NPM

NPM (Node Package Manager) is a tool that makes it easy to integrate third-party libraries and development tools into your projects. NPM repositories contain some of the most popular libraries, frameworks, and tools, such as jQuery, Angular, React, Express, ESLint, Sass, Browserify, Bootstrap, and Ionic, among many others.

In addition, NPM provides us with a complete script-based task automation system.

Installing NPM

To use NPM, we first need to install **Node.js** , since NPM is part of Node. Node allows JavaScript applications to run on the server using the v8 engine (the same as Chrome), so we can create both client and server applications with a single language: JavaScript.

Here are the links to download Node:

Windows and Mac Installer

Linux Installation (Repositories)

> It's best to install the latest **LTS** (Long Term Support) release, as it's more stable than a non-LTS release and offers better support from libraries and frameworks. For more information, see the Node.js Release Guide .

You can verify that the installation is correct by running the command **npm --version** in the console , which should print the current NPM version. Or **node --version** for the Node version.

We can consult the npm help using the commands:

- **npm -h** → Displays quick help and a list of typical npm commands
- **npm command -h** → Displays quick help with specific information about the npm command (npm install -h)
- **npm help command** → Opens a page in the browser or the Linux man page, showing help on a given command
- **npm help-search wordlist** → Returns a help list of commands that contain the searched word(s) (separated by spaces)

Creating an NPM project

An NPM project is a directory containing a file called **package.json** (in addition to all other files), which specifies the project's author, version, external dependencies, automated tasks, and so on. In this section, we'll look at how to install, update, or remove dependencies from our project (or across our system).

To create the package.json file in our project, we'll simply move to the main project directory (it doesn't have to be empty) and run **npm init** . It will ask us a few questions about our project and create the package.json file based on those questions. We can leave many of the values as default or empty by pressing Enter.

```
Press ^C at any time to quit.
package name: (ejemplo)
version: (1.0.0)
description: Proyecto de ejemplo
entry point: (index.js)
test command:
git repository:
keywords:
author: Minombre
license: (ISC)
About to write to /home/ /Documentos/ejemplo/package.json:

{
  "name": "ejemplo",
  "version": "1.0.0",
  "description": "Proyecto de ejemplo",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Minombre",
  "license": "ISC"
}
```

Sections of this section

Manage dependencies

Manage dependencies

Install dependencies

The command to install a package in our project is **npm install package-name** . If we run this command for the first time, we'll see that it creates a directory called **node_modules** within the project directory. This directory will contain all the packages installed via npm (including their dependencies). Let's try it out with **Day.js** .

npm i dayjs

This will install the latest stable version of the Day.js library (for date processing) by default in the node_modules directory. You can now use it in your code:

 # index.js

```
import "./node_modules/dayjs/dayjs.min.js";

console.log(dayjs('2025-08-25').format('DD/MM/YYYY')); // '25/08/2025'
```

Importing the library into code is still somewhat cumbersome, since we have to find the file containing the library, and we're basically using it as if we were manually including it in the HTML. Later, we'll see how to optimize this with tools like **Vite** .

Types of dependencies

By default, when we install a dependency, it is installed as **a runtime dependency** . A runtime dependency is one that the program needs to run in the browser. We typically refer to a JavaScript library when referring to this. There are also **development dependencies** , which are tools that only the programmer will use, such as compilers (from TypeScript to JavaScript), packagers like Webpack, Vite, code obfuscators, testing libraries, etc.

To install a package as a development dependency we use the **-D** or **--save-dev** option :

npm i -D uglify-js

Now we can see how in the package.json file, we have registered both the execution dependencies and the development dependencies:

 # package.json

```
{
  "name": "ejemplo",
  "version": "1.0.0",
  "description": "Proyecto de ejemplo",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Arturo",
  "license": "ISC",
  "dependencies": {
    "dayjs": "^1.11.9"
  },
  "devDependencies": {
    "uglify-js": "^3.17.4"
  }
}
```

Reinstall dependencies in a project

The node_modules directory can be deleted at any time. In fact, in code repositories (Github, for example), this directory is never uploaded. To reinstall dependencies at any time, both for production and development, which appear in package.json, simply run **npm install** .

Podemos instalar solo las dependencias de producción o solo las dependencias de desarrollo con **npm install --only=prod**, o **npm install --only=dev**.

Instalar paquetes globales

Ciertas herramientas o frameworks, como Angular o Nest, requieren la instalación de herramientas en el sistema para poder crear proyectos. Estos son paquetes globales que se instalan con la opción **-g** (ejemplo: **npm i -g @angular/cli**) Para saber donde se instalan estos paquetes ejecuta **npm root -g**.

Los paquetes globales contienen ejecutables (archivos JavaScript ejecutados bajo Node) que se registran para todo el sistema. Es decir, se pueden ejecutar desde la consola independientemente del directorio donde nos encontremos.

El comando npx

Para ejecutar la herramienta instalada localmente en el proyecto donde nos encontramos, bien sea porque queremos ejecutar la versión local en lugar de la global, o porque no la tenemos instalada de forma global, basta con poner el comando **npx** delante. Si no está instalada, la descargará y ejecutará. Ejemplo: **npx uglify-js -mc archivo.js**.

Listar paquetes instalados

Para poder ver los paquetes instalados en el directorio npm_modules escribiremos **npm list**. Si queremos ver en formato árbol, los paquetes que hay instalados y las dependencias que tiene cada uno de ellos, tenemos la opción **--depth nivel**. Donde nivel es la profundidad de dependencias a la que queremos descender.

```
$ npm list
├── bootstrap@5.3.2
├── dayjs@1.11.10
├── eslint@8.53.0
└── vite@4.5.0
```

```
$ npm list --depth 1
├─┬ bootstrap@5.3.2
│ └── @popperjs/core@2.11.8
├── dayjs@1.11.10
├─┬ eslint@8.53.0
│ ├── @eslint-community/eslint-utils@4.4.0
│ ├── @eslint-community/regexpp@4.10.0
│ ├── @eslint/eslintrc@2.1.3
│ └── @eslint/js@8.53.0
```

Otra opciones bastante útil es incluir **--include dev** (lista sólo los paquetes en desarrollo), **--include prod** (solo los paquetes en producción), **--long** (muestra una descripción de cada paquete), o **--global** (lista paquetes globales).

