# Introducción

A continuación, se describirán las principales características de la solución **MazeMapper**.

El repositorio se encuentra alojado en la página <https://github.com/reb0rt081/MazeMapper>. Ahora mismo en el repositorio coexisten 4 ramas, aunque la rama principal es *master*.

# Estructura

* **Node.js**:

Allows you to run web apps based on JavaScript code and run these files like they are web page files or scripts to execute.

It also allows you to download all Project dependencies so you have all libraries downloaded upon start. All npm packages contain a file, usually in the project root, called package.json - this file holds various metadata relevant to the project. This file is used to give information to npm that allows it to identify the project as well as handle the project's dependencies. It can also contain other metadata such as a project description, the version of the project in a particular distribution, license information, even configuration data - all of which can be vital to both npm and to the end users of the package. The package.json file is normally located at the root directory of a Node.js project.

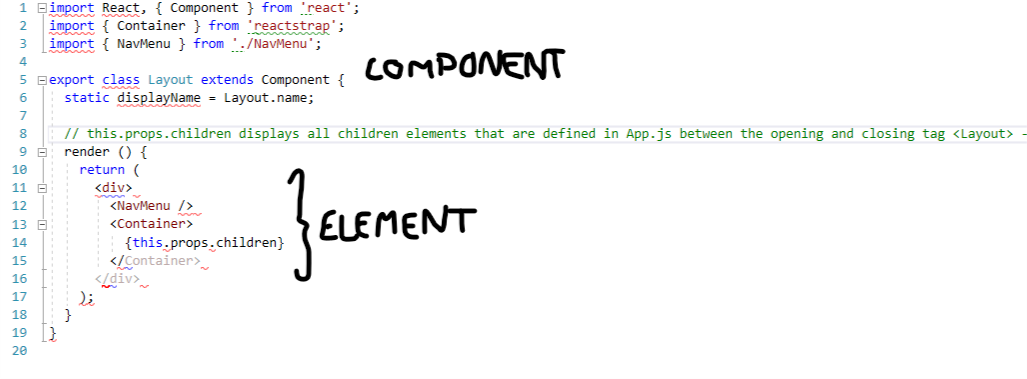
* **React**:

React can be used as a UI library to render elements, without enforcing a specific project structure, and that’s why it’s not strictly a framework.

React Elements are the smallest building blocks of React apps. They are more powerful than DOM elements because the React DOM makes sure to update them efficiently whenever something changes.

Components are larger building blocks that define independent and reusable pieces to be used throughout the application. They accept inputs called props and produce elements that are then displayed to the user.

React is based on JavaScript, but it’s mostly combined with JSX (JavaScript XML), a syntax extension that allows you to create elements that contain HTML and JavaScript at the same time.



* **Angular**:

Projects in Angular are structured into Modules, Components, and Services. Each Angular application has at least one root component and one root module.

Each component in Angular contains a Template, a Class that defines the application logic, and MetaData (Decorators). The metadata for a component tells Angular where to find the building blocks that it needs to create and present its view.

Angular templates are written in HTML but can also include Angular template syntax with special directives to output reactive data and render multiple elements, among other things.

Services in Angular are used by Components to delegate business-logic tasks such as fetching data or validating input. They are a distinct part of Angular applications. While Angular doesn’t enforce their use, it’s highly suggested to structure apps as a set of distinct services that can be reused.

Angular is built in TypeScript, so its use is recommended to get the most seamless experience, but plain JavaScript is also supported.

* **Vue**:

The Vue.js core library focuses on the View layer only. It’s called a progressive framework because you can extend its functionality with official and third-party packages, such as Vue Router or Vuex, to turn it into an actual framework.

Although Vue is not strictly associated with the MVVM (Model-View-ViewModel) pattern, its design was partly inspired by it. With Vue, you’ll be working mostly on the ViewModel layer, to make sure that the application data is processed in a way that allows the framework to render an up-to-date View.

Vue’s templating syntax lets you create View components, and it combines familiar HTML with special directives and features. This templating syntax is preferred, even though raw JavaScript and JSX are also supported.

