://www.example.com/' },  
 { url: 'http://www.bing.com/' },  
 { url: 'http://www.wikipedia.com/' },  
]);  
  
console.log('Crawler finished.');  
  
await Actor.exit();

**Call actor**

This example demonstrates how to start an Apify actor using [Actor.call()](https://docs.apify.com/sdk/js/reference/class/Actor" \l "call) and how to call the Apify API using [Actor.newClient()](https://docs.apify.com/sdk/js/reference/class/Actor" \l "newClient). The script gets a random weird word and its explanation from [randomword.com](https://randomword.com/) and sends it to your email using the [apify/send-mail](https://apify.com/apify/send-mail" \t "_blank) actor.

To make the example work, you'll need an [Apify account](https://console.apify.com/" \t "_blank). Go to the [Settings - Integrations](https://console.apify.com/account?tab=integrations) page to obtain your API token and set it to the [APIFY\_TOKEN](https://docs.apify.com/sdk/js/docs/guides/environment-variables#APIFY_TOKEN) environment variable, or run the script using the Apify CLI. If you deploy this actor to the Apify Cloud, you can do things like set up a scheduler to run your actor early in the morning.

To see what other actors are available, visit the [Apify Store](https://apify.com/store" \t "_blank).

To run this example on Apify Platform, use the apify/actor-node-puppeteer-chrome image for your Dockerfile.

import { Actor } from 'apify';  
import { launchPuppeteer } from 'crawlee';  
  
await Actor.init();  
  
*// Launch the web browser.*  
const browser = await launchPuppeteer();  
  
console.log('Obtaining own email address...');  
const apifyClient = Actor.newClient();  
const { email } = await apifyClient.user().get();  
  
*// Load randomword.com and get a random word*  
console.log('Fetching a random word.');  
const page = await browser.newPage();  
await page.goto('https://randomword.com/');  
const randomWord = await page.$eval('#shared\_section', (el) => el.outerHTML);  
  
*// Send random word to your email. For that, you can use an actor we already*  
*// have available on the platform under the name: apify/send-mail.*  
*// The second parameter to the Actor.call() invocation is the actor's*  
*// desired input. You can find the required input parameters by checking*  
*// the actor's documentation page: https://apify.com/apify/send-mail*  
console.log(`Sending email to ${user.email}...`);  
await Actor.call('apify/send-mail', {  
 to: email,  
 subject: 'Random Word',  
 html: `<h1>Random Word</h1>${randomWord}`,  
});  
console.log('Email sent. Good luck!');  
  
*// Close Browser*  
await browser.close();  
  
await Actor.exit();

**Cheerio crawler**

This example demonstrates how to use [CheerioCrawler](https://crawlee.dev/api/cheerio-crawler/class/CheerioCrawler" \t "_blank) to crawl a list of URLs from an external file, load each URL using a plain HTTP request, parse the HTML using the [Cheerio library](https://www.npmjs.com/package/cheerio) and extract some data from it: the page title and all h1 tags.

[**Run on**](https://console.apify.com/actors/kk67IcZkKSSBTslXI?runConfig=eyJ1IjoiRWdQdHczb2VqNlRhRHQ1cW4iLCJ2IjoxfQ..dOZILM56nUwSSVMoLgQB0brbbjQm2W2FDao35eLD72s&asrc=run_on_apify)

import { CheerioCrawler, log, LogLevel } from 'crawlee';  
  
*// Crawlers come with various utilities, e.g. for logging.*  
*// Here we use debug level of logging to improve the debugging experience.*  
*// This functionality is optional!*  
log.setLevel(LogLevel.DEBUG);  
  
*// Create an instance of the CheerioCrawler class - a crawler*  
*// that automatically loads the URLs and parses their HTML using the cheerio library.*  
const crawler = new CheerioCrawler({  
 *// The crawler downloads and processes the web pages in parallel, with a concurrency*  
 *// automatically managed based on the available system memory and CPU (see AutoscaledPool class).*  
 *// Here we define some hard limits for the concurrency.*  
 minConcurrency: 10,  
 maxConcurrency: 50,  
  