Gestión de versiones

Por defecto, cuando instalamos un paquete, se instalará la última versión estable. Pero a veces, nuestro proyecto necesita incluir una versión anterior, por ejemplo, para soportar navegadores antiguos.

Las versiones de los paquetes normalmente tienen tres números **X.Y.Z**. La mayoría de paquetes siguen las reglas SemVer (Semantic Versioning) para aplicar la numeración. Las reglas son las siguientes:

- Cuando la **Z** es incrementada, significa que un bug o problema ha sido solucionado, pero no se añade ninguna funcionalidad nueva.
- Cuando la **Y** es incrementada, significa que nuevas funcionalidades han sido añadidas, pero que esto no afecta al código de las versiones previas (siempre que se mantenga la X). Por ejemplo, una aplicación hecha con la versión 2.2 debería seguir funcionando con la versión 2.5 (pero al revés no tiene por qué).
- El número **X** se incrementará sólo si han habido suficientes cambios de forma que no se garantiza que las aplicaciones que utilicen versiones anteriores funcionen compatibles con la nueva. Por tanto debemos tener cuidado cuando hagamos una migración y vigilar los cambios que se han producido en la librería o framework.

Si queremos instalar una versión específica de una librería en lugar de la última versión, especificaremos la versión después del nombre del paquete: **paquete@x.y.z**. Ejemplos de posibilidades que tenemos:

- **npm i libreria@"2.5.3"** → Instala la versión 2.5.3
- **"<2.0.0"** → Instala la versión justo anterior a la 2.0.0
- **"*" ó "x"** → Instala la última versión de un paquete (cuidado con los cambios que pueda tener esa versión y cómo nos puede afectar…).
- **"3"** ó **"3.x"**, ó **"3.x.x"** → Esto instala la última versión de un paquete siempre que sea 3.x.x (no se actualizará a la versión 4.x.x).
- **"^3.3.5"** → Instalará la última versión 3 (3.x.x) de un paquete, pero como mínimo deberá ser la versión 3.3.5 (el carácter ^ significa que sólo el primer número, 3, debe respetarse). Este es el comportamiento por defecto de NPM
- **"3.3"** ó **"3.3.x"** → Instalará la última versión 3.3.x de un paquete (nunca se actualizará a 3.4.x o posterior).
- **"~3.3.5"** → Instalará la última versión de 3.3 (no actualizará a la 3.4 o superior) pero como mínimo deberá ser la versión 3.3.5 (el carácter ~ significa que los 2 primeros números, 3.3, deben respetarse). Equivale a "3.3.x".
- **latest** → Instala la última versión estable.
- **next** → Instala la que será la próxima versión estable, aunque todavía no lo es (beta, RC)

Actualizar paquetes

Para actualizar todas las dependencias de un proyecto ejecutaremos **npm update** . Mirará en el archivo package.json para ver a qué versiones se permite actualizar (por ejemplo, si la versión instalada es ^3.1.0 podrá actualizar a la 3.2.0 pero no a la 4.0.0). Para actualizar un paquete en concreto utilizaremos **npm update paquete**.

Si queremos instalar una versión más actual que la permitida en package.json, podemos volver a reinstalar el paquete con **npm install** (ejemplo: npm install paquete@latest), que instalará la versión especificada y actualizará el archivo package.json.

Eliminar paquetes

To remove a package from our project, we'll type **npm uninstall package** . Instead of uninstall, we can use **remove** , **rm** , **un** , **r** , or **unlink** to do the same. To uninstall a global package, we'll use the **-g** option .

Automate tasks

In our **package.json** file , in addition to managing dependencies, we can create some useful scripts for our project (running a web server, testing our application, minifying and packaging code, etc.). These scripts are commands and they all have a name that identifies them. We must put these tasks inside the **"script"** section . The syntax of the script will be: **"script-name": "Command to execute"** , where the command can be a system command (it will not be cross-platform), or any executable tool that we have installed with NPM in the project (without needing to use npx). To run a script we will use: **npm run script-name** .

 # package.json

```
  "scripts": {
      "hola": "echo \"Hola mundo\""
  }
```

```
$ npm run hola

> ejemplo-npm@1.0.0 hola
> echo "Hola mundo"

Hola mundo
```

Startup script

There are some commonly used script names, such as **start** or **test** , that can be run without typing **run** . The **start** script is typically used to run a web server, so we can test our application (required, for example, when using modules: <script type="module">).

For example, we can install **lite-server** , a lightweight development server to test our application, when we run npm start . This server also detects file changes and automatically reloads the application. We install it with **npm i -D lite-server** .

 # package.json

```
  "scripts": {
    "start": "lite-server"
  },
```

```
$ npm start

> ejemplo-npm@1.0.0 start
> lite-server

Did not detect a `bs-config.json` or `bs-config.js` override file.
** browser-sync config **
{
  injectChanges: false,
  files: [ './**/*.{html,htm,css,js}' ],
  watchOptions: { ignored: 'node_modules' },
  server: {
    baseDir: './',
    middleware: [ [Function (anonymous)], [Function (anonymous)] ]
  }
}
[Browsersync] Access URLs:
 ------------------------------------
       Local: http://localhost:3000
    External: http://10.184.31.44:3000
 ------------------------------------
          UI: http://localhost:3001
 UI External: http://localhost:3001
 ------------------------------------
[Browsersync] Serving files from: ./
[Browsersync] Watching files...
```

Test script

In a real project, it is normal to have a battery of tests for the application made with a testing framework such as Jest, Jasmine, Karma, ... It is a good idea to launch these tests with the **test** script .

To simplify the example, we'll use a linter, in this case **ESLint** , which enforces a style and best practices in the application code. It's very useful for maintaining well-structured and consistent code in projects, especially involving multiple programmers.