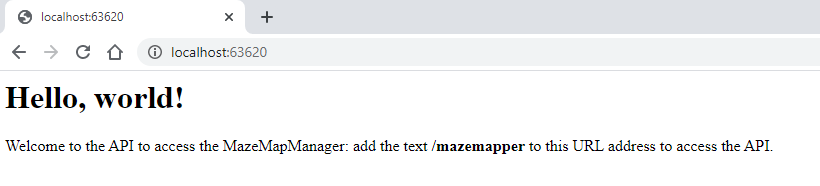
Components in Vue are small, self-contained, and can be reused throughout the application. Single File Components (SFCs) with the .vue extension contain HTML, CSS, and JavaScript so that all relevant code resides in one file.

SFCs are the recommended way to organize code in Vue.js projects, especially larger ones. Tools such as Webpack or Browserify are required to transpile SFCs into working JavaScript code.

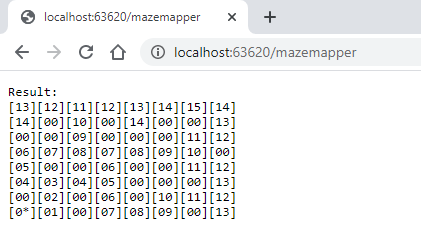
# Descripción de la solución

* *MazeMapper.Web.Backend.sln*:

API backend en .NET core que al ejecutar expone una interfaz a cualquier cliente web que intente acceder a la ruta *//http/localhost:[puerto]/mazemapper*. Por defecto el *[puerto]* debería ser el 63620.