 *// On error, retry each page at most once.*  
 maxRequestRetries: 1,  
  
 *// Increase the timeout for processing of each page.*  
 requestHandlerTimeoutSecs: 30,  
  
 *// Limit to 10 requests per one crawl*  
 maxRequestsPerCrawl: 10,  
  
 *// This function will be called for each URL to crawl.*  
 *// It accepts a single parameter, which is an object with options as:*  
 *// https://sdk.apify.com/docs/typedefs/cheerio-crawler-options#handlepagefunction*  
 *// We use for demonstration only 2 of them:*  
 *// - request: an instance of the Request class with information such as URL and HTTP method*  
 *// - $: the cheerio object containing parsed HTML*  
 async requestHandler({ request, $ }) {  
 log.debug(`Processing ${request.url}...`);  
  
 *// Extract data from the page using cheerio.*  
 const title = $('title').text();  
 const h1texts = [];  
 $('h1').each((index, el) => {  
 h1texts.push({  
 text: $(el).text(),  
 });  
 });  
  
 *// Store the results to the dataset. In local configuration,*  
 *// the data will be stored as JSON files in ./storage/datasets/default*  
 await Actor.pushData({  
 url: request.url,  
 title,  
 h1texts,  
 });  
 },  
  
 *// This function is called if the page processing failed more than maxRequestRetries+1 times.*  
 failedRequestHandler({ request }) {  
 log.debug(`Request ${request.url} failed twice.`);  
 },  
});  
  
*// Run the crawler and wait for it to finish.*  
await crawler.run();  
  
log.debug('Crawler finished.');

**Crawl a single URL**

This example uses the [got-scraping](https://github.com/apify/got-scraping) npm package to grab the HTML of a web page.

[**Run on**](https://console.apify.com/actors/kk67IcZkKSSBTslXI?runConfig=eyJ1IjoiRWdQdHczb2VqNlRhRHQ1cW4iLCJ2IjoxfQ.eyJpbnB1dCI6IntcImNvZGVcIjpcImltcG9ydCB7IGdvdFNjcmFwaW5nIH0gZnJvbSAnZ290LXNjcmFwaW5nJztcXG5cXG4vLyBHZXQgdGhlIEhUTUwgb2YgYSB3ZWIgcGFnZVxcbmNvbnN0IHsgYm9keSB9ID0gYXdhaXQgZ290U2NyYXBpbmcoeyB1cmw6ICdodHRwczovL3d3dy5leGFtcGxlLmNvbScgfSk7XFxuY29uc29sZS5sb2coYm9keSk7XFxuXCJ9Iiwib3B0aW9ucyI6eyJidWlsZCI6ImxhdGVzdCIsImNvbnRlbnRUeXBlIjoiYXBwbGljYXRpb24vanNvbjsgY2hhcnNldD11dGYtOCIsIm1lbW9yeSI6MTAyNCwidGltZW91dCI6MTgwfX0.0S1i1yD10_82mLCH3VWFtCZTU4-BDrDU1UGY208IqgE&asrc=run_on_apify)

import { gotScraping } from 'got-scraping';  
  
*// Get the HTML of a web page*  
const { body } = await gotScraping({ url: 'https://www.example.com' });  
console.log(body);

If you don't want to hard-code the URL into the script, refer to the [Accept User Input](https://docs.apify.com/sdk/js/docs/examples/accept-user-input) example.

**Puppeteer crawler**

This example demonstrates how to use [PuppeteerCrawler](https://crawlee.dev/api/puppeteer-crawler/class/PuppeteerCrawler" \t "_blank) in combination with [RequestQueue](https://docs.apify.com/sdk/js/reference/class/RequestQueue) to recursively scrape the [Hacker News website](https://news.ycombinator.com/) using headless Chrome / Puppeteer.