There are packages that can be installed and configured using the init command. To install ESLint, we'll run **npm init @eslint/config@latest** .

```
Need to install the following packages:
@eslint/create-config@1.3.1
Ok to proceed? (y)


> ejemplo-handlebars@0.0.0 npx
> create-config

@eslint/create-config: v1.3.1

✔How would you like to use ESLint? · problems
✔What type of modules does your project use? · esm
✔Which framework does your project use? · none
✔Does your project use TypeScript? · javascript
✔Where does your code run? · browser
The config that you've selected requires the following dependencies:

eslint, globals, @eslint/js
✔Would you like to install them now? · No / Yes
✔Which package manager do you want to use? · npm
```

A configuration file called **eslint.config.js** will be created with the chosen options . We can add, remove, or change some of the default rules, for example, instead of displaying an error when a variable is not used, display a warning (changing "error" to "warn"), or set the indentation to two spaces:

 # eslint.config.mjs

```
import js from "@eslint/js";
import globals from "globals";
import { defineConfig } from "eslint/config";

export default defineConfig([
  {
    files: ["**/*.{js,mjs,cjs}"],
    plugins: { js },
    extends: ["js/recommended"],
    languageOptions: { globals: globals.browser },
    rules: {
      indent: ["error", 2],
      "no-unused-vars": "warn"
    },
  },
]);
```

Eslint Rules List: https://eslint.org/docs/latest/rules/

Code formatting rules (indentation, semicolons, quotes, line length, etc.) have been deprecated in ESLint and will be discontinued in the future. This is because that role is being left to code formatting tools like Prettier . For more information, see Deprecation of formatting rules .

Now we can put the command in our package.json, so that when we run **npm test** , it runs eslint on the files we want:

 # package.json

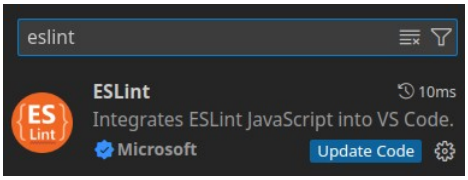
```
"scripts": {
  "start": "lite-server",
  "test": "eslint index.js"
},
```

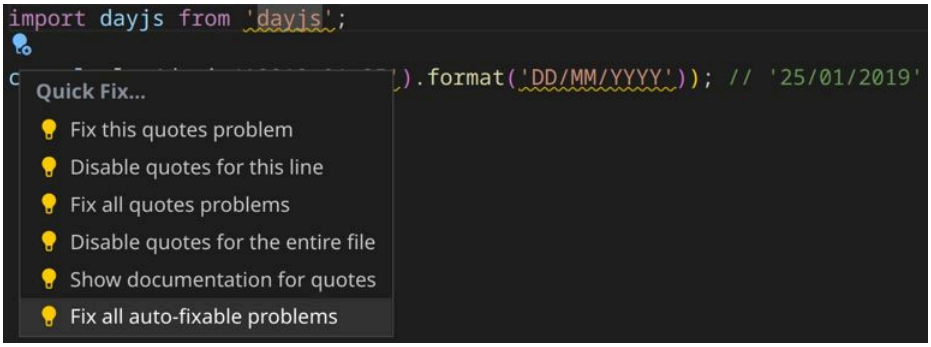
If we want it to act on all the files in a folder for example, we can use wildcard characters like the asterisk ' * ': **eslint src/*.js** .

```
$ npm test

> ejemplo-npm@1.0.0 test
> eslint src/*


/home/                                                    /ejemplo-npm/sr
   6:1   error    Expected indentation of 4 spaces but found 3  indent
   9:6   warning  Strings must use doublequote                  quotes
  10:21  error    Missing semicolon                             semi

× 3 problems (2 errors, 1 warning)
  2 errors and 1 warning potentially fixable with the `--fix` option.
```

ESLint VSCode Extension

To help us with ESLint errors, we have an extension for Visual Studio Code, which will automatically detect errors and warnings in the editor, giving us the option of auto-correction whenever possible.





Ignore files

If we don't want the ESLint plugin to check some files, because they are not code files, but configuration files (or packaged in the dist folder) and we don't want to force any style there, we can indicate the files and folders we don't want it to check with the **ignore** property :

 # eslint.config.mjs

```
import globals from "globals";
import pluginJs from "@eslint/js";


export default [
  {languageOptions: { globals: globals.browser }},
  pluginJs.configs.recommended,
  {
    rules: { /* Reglas propias */},
    ignores: ["eslint.config.js", "compiled.js", "dist/**"]
  }
];
```

Pre and post-script tasks

Some tasks may require multiple steps, such as minifying our JavaScript and CSS files before running the application (startup script). To run more than one task or command, we can link them with && (and). These commands will execute in order until one fails.

 # package.json

```
"scripts": {
    "start": "lite-server",
    "start:test": "eslint index.js && npm start",
}
```

However, it's cleaner to use NPM's predefined **hooks (prefixes) pre** and **post** . Simply create a script with one of these prefixes, and it will run before (pre) or after (post) the main script we want to run. In this example, we'll create a pretest script.

package.json

```
"scripts": {
    "prestart": "echo 'We are going to make some tests'",
    "start": "lite-server",
    "test": "eslint index.js",
}
```

Now, if we run **npm start , prestart** and **start** will be executed in that order. If one script fails, the next one won't run, so if the application doesn't pass the tests, the development server won't launch.

Vite

Vite is a very useful tool for developing a web application, as it has a series of comprehensive features that make it unique.

- Integrate a web server for application development, just run **vite**
- To prepare the application for production, Vite uses esbuild to minify the code (including CSS and TypeScript, for example) and rollup to package everything or just a few files. To package the code, we'll use the **vite build** command .
- **It has built-in TypeScript** support , so we can work directly with TypeScript (**.ts**) files without any issues. It also has built-in support for files with the **.jsx** or **.tsx** extension that use frameworks like React or Vue.
- It supports importing CSS directly from JavaScript code, as well as preprocessors like **sass** , **stylus** , or **less** . However, you'll need to first install the preprocessor you want to use in your project (for example, **npm i -D sass**).
- Major client-side frameworks have already migrated to Vite as their development server. **Angular** , **React** , **Vue** , and **Quick all** include it by default.