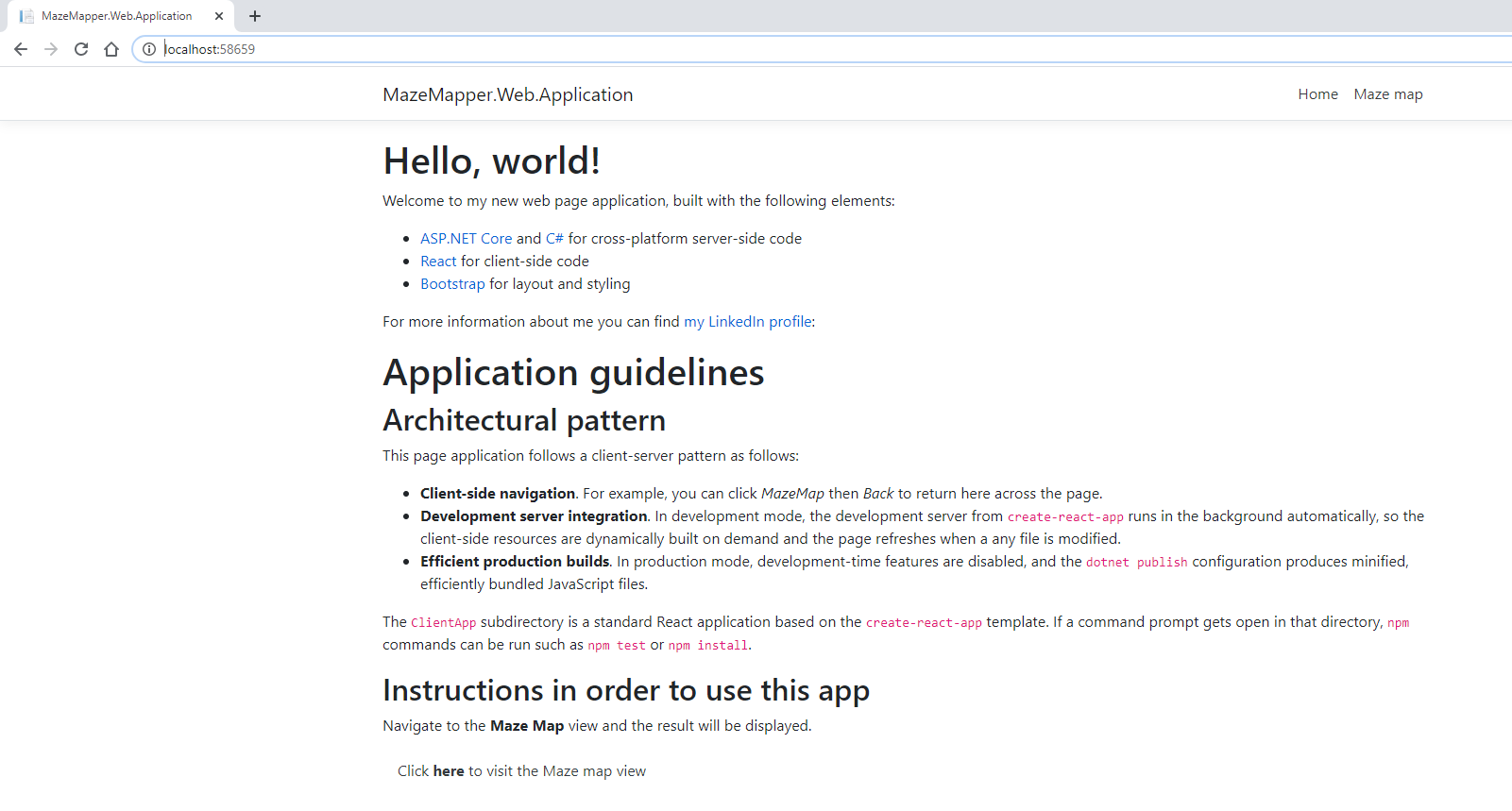


Página por defecto de la API backend.

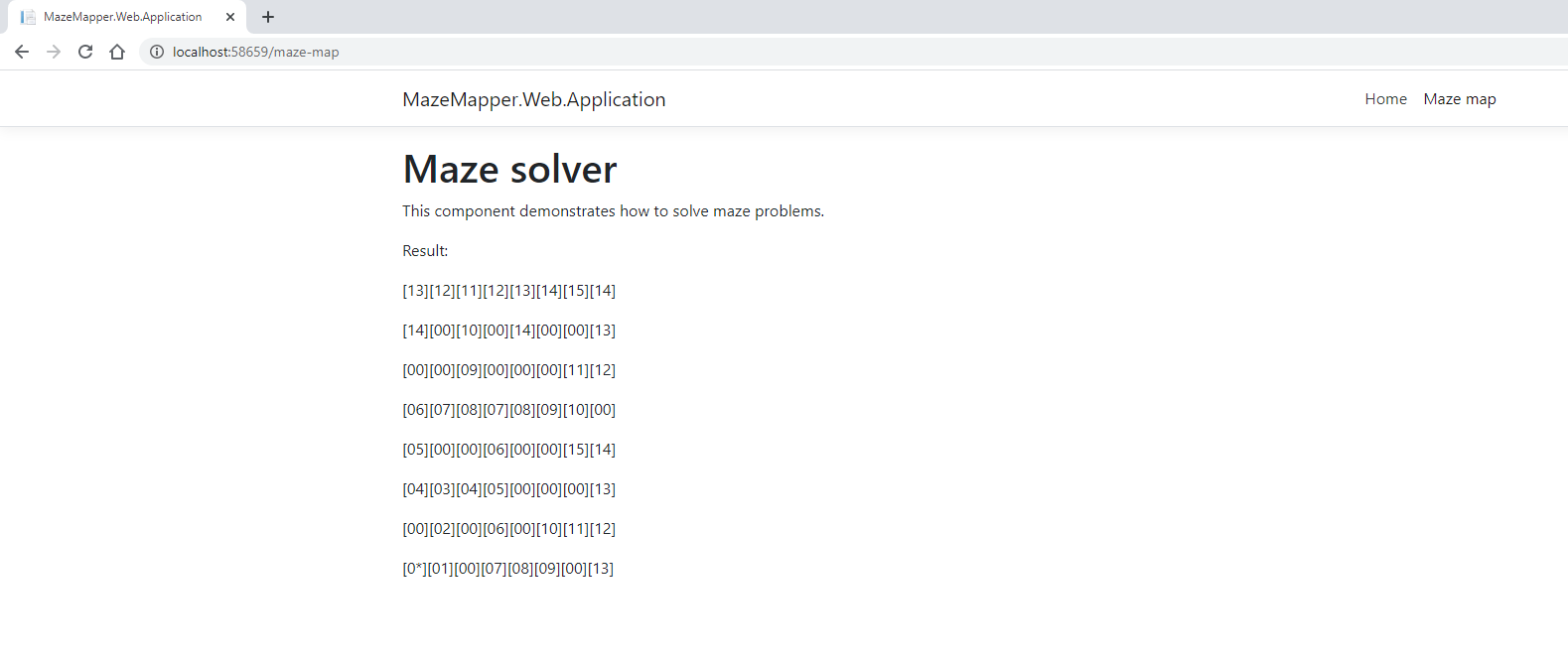


API backend que muestra el resultado.