The crawler starts with a single URL, finds links to next pages, enqueues them and continues until no more desired links are available. The results are stored to the default dataset. In local configuration, the results are stored as JSON files in ./storage/datasets/default

**tip**

To run this example on the Apify Platform, select the apify/actor-node-puppeteer-chrome image for your Dockerfile.

[**Run on**](https://console.apify.com/actors/7tWSD8hrYzuc9Lte7?runConfig=eyJ1IjoiRWdQdHczb2VqNlRhRHQ1cW4iLCJ2IjoxfQ.-.88cqtP3DJA1811DUd2fOqdjsLFRPvz91Pi_WHe8Yt5U&asrc=run_on_apify)

import { Actor } from 'apify';  
import { PuppeteerCrawler } from 'crawlee';  
  
await Actor.init();  
  
*// Create an instance of the PuppeteerCrawler class - a crawler*  
*// that automatically loads the URLs in headless Chrome / Puppeteer.*  
const crawler = new PuppeteerCrawler({  
 *// Here you can set options that are passed to the launchPuppeteer() function.*  
 launchContext: {  
 launchOptions: {  
 headless: true,  
 *// Other Puppeteer options*  
 },  
 },  
  
 *// Stop crawling after several pages*  
 maxRequestsPerCrawl: 50,  
  
 *// This function will be called for each URL to crawl.*  
 *// Here you can write the Puppeteer scripts you are familiar with,*  
 *// with the exception that browsers and pages are automatically managed by the Apify SDK.*  
 *// The function accepts a single parameter, which is an object with the following fields:*  
 *// - request: an instance of the Request class with information such as URL and HTTP method*  
 *// - page: Puppeteer's Page object (see https://pptr.dev/#show=api-class-page)*  
 async requestHandler({ request, page, enqueueLinks }) {  
 console.log(`Processing ${request.url}...`);  
  
 *// A function to be evaluated by Puppeteer within the browser context.*  
 const data = await page.$$eval('.athing', ($posts) => {  
 const scrapedData = [];  
  
 *// We're getting the title, rank and URL of each post on Hacker News.*  
 $posts.forEach(($post) => {  
 scrapedData.push({  
 title: $post.querySelector('.title a').innerText,  
 rank: $post.querySelector('.rank').innerText,  
 href: $post.querySelector('.title a').href,  
 });  
 });  
  
 return scrapedData;  
 });  
  
 *// Store the results to the default dataset.*  
 await Actor.pushData(data);  
  
 *// Find a link to the next page and enqueue it if it exists.*  
 const infos = await enqueueLinks({  
 selector: '.morelink',  
 });  
  
 if (infos.length === 0) console.log(`${request.url} is the last page!`);  
 },  
  
 *// This function is called if the page processing failed more than maxRequestRetries+1 times.*  
 failedRequestHandler({ request }) {  
 console.log(`Request ${request.url} failed too many times.`);  
 },  
});  
  
*// Run the crawler and wait for it to finish.*  
await crawler.run(['https://news.ycombinator.com/']);  
  
console.log('Crawler finished.');  
  
await Actor.exit();

**Puppeteer with proxy**

This example demonstrates how to load pages in headless Chrome / Puppeteer over [Apify Proxy](https://docs.apify.com/proxy" \t "_self).

To make it work, you'll need an Apify account with access to the proxy. Visit the [Apify platform introduction](https://docs.apify.com/sdk/js/docs/guides/apify-platform) to find how to log into your account from the SDK.

**tip**

To run this example on the Apify Platform, select the apify/actor-node-puppeteer-chrome image for your Dockerfile.