More features of Vite

In many projects, **Vite** is replacing tools like **Webpack** . The main difference is speed, as during development, Vite works directly with modules in the browser, while Webpack packages all the code and includes it in the HTML without using modules, as it is designed for older browsers that don't understand JavaScript modules.

The work of processing TypeScript and minifying code is also much faster thanks to the use of **esbuild** .

Install Vite

To create a new project with Vite, we'll run the command **npm create vite@latest** . We'll then open the directory created with Visual Studio Code and run **npm install** to install the dependencies.

```
$ npm create vite@latest

> npx
> create-vite

✔ Project name: … vite-project
✔ Select a framework: › Vanilla
✔ Select a variant: › JavaScript

Scaffolding project in /home/      /Documentos/vite-project...

Done. Now run:

  cd vite-project
  npm install
  npm run dev
```

Alternatively, if we already have a project created and want to use Vite, we can install it directly with **npm i vite@latest** .

Managing the project with Vite

Vite is designed to integrate with NPM. In the **package.json** file , we'll have a series of scripts to launch the application in development mode (**start**) or package it for production (**build**). Starting with the project created with Vite, to simplify and standardize it compared to other similar tools, the "dev" script has been replaced with "start."

 # package.json

```
{
  "name": "vite-project",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "start": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "devDependencies": {
    "vite": "^5.4.1"
  }
}
```

To run the development server, we'll use **npm start** . By default, it will run at **http://localhost:5173/** and load the **index.html** file . The code (JavaScript, TypeScript, CSS, SASS, etc.) it will process is the code we include in the HTML (style, script) and the code we import from the code files

with import.

If we run **npm run build** , it will create a directory called **dist/** ready to deploy on an external web server with the packaged code.

In the NPM example where we installed **DayJS** to manage dates, we can now change the import because Vite recognizes that "dayjs" is a library and automatically searches for it within **node_modules** . This way, we'll no longer have the problem with ESLint not recognizing the dayjs variable or function.

 # index.js

```
import dayjs from "dayjs";

console.log(dayjs("2019-01-25").format("DD/MM/YYYY")); // '25/01/2019'
```

Static File Management

Static files (images, videos, etc.) can be referenced from HTML or JavaScript code as we normally would, with one exception.

When we run **npm run build** and Vite packages the code into **dist/** , it includes the code files, CSS, images, etc. that are referenced from the HTML in that directory along with index.html. When referencing a static file from JavaScript code, we must use import so that Vite knows we need it and includes it in the dist/ folder in production.

 # index.js

```
import logo from './logo.png'; // Url a la imagen

document.getElementById('logo').append(logo);
```

It also detects files included in the HTML, such as CSS, etc. The exception is files in the public directory, which will always be copied as is to the dist folder.

The public directory

By default, all files referenced in the HTML or imported from code files will be included in the **dist/assets** directory. Vite will automatically change the path from **./file** to **/assets/file as long as we follow the rules explained in the previous section.**

Vite will also rename files, adding some kind of suffix (it also handles updating this in the HTML and JavaScript). However, we can put static files directly into the **public** directory , and they'll be copied as is to the **dist/** directory . This is useful when:

- Let's have files that are never referenced in the source code (or not via import)
- They must keep the same file name (without adding a hash as a suffix to the name)
- Generally when they are files that do not need to be processed and minified by Vite, such as images or videos.

From the HTML and JavaScript code, these files should always be referenced as if they were in the server root. We won't use **public/logo.png ,** but rather **/logo.png** .

Multi-page application

Vite's default configuration is designed for **SPA** (Single Page Applications), where we only have one HTML file (index.html). This is normal for applications developed with client-side frameworks like Angular, React, or Vue, for example. This page will never reload, and the JavaScript code will be responsible for modifying its content.

If we want to have an application with multiple HTML or entry points, we must modify Vite's configuration. To do this, we create a file in the project root called **vite.config.js** . Within that file, we must specify the HTML documents for our application, which will be included in the **dist/** directory along with the files they reference when generating the production code.

vite.config.js

```
import { resolve } from 'path'
import { defineConfig } from 'vite'

export default defineConfig({
  build: {
    rollupOptions: {
      input: {
        index: resolve(__dirname, 'index.html'),
        page2: resolve(__dirname, 'page2.html'),
      },
    },
  },
})
```

Web Storage API

Cookies have traditionally been used to store data persistently on the client. However, they have several limitations, such as a maximum size of 4KB per domain, and the fact that they are not sent to the server when it is running on a different domain (or port) than the client application. Therefore, they are not a viable option when the server is a Rest API, for example.

The Web Storage API has been around for several years, allowing up to 5MB of storage per domain. In this case, the client application is responsible for managing this storage and sending the data it needs to the server.

This storage system is very easy to use. There's a global object (within window) called **localStorage** . From this object, we can store data persistently. This data is related to the domain where the client web application is running.

The functionality of this API is very simple. Some examples:

 # index.js

```
// Creamos la variable usuario con el valor "Pepe"
localStorage.setItem("usuario", "Pepe");
// También se puede crear así:
localStorage.usuario = "Pepe";

//Obtener el valor guardado
console.log(localStorage.getItem("usuario")); // Imprime "Pepe"
// O así
console.log(localStorage.usuario); // Imprime "Pepe"

// Borrar una variable/entrada
localStorage.removeItem("usuario");

// Borramos todos los datos almacenados
localStorage.clear();
```

As you've seen, the data is stored in key-value pairs . Both values are saved as strings, so if we want to save an object or array (JSON), or an image, we must serialize them first.

Unless we deliberately clear the browser data, the saved data will still be there the next time we access our page. If we want data to be cleared when we close the page (only for the current session), we can use **sessionStorage** (used identically) instead.

Using LocalStorage to save credentials

Web applications that have the client side separated from the server (running on different domains or web servers) cannot use cookies to have the client store information that is sent to the server with each request, and therefore session-based authentication cannot be used.

This is why server-side web services APIs typically use token authentication. A web services API is said to be stateless because it doesn't store any information related to connecting clients. That is, each request to the server is completely independent of the previous one, and the client is responsible for providing the server with the information necessary to complete that request. For example, the credentials of the logged-in user.