* *MazeMapper.Web.Frontend.sln*: aplicación web en *React* que, tras presentar una vista con la explicación de la solución, permite navegar a una vista que ofrece la solución a un laberinto por defecto. La aplicación permite navegar entre estas dos páginas sin mayor problema así como a otras páginas externas.



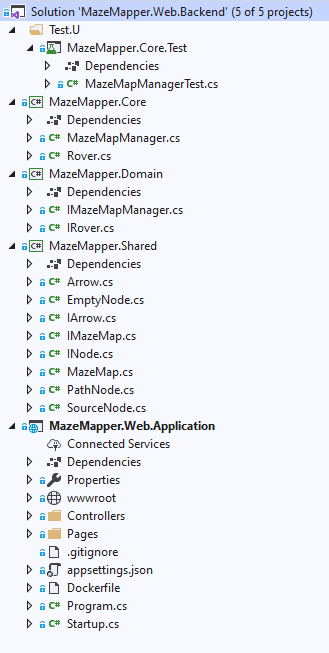
Vista “Home”.



Vista “Maze map”.

## Backend

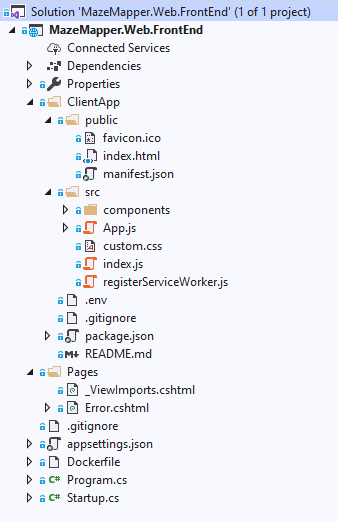
El **Backend** está formado por una solución que contiene 5 proyectos o librerías



* **MazeMapper.Core.Test**. Librería con test unitarios para testear el MazeMapper.
* **MazeMapper.Core**. Librería que incluye la clase **Rover** que permite recorrer laberintos y el **MazeMapManager** que asiste en la resolución de un laberinto así como gestiona y sincroniza la exploración de los Rovers. Cada instancia de **MazeMapManager** resuelve un laberinto.
* **MazeMapper.Domain**. Librería con las interfaces utilizadas para definir un objeto **Rover** y **MazeMapManager**.
* **MazeMapper.Shared**. Contiene la estructura básica de los objetos necesarios para construir un laberinto.
* **MazeMapper.Web.Application**. Librería que monta una API backend que expone el **MazeMapManager** en una *endpoint* y resuelve un laberinto por defecto.

## Frontend

El **Frontend** está formado por una solución que contiene únicamente un proyecto que consiste en una librería para ejecutar una aplicación web con *react*.



## Funcionamiento del MazeMapManager (backend)

El **MazeMapManager** permite resolver un laberinto, estableciendo el mínimo número de pasos que hay que dar desde cualquier casilla a la salida.

El **MazeMapManager** una vez inicializado permite cargar un laberinto utilizando el método .BuildMazeMapFromString().

El formato de los datos de entrada puede ser un texto formado por 1s, 0s y un asterisco (\*) que define la salida del laberinto.

000010

\*11110

000000

En el ejemplo de arriba se puede ver un laberinto donde los 0s indican casillas vacías y los 1s casillas que representan pasillos. El asterisco indica la salida del laberinto. Es necesario escribir este texto añadiendo nuevas líneas y dotándolo de una estructura rectangular (donde todas las líneas de texto tienen el mismo número de caracteres).

Una vez introducido el laberinto el **MazeMapManager** lo carga con una abstracción del laberinto formada por una matriz (que representa todas las casillas) nodos (que representan casillas vacías, caminos o la casilla de salida) y flechas (que representan conexiones entre casillas de pasillo o la casilla de salida).