[**Run on**](https://console.apify.com/actors/7tWSD8hrYzuc9Lte7?runConfig=eyJ1IjoiRWdQdHczb2VqNlRhRHQ1cW4iLCJ2IjoxfQ..Z2NfopKj1DbaGy58OZ3N2Og8hM7AvkFTeEbBFCwOtGk&asrc=run_on_apify)

import { Actor } from 'apify';  
import { PuppeteerCrawler } from 'crawlee';  
  
await Actor.init();  
  
*// Proxy connection is automatically established in the Crawler*  
const proxyConfiguration = await Actor.createProxyConfiguration();  
  
const crawler = new PuppeteerCrawler({  
 proxyConfiguration,  
 async requestHandler({ page }) {  
 const status = await page.$eval('td.status', (el) => el.textContent);  
 console.log(`Proxy Status: ${status}`);  
 },  
});  
  
console.log('Running Puppeteer script...');  
  
await crawler.run(['http://proxy.apify.com']);  
  
console.log('Puppeteer closed.');  
  
await Actor.exit();

**Playwright crawler**

This example demonstrates how to use [PlaywrightCrawler](https://crawlee.dev/api/playwright-crawler/class/PlaywrightCrawler" \t "_blank) in combination with [RequestQueue](https://docs.apify.com/sdk/js/reference/class/RequestQueue) to recursively scrape the [Hacker News website](https://news.ycombinator.com/) using headless Chrome / Playwright.

The crawler starts with a single URL, finds links to next pages, enqueues them and continues until no more desired links are available. The results are stored to the default dataset. In local configuration, the results are stored as JSON files in ./storage/datasets/default

**tip**

To run this example on the Apify Platform, select the apify/actor-node-playwright-chrome image for your Dockerfile.

[**Run on**](https://console.apify.com/actors/6i5QsHBMtm3hKph70?runConfig=eyJ1IjoiRWdQdHczb2VqNlRhRHQ1cW4iLCJ2IjoxfQ.-.kt3UXVDBPuO5BGjh8ZGuYNiKR_W8ilbU9XBQeJvrZFs&asrc=run_on_apify)

import { Actor } from 'apify';  
import { PlaywrightCrawler } from 'crawlee';  
  
await Actor.init();  
  
*// Create an instance of the PlaywrightCrawler class - a crawler*  
*// that automatically loads the URLs in headless Chrome / Playwright.*  
const crawler = new PlaywrightCrawler({  
 launchContext: {  
 *// Here you can set options that are passed to the playwright .launch() function.*  
 launchOptions: {  
 headless: true,  
 },  
 },  
  
 *// Stop crawling after several pages*  
 maxRequestsPerCrawl: 50,  
  
 *// This function will be called for each URL to crawl.*  
 *// Here you can write the Playwright scripts you are familiar with,*  
 *// with the exception that browsers and pages are automatically managed by the Apify SDK.*  
 *// The function accepts a single parameter, which is an object with a lot of properties,*  
 *// the most important being:*  
 *// - request: an instance of the Request class with information such as URL and HTTP method*  
 *// - page: Playwright's Page object (see https://playwright.dev/docs/api/class-page)*  
 async requestHandler({ request, page, enqueueLinks }) {  
 console.log(`Processing ${request.url}...`);  
  
 *// A function to be evaluated by Playwright within the browser context.*  
 const data = await page.$$eval('.athing', ($posts) => {  
 const scrapedData = [];  
  
 *// We're getting the title, rank and URL of each post on Hacker News.*  
 $posts.forEach(($post) => {  
 scrapedData.push({  
 title: $post.querySelector('.title a').innerText,  
 rank: $post.querySelector('.rank').innerText,  
 href: $post.querySelector('.title a').href,  
 });  
 });  
  
 return scrapedData;  
 });  
  
 *// Store the results to the default dataset.*  
 await Actor.pushData(data);  
  
 *// Find a link to the next page and enqueue it if it exists.*  
 const infos = await enqueueLinks({  
 selector: '.morelink',  
 });  
  
 if (infos.processedRequests.length === 0) console.log(`${request.url} is the last page!`);  
 },  
  
 *// This function is called if the page processing failed more than maxRequestRetries+1 times.*  
 failedRequestHandler({ request }) {  
 console.log(`Request ${request.url} failed too many times.`);  
 },  
});  
  
*// Run the crawler and wait for it to finish.*  
await crawler.run(['https://news.ycombinator.com/']);  
  
console.log('Crawler finished.');  
  
await Actor.exit();