



Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**



Grado en Ingeniería Informática

Trabajo Fin de Grado

**Desarrollo de un Plugin Visual Studio
Code para EasyInterface**

Autor: Marta Toca Carballo

Tutor: Guillermo Román Díez

Madrid, julio 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado

Grado en Ingeniería Informática

Título: Desarrollo de un Plugin Visual Studio Code para EasyInterface

Julio 2025

Autor: Marta Toca Carballo

Tutor: Guillermo Román Díez

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software

ETSI Informáticos

Universidad Politécnica de Madrid

Resumen

EasyInterface permite a investigadores ofrecer herramientas de análisis de código en una misma plataforma y cuenta actualmente con un cliente web, un plugin de Eclipse y salida por la terminal. Este trabajo de fin de grado busca la creación e implementación de una extensión de Visual Studio Code que sirva de cliente al ya existente servidor de EasyInterface.

La proliferación de herramientas de análisis desarrolladas por la comunidad investigadora suele requerir interfaces específicas o conocimientos avanzados para su uso. Debido al creciente uso de un entorno de desarrollo ampliamente utilizado como Visual Studio Code, es crucial acercar dichas herramientas para facilitar su acceso y uso.

Para el desarrollo de la extensión se ha utilizado TypeScript y la API de extensiones de VSCode, siguiendo una arquitectura modular que permite la integración sencilla de nuevas funcionalidades en un futuro. Este proceso ha incluido la revisión de requisitos, diseño de la interfaz de usuario, implementación de la comunicación con el servidor EasyInterface y validación mediante pruebas.

La extensión permite ejecutar herramientas de análisis alojadas en EasyInterface directamente desde VSCode, visualizar los resultados en el propio editor y gestionar la interacción de forma intuitiva. Se ha validado su funcionamiento con varios casos de uso reales y se ha comprobado su facilidad de uso y adaptabilidad a diferentes contextos.

Abstract

EasyInterface enables researchers to offer code analysis tools on a single platform, currently providing a web client, an Eclipse plugin and remote shell. This final degree project aims to create and implement a Visual Studio Code extension that serves as a client for the existing EasyInterface server.

The proliferation of analysis tools developed by the research community often requires specific interfaces or advanced knowledge to use. Due to the increasing adoption of a widely used development environment like Visual Studio Code, it is crucial to bring these tools closer to users to facilitate their access and usage.

The extension has been developed using TypeScript and the VSCode extension API, following a modular architecture that allows for the easy integration of new functionalities in the future. This process has included requirements review, user interface design, implementation of communication with the EasyInterface server, and validation through testing.

The extension allows users to execute analysis tools hosted on EasyInterface directly from VSCode, view results within the editor itself, and manage interactions intuitively. Its functionality has been validated with several real-world use cases, demonstrating its ease of use and adaptability to different contexts.

Tabla de contenidos

1 Introducción	1
1.1 Definición del trabajo	1
1.2 Objetivos	2
2 Trabajos previos	3
3 Diseño	4
3.1 Extensión en Visual Studio Code	4
3.2 Conexión con EasyInterface	5
3.3 Menú de selección de parámetros	8
3.4 Outline del código	12
3.5 Selector de aplicación	14
3.6 Envío a EasyInterface	16
3.7 Recepción de la respuesta y salida en pantalla	17
4 Manual de usuario	22
5 Resultados y conclusiones	50
6 Análisis de impacto	51
7 Bibliografía	52
8 Anexos	53

1 Introducción

1.1 Definición del trabajo

EasyInterface es una interfaz desarrollada en el contexto de proyectos de investigación, cuyo propósito principal es facilitar a los investigadores la integración rápida de los resultados obtenidos a partir de distintos análisis realizados sobre su código fuente. Esto incluye la posibilidad de añadir información directamente en la interfaz de consola, así como el uso de elementos como marcadores e interacciones con el código fuente.

Actualmente, EasyInterface cuenta con una interfaz web que permite integrar estos análisis de forma visual y accesible. Sin embargo, los editores de código avanzados y entornos de desarrollo integrados como Visual Studio Code de Microsoft (VSC o VSCode de aquí en adelante) han supuesto una gran mejoría en los últimos años para los desarrolladores. VSCode ofrece una experiencia ágil y sencilla, adaptándose a las necesidades específicas de cada desarrollador con su gran número de extensiones y herramientas.

Una extensión de VSCode de EasyInterface serviría para integrar directamente en el entorno de desarrollo de manera rápida y eficiente los resultados de análisis sobre el código fuente así como un menú de funcionalidades y parámetros del análisis a realizar. De esta manera, la extensión permitiría a los investigadores visualizar, interactuar y gestionar todo el proceso, desde la configuración hasta la visualización de resultados, desde una única herramienta de trabajo.

El enfoque escogido va a ser integrar al máximo la extensión con la propia interfaz. Para ello, comenzaremos por realizar una investigación exhaustiva de cómo desarrollar una extensión en TypeScript, así como de cómo funciona EasyInterface, qué tipo de análisis dispone y qué tipo de salidas produce una llamada, de manera que podamos mostrarlas al usuario de distintas maneras (consola, marcadores, resaltado de palabras o líneas, etc).

Después de la investigación inicial, en el desarrollo iremos añadiendo funcionalidades de forma gradual, siguiendo un modelo de ciclo de vida incremental. Según el tiempo del que se disponga, contando con posibles imprevistos, se podrán añadir funcionalidades no contempladas al inicio.