Authentication credentials are typically sent to the server in an HTTP header. The standard approach is to use the **Authorization** header , although this depends on how the server is configured. Three methods are commonly used to send credentials:

- **Basic Authentication** : The user's credentials (username and password) are sent to the server with each request in this format: 'username:password' encoded in base64 and prefixed with 'Basic'. Example → *Authorization: "Basic YWxhZGRpbjpvcGVuc2VzYW1l"* .
- **Digest authentication** : This is based on sending multiple pieces of data with the Digest prefix. This includes an MD5 hash of the user, the password, and a random number returned by the server in the last request.
- **JWT token authentication** : This is the most common. When the user submits their username and password (via a login form), the server generates a token, for example in JWT (JSON Web Token) format, and sends it as a response to the client. From now on, these will be the client's credentials, which must be sent to the server in every request from then on. These are usually preceded by the Bearer prefix: *Authorization: "Bearer TOKEN_JWT"* .

> The enhanced version of the JWT token includes the use of an **additional refresh token** . This is also stored on the server and can be invalidated at any time (when the user logs out, for example). In these cases, we'll set a very short expiration date for the JWT token, and it can be automatically renewed using the refresh token, which will have a significantly longer expiration time.

To manage the token from JavaScript, we'll store it (when we receive the login response) in a local storage system like localStorage. From then on, the simplest thing to do is check if we have a stored token every time we send an HTTP request to the server, and if so, send it.

 # http.class.js

```
export class Http {
  async ajax(method, url, body = null) {
    const json = body && ! body instanceof FormData;
    const headers = body && json ? { "Content-Type": "application/json" } : {};
    const token = localStorage.getItem('token');
```

```
    if(token) headers.Authorization = 'Bearer ' + token;

    const resp = await fetch(url, { method, headers, json ? JSON.stringify(body) : body });

    // Procesar respuesta
  }

  // Resto de métodos
}
```

Geolocation API

The Geolocation API allows you to use JavaScript to query a user's geolocation coordinates (much more accurate when using a GPS system → e.g., mobile phone). This is a very important feature today, as it allows you to position yourself on the map, find nearby users, measure distances, etc. It's ideal to combine it with a mapping API such as Google Maps, Mapbox, ArcGIS Maps, or similar.

Location consists of two coordinate components: **latitude** , which is the distance north (positive) or south (negative). And **longitude** , which is the distance east (positive) or west (negative) from the Greenwich Meridian.

We use **navigator.geolocation.getCurrentPosition** to geolocate ourselves. This method is asynchronous (runs in the background), so it doesn't return anything immediately. We'll pass it a function that will be executed when the browser finally finds the location. This function will receive a parameter containing the location information.

 # index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ejemplo de cómo obtener la geolocalización</title>
    <link rel="stylesheet" href="style.css">
    <script src="index.js" type="module"></script>
  </head>
  <body>
    <div>
        <p id="coordenadas">Obteniendo información de la localización...</p>
    </div>
  </body>
</html>
```

 # index.js

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ejemplo de cómo obtener la geolocalización</title>
    <link rel="stylesheet" href="style.css">
    <script src="index.js" type="module"></script>
  </head>
  <body>
    <div>
        <p id="coordenadas">Obteniendo información de la localización...</p>
    </div>
  </body>
</html>
```

Don't forget to give your browser permission to get your location.

To handle errors, we can pass a second function to the geolocation method. This function will be called instead of the first if an error occurs, and will receive as a parameter an object with a property called code containing the type of error.

Additionally, it's a good idea to use **promises** to manage this asynchronous application so that we can manage the geolocation result outside of the function we pass to it.

 # index.js

```
function getLocation() {
    return new Promise((resolve, reject) => {
        navigator.geolocation.getCurrentPosition(
            pos => {
                resolve(pos.coords);
            },
            error => {
                switch (error.code) {
                case error.PERMISSION_DENIED: // User didn't allow the web page to retrieve location
                    reject("User denied the request for Geolocation.");
                    break;
                case error.POSITION_UNAVAILABLE: // Couldn't get the location
                    reject("Location information is unavailable.");
                    break;
                case error.TIMEOUT: // The maximum amount of time to get location information has passed
                    reject("The request to get user location timed out.");
                    break;
```

```
                default:
                    reject("An unknown error occurred.");
                    break;
                }
            }
        );
    });
}

async function showMap() {
      let coords = await getLocation();
      // Mostrar el mapa con el marcador, etc.
}
```

These are the location properties we can get (some of these will only be available when using a GPS, such as on a mobile phone):

- **coords.latitude** → Latitude, decimal number.
- **coords.longitude** → Longitude, a decimal number.
- **coords.accuracy** → The accuracy, in meters.
- **coords.altitude** → Altitude, in meters above sea level (if available).
- **coords.altitudeAccuracy** → The altitude accuracy (if available).
- **coords.heading** → The heading in degrees (if available).
- **coords.speed** → The speed in meters/second (if available)
- **timestamp** → The response time, UNIX timestamp (if available).

Geolocation options

There's a third parameter we can pass to the **getCurrentPosition** method . This parameter is a JSON object containing one or more of these properties:

- **enableHighAccuracy** → A Boolean indicating whether the device should use as much precision as possible to obtain the most accurate position (default is false). This option consumes more battery and time.
- **timeout** → Time in milliseconds to wait for the position, or an error will be thrown. Default is 0 (waits indefinitely).
- **maximumAge** → Time in milliseconds that the browser caches the last position. If a new position is requested before the time expires, the browser returns the cached data directly.

 # index.js

```
navigator.geolocation.getCurrentPosition(
    positionFunction,
    errorFunction,
    {
        enableHighAccuracy: true,
        timeout: 6000, // 6 segundos para timeout
        maximumAge: 30000 // 30 segundos de caché
    }
);
```

Map management

OpenLayers (mapa)

The vast majority of advanced mapping APIs require payment information to register and use the API, even if it's free. To avoid surprises, we'll use an API that doesn't require this, such as OpenLayers . Ultimately, the concepts are similar across all of them, so learning to use others once you're familiar with this one should be straightforward.