Una vez cargado el laberinto en el **MazeMapManager** se puede resolver dicho laberinto llamando al método.SolveMazeAsync(). Se trata de una tarea asíncrona que devuelve un texto como el siguiente:

[13][12][11][12][13][14][15][14]

[14][00][10][00][14][00][00][13]

[00][00][09][00][00][00][11][12]

[06][07][08][07][08][09][10][00]

[05][00][00][06][00][00][11][12]

[04][03][04][05][00][00][00][13]

[00][02][00][06][00][10][11][12]

[0\*][01][00][07][08][09][00][13]

* *[00]*: indica casillas vacías.
* *[0\*]*: indica la casilla de salida.
* *[09]*: indica que se necesita dar un mínimo de 9 pasos para llegar a la salida desde esta casilla.

Hay una serie de *unit tests* definidos que aseguran el correcto funcionamiento de esta clase.

Cuando se carga la API backend por defecto se define un laberinto como el siguiente:

11111111

10101001

00100011

11111110

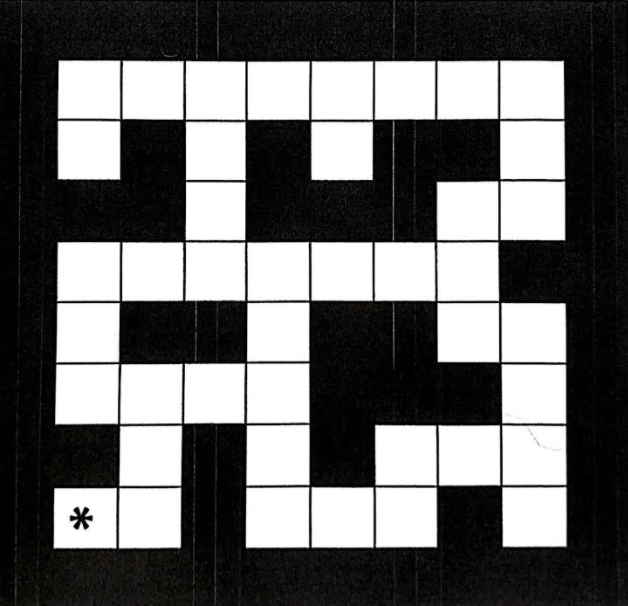
10010011

11110001

01010111

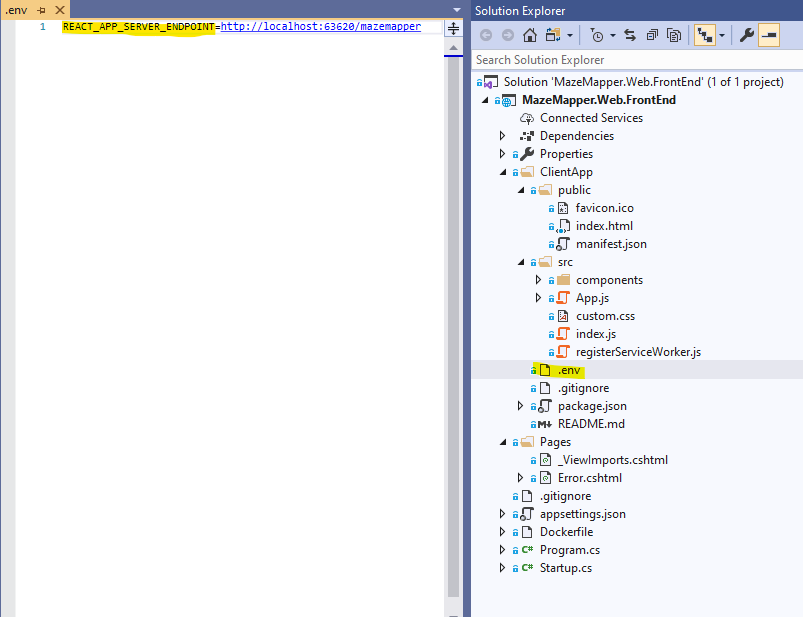
\*1011101

Que se corresponde con este laberinto:



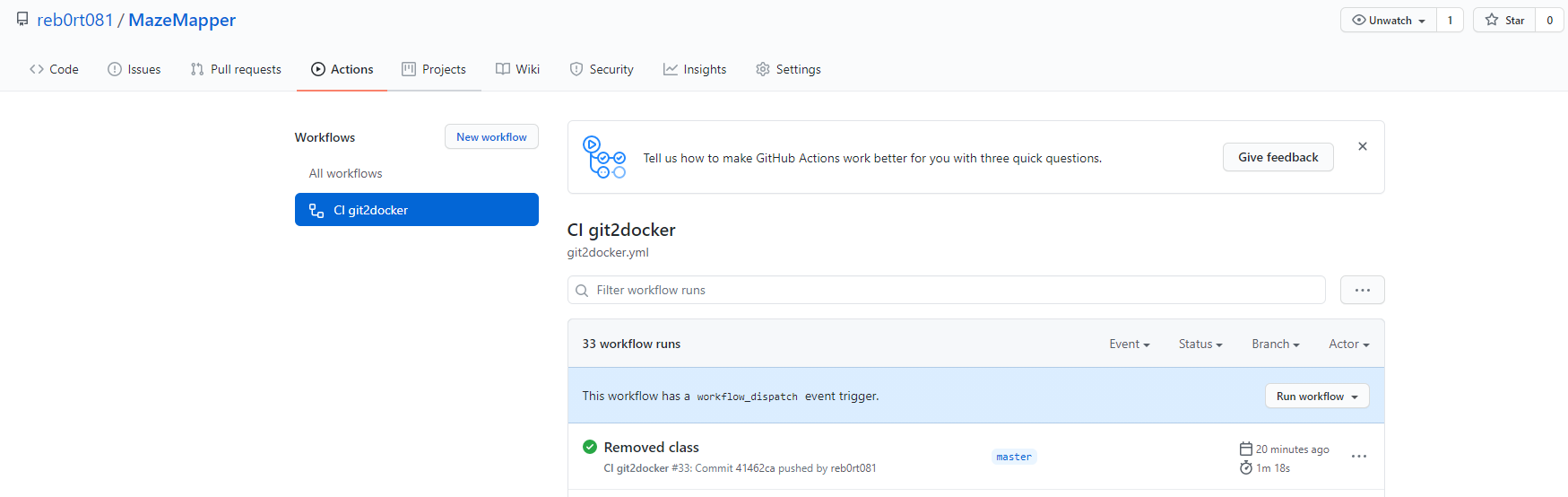
## Funcionamiento del frontend

Para que la aplicación cliente web funcione necesita que se ajuste la variable REACT\_APP\_SERVER\_ENDPOINT para que apunte al servidor. El puerto por defecto de la parte backend debería ser el 63620.

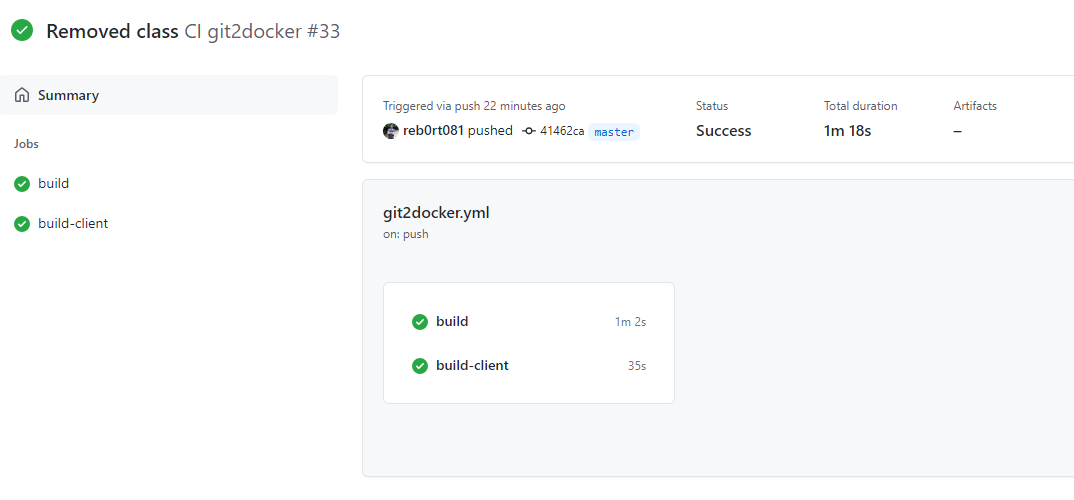


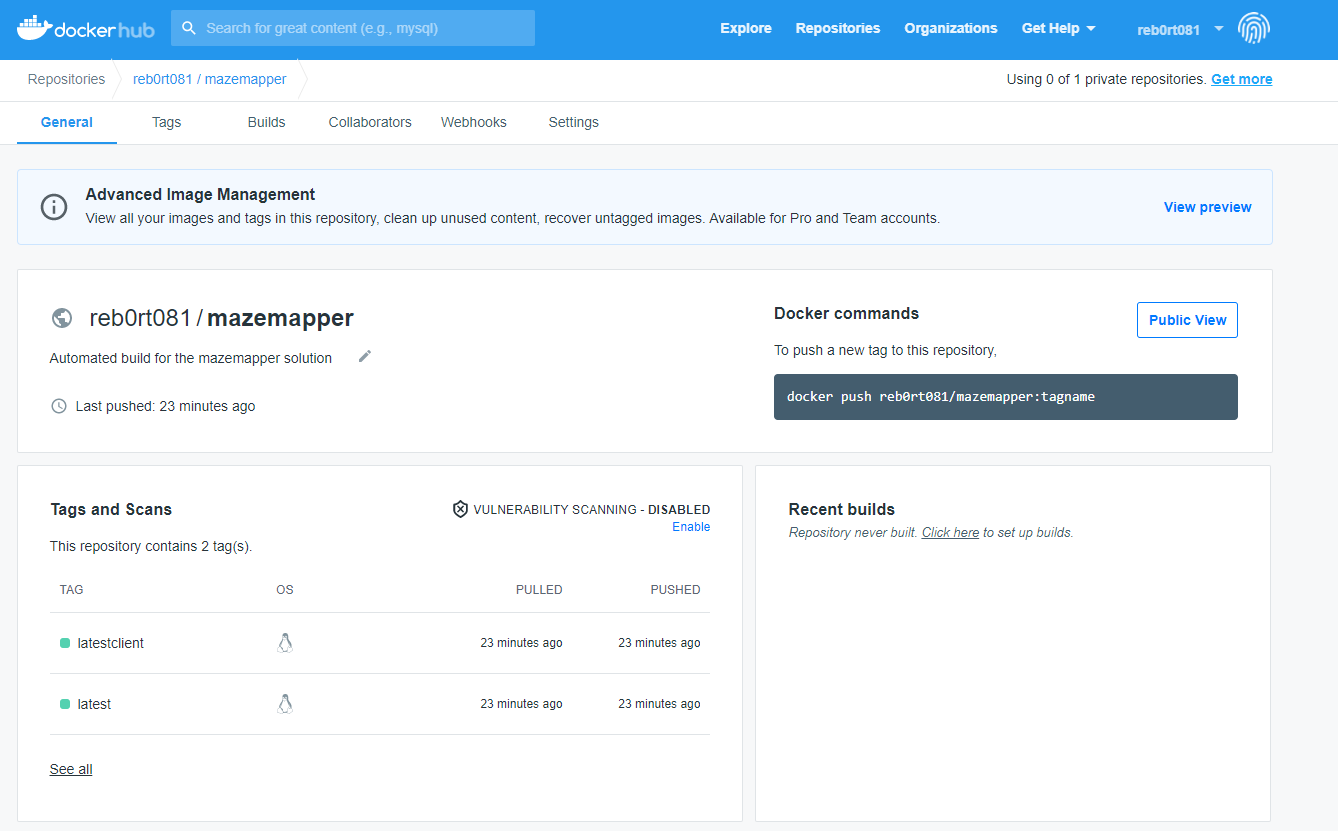
# Despliegue de la solución y automatización

Existe un archivo *.yml* (*git2docker.yml*), disponible en el repositorio *github,* que se ejecuta automáticamente cada vez que hay un *push* o una *pull request* en la rama *master*.



Este código ejecuta dos *jobs* que permiten crear dos imágenes en *Docker Hub* a partir de los correspondientes *Dockerfile*: uno para crear una imagen de la API backend y otro para la imagen de la aplicación web cliente.





En teoría estas imágenes podrían instalarse y las aplicaciones podrían ejecutarse.

Desgraciadamente con mi portátil tengo problemas para ejecutar *Docker* (la política de mi empresa me lo impide) y tampoco he podido utilizar *Docker compose*.

Tanto el archivo *.yml* como los *Dockerfile* se encuentran disponibles en el código fuente.