OpenLayers is capable of working with various map servers. In our case, we'll be working with OpenStreetMap , which is open and free, but we could also work with Mapbox or Bing Maps, for example.

To install the library we will run **npm i ol** in our project.

Show a map

To display a map with OpenLayers, we must take into account at least the following:

- Call the **useGeographic()** function to enable geographic coordinates (latitude and longitude) to be used to position the map and its elements.
- Create a view (**View**) that will then be associated with the map. The view defines the position of the map being displayed (coordinates) and the zoom, among other things. It would be like the camera in a scene.
- Create a map (**Map**) where we will indicate the source of the images. In our case, OpenStreetMap (**OSM**), as well as the view (**view**) and the id or reference to the HTML element where the map will be displayed (**target**).
- **A VectorLayer** object associated with the map that will later be used to draw things on the map such as points, etc.

> This library receives the coordinates in reverse order to others like Google Maps, so the coordinate array will contain **[longitude, latitude]** .

 # index.html

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite App</title>
    <script type="module" src="/index.js"></script>
    <link rel="stylesheet" href="node_modules/ol/ol.css">
    <style>
      .map {
        width: 100%;
        height: 400px;
      }
    </style>
  </head>
  <body>
    <div id="map" class="map"></div>
  </body>
</html>
```

 # index.js

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite App</title>
    <script type="module" src="/index.js"></script>
    <link rel="stylesheet" href="node_modules/ol/ol.css">
    <style>
      .map {
        width: 100%;
        height: 400px;
      }
    </style>
  </head>
  <body>
    <div id="map" class="map"></div>
  </body>
</html>
```

Adding bookmarks

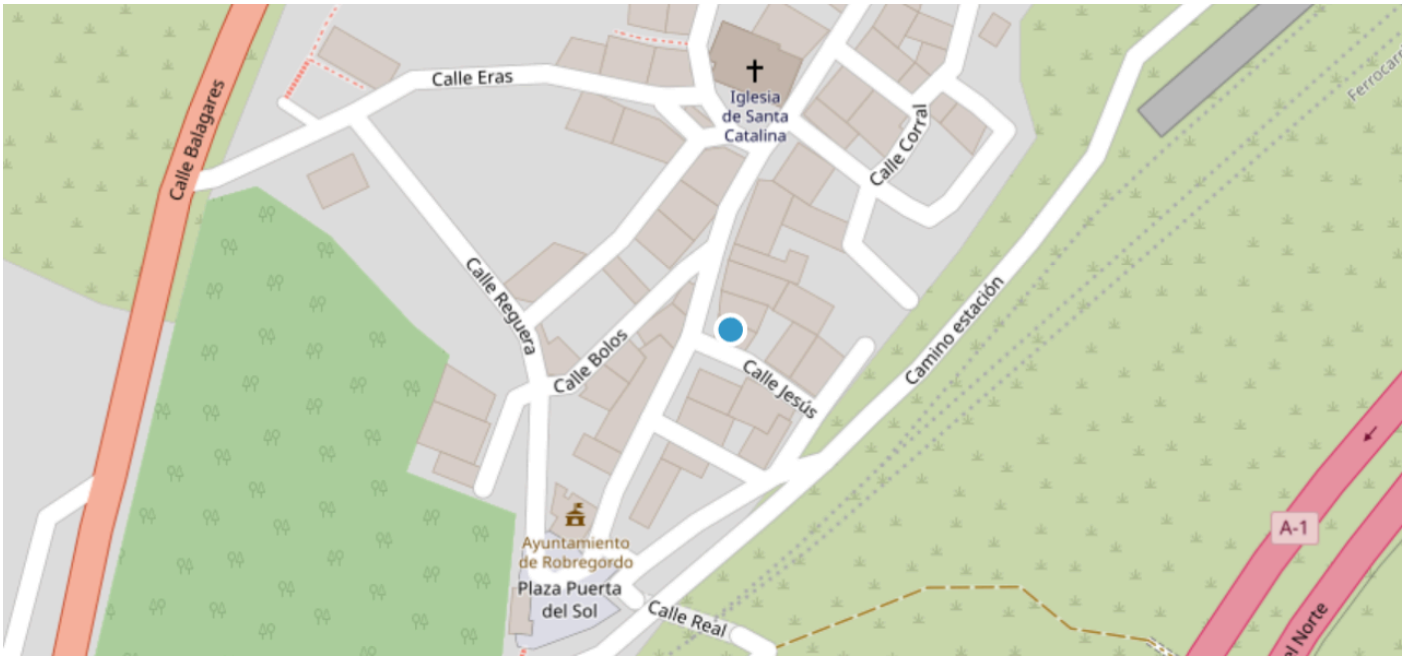
The extra elements added to the map are called **Features and are added to the VectorLayer** layer . Although images can be used, we'll simply create a dot as a marker for simplicity.

En este ejemplo vamos a crear un mapa inicialmente centrado en la geolocalización del usuario con un punto en el centro que marque su posición. Utilizaremos una función para generar los marcadores (features) y así la podremos reutilizar si queremos añadir varios marcadores al mapa.

 # index.js

```
import { View, Map, Feature } from "ol";
import { OSM, Vector as VectorSource } from "ol/source";
import { Tile as TileLayer, Vector as VectorLayer } from "ol/layer";
import { useGeographic } from "ol/proj";
import { Circle as CircleStyle, Fill, Stroke, Style } from "ol/style.js";
import { Point } from "ol/geom";

useGeographic();

const place = [-3.5935, 41.1061];

const view = new View({
  center: place,
  zoom: 18,
});

const map = new Map({
  layers: [
    new TileLayer({
      source: new OSM(),
    }),
  ],
  target: "map",
  view: view,
});

const vectorLayer = new VectorLayer({
  map: map,
  source: new VectorSource({
    features: [],
  }),
});

function createMarker(coordinates, color="#3399CC", fill="#fff") {
  const positionFeature = new Feature({ geometry: new Point(coordinates)});
  positionFeature.setStyle(
    new Style({
```

```
      image: new CircleStyle({
        radius: 9,
        fill: new Fill({
          color: color,
        }),
        stroke: new Stroke({
          color: fill,
          width: 3,
        }),
      }),
    })
  );

  return positionFeature;
}

vectorLayer.getSource().addFeature(createMarker(place));
```



Eventos del mapa

Openlayers soporta múltiples tipos de interacciones, sobre todo de cara a trabajar con los elementos que podemos dibujar en un mapa en las diferentes capas que podemos añadir. En este caso vamos a ver un ejemplo de como gestionar un click en el mapa, obtener las coordenadas y centrar el mapa y el marcador en ese punto.

 # index.js

```
// Creación del mapa
// Función createMarker(...)

const marker = createMarker(place);
vectorLayer.getSource().addFeature(marker);

map.on("click", (e) => {
  marker.setGeometry(new Point(e.coordinate));
  view.setCenter(e.coordinate);
});
```

Geoapify (Geocodificación)

Una funcionalidad que se suele buscar en una aplicación que trabaje con mapas es la búsqueda de lugares o geocodifación (**Geocoding**). Para ello necesitamos utilizar otra librería, que aunque sea de pago, ofrece una cuota gratuita sin necesidad de introducir información de pago, y con un número de consultas diarias limitadas. Esto es debido a que es un proceso que gasta muchos recursos, principalmente de almacenamiento y memoria, por lo que las opciones gratuitas son muy limitadas.

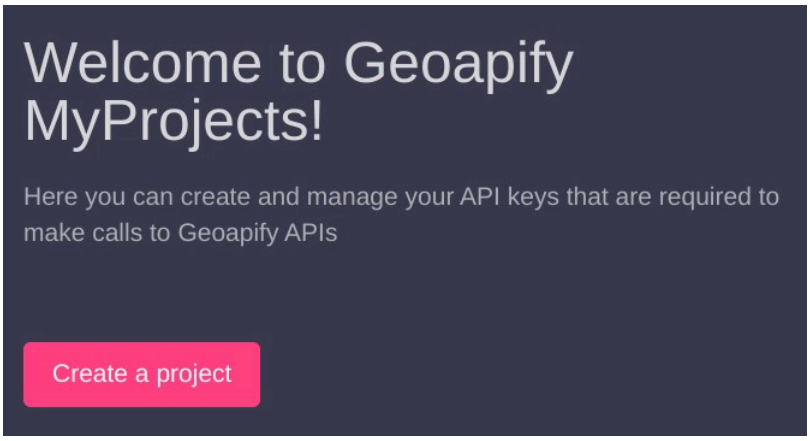
Utilizaremos el servicio de geocodificación de **Geoapify**. Concretamente la geocodificación directa. Estos son los 2 tipos de gecodificación que hay:

- **Geocodificación directa**: A partir de una dirección o el nombre de un lugar te devuelve una o más coincidencias con información del lugar y sus **coordenadas**.
- **Geocodificación inversa**: A partir de unas coordenadas, devuelve información de la dirección, negocio, parque, etc que se encuentra en dicho lugar.

Para instalar la librería de geodificación de Geoapify ejecutaremos **npm install @geoapify/geocoder-autocomplete**.

Obtener una clave de API

Antes de usar el servicio, debemos generar una clave de API. Para ello nos registrarmos en la página de Geoapify, y una vez iniciada sesión, creamos un proyecto.



Una vez creado el proyecto, se nos creará una clave de API que necesitaremos para que funcione la gecodificación en nuestro proyecto.

Mostrar sugerencias

Finalmente, incorporamos la librería a nuestro código. Debemos hacer lo siguiente:

- Importar el CSS (desde nuestro archivo CSS con @import, desde JavaScript con import o desde html). En este caso lo haremos de la primera forma.
- Añadir un ** ** donde la librería situará un input para que el usuario escriba y nos muestre las sugerencias. Este div tendrá una id y la clase **autocomplete-container** (debe tener position **relative** o **absolute**)
- In the code, we'll create a **GeocoderAutocomplete object, which we'll link to the created earlier. The " select** " event that tells us the user has selected a location applies to the GeocoderAutocomplete object.
- Set the **debounceDelay** option in the options when creating the object. This is the number of milliseconds that will pass from the moment the user stops typing until the API query begins. This way, we'll avoid many unnecessary requests.

 # index.html

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite App</title>
    <script type="module" src="/index.js"></script>
    <link rel="stylesheet" href="node_modules/ol/ol.css">
    <link rel="stylesheet" href="style.css">
    <style>
      .map {
        width: 100%;
        height: 400px;
      }
    </style>
  </head>
  <body>
    <div id="autocomplete" class="autocomplete-container"></div>
```

```
    <div id="map" class="map"></div>
  </body>
</html>
```

style.css

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite App</title>
    <script type="module" src="/index.js"></script>
    <link rel="stylesheet" href="node_modules/ol/ol.css">
    <link rel="stylesheet" href="style.css">
    <style>
      .map {
        width: 100%;
        height: 400px;
      }
    </style>
  </head>
  <body>
    <div id="autocomplete" class="autocomplete-container"></div>
    <div id="map" class="map"></div>
  </body>
</html>
```

index.js

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite App</title>
    <script type="module" src="/index.js"></script>
    <link rel="stylesheet" href="node_modules/ol/ol.css">
    <link rel="stylesheet" href="style.css">
    <style>
      .map {
        width: 100%;
        height: 400px;
      }
    </style>
  </head>
  <body>
    <div id="autocomplete" class="autocomplete-container"></div>
    <div id="map" class="map"></div>
  </body>
</html>
```

Champs-Élysé

- Champs Élysées 75008 París, Francia
- Théâtre des Champs-Elysées Avenue Montaigne, 75008 París, Francia
- **Champs-Élysé**es - Clemenceau Avenue des Champs-Élysées, 75008 París, Francia
- Champs-Elysées París, Isla de Francia, Francia
- Jardins des **Champs-Élysé**es 75008 París, Francia

Language detection

The Language Detector API allows you to detect with a certain degree of certainty the language in which a text is written. This can be useful when using other tools such as the Translator API.

Check language support

The static method availability checks whether the model has support for a language. To do this, we pass it an array containing the list of languages we want to work with (in this case, detecting them in a text).

The call receives an array of the languages to be checked and returns a promise with a string indicating the support. The possible results are:

- **available** : The browser supports the language
- **downloadable** : The browser needs to download the AI model for that language
- **downloading** : The browser is downloading language support
- **unavailable** : The browser does not support language detection

 # index.js

```
async function langSupport(language) {
  const availability = await LanguageDetector.availability({
    expectedInputLanguages: [language],
  });
  console.log(availability);
}

langSupport('es'); // available
```

Detecting the language

To detect the language of a text, we must first create an instance of the **LanguageDetector** class using the static **create** method . We pass it an array of languages it should be able to detect, and if it supports them, it will return a promise with the LanguageDetector instance, or an error otherwise.

With the received object, we can use the **detect** method with a text string. It will return an array of the detected languages based on the confidence level (from highest to lowest). The correct language will generally be in the first position (if it was detected correctly).

The LanguageDetector object can only be used once to detect the language of a text. If we want to detect the language of more texts, we must create new objects.

 # index.js

```
async function detectLanguage(text) {
   const detector = await LanguageDetector.create({
    expectedInputLanguages: ["en", "es", "fr"],
  });
  const results = await detector?.detect(text);
  console.log(text);
  console.log(results);
  console.log(`Language detected: ${results[0]?.detectedLanguage}`);
}

detectLanguage("En un lugar de la mancha de cuyo nombre no quiero acordarme"), // es
detectLanguage("There was a wolf near the castle where the king lived") // en
detectLanguage("An einem Ort in La Mancha, an dessen Namen ich mich nicht erinnern möchte") // de
```

As you can see, even though you created the object without expecting to find the German language (de), if the browser has support downloaded, it's able to detect it without issue. What the language array tells you is that if you need to download a language that isn't installed, you should do so immediately.

Translation

To generate a translation from one language to another, we can use the **Translator** class . First, we'll create an object of the class by calling the static **create** method and passing it the language of the original text and the one we want to translate into. Once the object is created, we call the **translate** method and pass it the original text. It will return a promise with the translated text, or an error if something goes wrong.

> If we try to test this feature as soon as the page loads, we may see an error when creating the object, indicating that it requires user interaction. To be sure, it's best to link this feature to a call generated from an event or similar.

 # index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script type="module" src="index.js"></script>
</head>
<body>
    <p><textarea type="text" id="input" rows="5" cols="50"></textarea></p>
    <p>
        <select id="lang">
            <option value="en">English</option>
            <option value="es">Spanish</option>
            <option value="de">German</option>
        </select>
        <button id="button">Translate</button></p>
    <p><textarea type="text" id="output" rows="5" cols="50" readonly></textarea></p>
</body>
</html>
```

 # index.js

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script type="module" src="index.js"></script>
</head>
<body>
    <p><textarea type="text" id="input" rows="5" cols="50"></textarea></p>
    <p>
        <select id="lang">
            <option value="en">English</option>
            <option value="es">Spanish</option>
            <option value="de">German</option>
        </select>
        <button id="button">Translate</button></p>
    <p><textarea type="text" id="output" rows="5" cols="50" readonly></textarea></p>
</body>
</html>
```

Translate in streaming

The original text may be long and take a long time to process, creating a wait that the user may interpret as something wrong. Therefore, we can have the translator return *pieces* of the translated text to us, and we can gradually add them to the interface, so the user understands that the translation is progressing. For this, we have the **translateStreaming** method .

 # index.js

```
//...

async function translate(text, input, output) {
  // Creamos objeto Translator
  return translator.translateStreaming(text);
}

button.addEventListener("click", async () => {
```

```
  button.disabled = true;
  try {
    //...
    const stream = await translate(text, inputLang, outputLang);
    output.value = "";
    for await (const chunk of stream) {
      console.log(chunk);
      output.value += chunk;
    }
  } catch (error) {
    output.value = error.toString();
  } finally {
    button.disabled = false;
  }
});
```

At the time of writing, at least in the Chrome browser, this feature doesn't work as expected, returning a single chunk of translated text.

Generate summaries

The Summarizer API allows you to generate different types of summaries from a longer text. It generates a summary of different types depending on the context provided.

To create the **Summarizer** object , we use the static create method. This method is passed several options:

- **sharedContext** : Provides information to the model about the nature of the summary
- **type** : Type of summary to generate. Its value can be: **headline** (a headline that captures the essence of the text), **key-points** (list of the most important points), **teaser** (summary of the most interesting points), **tldr** (general summary of the text).
- **length** : Summary length. Values can be short, medium, or long. This has a different effect depending on the summary type. Learn more about the length parameter .
- **format** : Format of the generated text. Can be **markdown** or **plain-text** .
- **expectedInputLanguages** : Array of languages in which the original text could be, and should be supported by the translator
- **outputLanguage** : Language in which to generate the summary.

Next, we call the summarize method on the created object, and it will return a promise with the generated summary text, or an error if something went wrong.

 # index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script type="module" src="index.js"></script>
</head>
<body>
    <p><textarea type="text" id="input" rows="5" cols="50"></textarea></p>
    <p>
        <select id="lang">
            <option value="en">English</option>
            <option value="es">Spanish</option>
            <option value="de">German</option>
        </select>
        <button id="button">Summarize</button></p>
    <p><textarea type="text" id="output" rows="5" cols="50" readonly></textarea></p>
</body>
</html>
```

 # index.js

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script type="module" src="index.js"></script>
</head>
<body>
    <p><textarea type="text" id="input" rows="5" cols="50"></textarea></p>
    <p>
        <select id="lang">
            <option value="en">English</option>
            <option value="es">Spanish</option>
            <option value="de">German</option>
        </select>
        <button id="button">Summarize</button></p>
    <p><textarea type="text" id="output" rows="5" cols="50" readonly></textarea></p>
</body>
</html>
```

As with the Translation API, the Summary API can also return the result as a stream using the **summarizeStreaming** method .