Por último, la extensión será publicada para que pueda estar disponible y ser utilizada por la comunidad de investigadores. Con el fin de facilitar su adopción y garantizar que los usuarios puedan aprovechar al máximo todas sus funcionalidades, se elaborarán guías de uso detalladas, así como tutoriales prácticos. Estos recursos incluirán ejemplos de configuración, envío de análisis al servidor, interpretación de resultados y personalización de la extensión dentro de Visual Studio Code. De esta manera, se busca que la herramienta sea accesible incluso para quienes no estén familiarizados previamente con EasyInterface.

1.2 Objetivos

En resumen, los objetivos del proyecto incluyen los siguientes:

- Estudiar la creación de un plugin en VS Code.
- Conectar con el servidor al iniciar y recibir la lista de parámetros de configuración.
- Mostrar al usuario los análisis disponibles en un menú.
- Mostrar al usuario los parámetros pertinentes a cada tipo de análisis.
- Mostrar e interactuar con el outline del código fuente.
- Enviar al servidor la petición del análisis.
- Procesar la información del servidor y mostrarla al usuario.
- Implementar la extensión de una manera cohesiva con la interfaz de VS Code, manteniendo la apariencia característica del editor.
- Memoria y presentación.

2 Trabajos previos

2.1 Easyinterface Web: una base sólida para la extensión.

EasyInterface es un framework para facilitar la creación de interfaces gráficas desde una línea de comandos y un output estándar. Su arquitectura cuenta con un servidor, donde se alojan las aplicaciones, y un cliente con interfaces visuales para configurar, lanzar y visualizar los resultados del análisis.

En la versión actual cuenta con una interfaz web con las siguientes características: editor de código, consola integrada, gestor de archivos y un panel de navegación del *outline* del código. Los resultados de los análisis pueden representarse usando marcadores, resaltadores de fragmentos de código, cuadros de diálogo, mensajes emergentes, etc.

A pesar de su gran utilidad, el uso de EasyInterface continúa limitado a la web, lo que resulta en una desconexión con entornos de desarrollo muy utilizados como Visual Studio Code. Debido a esto, una extensión nativa en este editor permitiría a los usuarios incorporar todo el proceso de análisis en su rutina de trabajo habitual.

2.2 Extensiones similares en VSCode.

Visual Studio Code se encuentra entre los editores de código más destacados, especialmente gracias a su ecosistema de extensiones que permite a los usuarios adaptar su entorno según sus requerimientos. En el ámbito del análisis de código, hay varias extensiones que brindan características parecidas a las de EasyInterface, incluyendo:

- **SonarLint:** proporciona análisis estático en tiempo real para detectar errores, vulnerabilidades y malas prácticas.
- **CodeQL:** desarrollado por GitHub, permite consultas semánticas sobre el código para detectar patrones complejos de errores o inseguridades.
- **ESLint / Pylint:** herramientas ampliamente adoptadas para verificar estilo y calidad en JavaScript y Python, respectivamente.

Estas extensiones ilustran cómo el análisis integrado directamente en el editor potencia la productividad y reduce el cambio de contexto. La extensión de EasyInterface para VSCode se basa en este enfoque, permitiendo a los usuarios enviar análisis al servidor EasyInterface, configurar parámetros desde una interfaz amigable, y visualizar los resultados directamente en el editor.

3 Diseño

3.1 Extensión en Visual Studio Code

Una extensión en Visual Studio Code permite ampliar la experiencia general de uso, agregando más características para extender su capacidad. Esto se puede realizar añadiendo funcionalidades del editor, integrando nuevas herramientas, paneles, comandos, etc.

Para comenzar a desarrollar nuestra extensión en Visual Studio Code, los requisitos previos incluyen la previa instalación de [Node.js](#), un entorno de ejecución JavaScript utilizado para desarrollar aplicaciones escalables del lado del servidor y de red, Git, un sistema distribuido de control de versiones, y por último el propio Visual Studio Code.

Una vez tenemos todo instalado, usamos Yeoman (`yo code`), una herramienta de scaffolding de proyectos, junto con el generador específico “yo code” para crear la estructura inicial de un proyecto de TypeScript. Hemos elegido TypeScript, un superconjunto tipado de JavaScript recomendado oficialmente para el desarrollo de extensiones en VSCode, por sus numerosas ventajas: definición de tipos explícitos, interfaces y clases, lo cual ayuda a detectar errores en tiempo de compilación y facilita el mantenimiento; integración avanzada en VSCode con mejor experiencia de desarrollo y una depuración más sencilla debido a su tipado estático. TypeScript se transpila a JavaScript, asegurando la compatibilidad con el entorno de ejecución de extensiones de VSCode.

La estructura básica de un proyecto en VS Code tras usar Yeoman es la siguiente:

```
.
├── .vscode
│   ├── launch.json    // configuración para lanzar y depurar la
extensión
│   └── tasks.json      // configuración para compilación de TS
├── .gitignore          // ignora la salida de compilación y la
carpeta node_modules
├── README.md           // descripción de la extensión
├── src
│   └── extension.ts    // código fuente de la extensión
├── package.json        // manifiesto de la extensión
└── tsconfig.json       // configuración de TypeScript
```

Los dos archivos más importantes son `extension.ts` y `package.json`. El archivo `extension.ts` contiene el código principal de la extensión, funcionando como su backend. Aquí se implementan las funciones, comandos y manejadores de eventos que definen el comportamiento de la extensión cuando está activa. Siguiendo el modelo vista-controlador, `extension.ts` actuaría como el controlador.

La función `activate` de este archivo se ejecuta cuando ocurre un evento de activación. Mientras el archivo `extension.ts` es el punto de entrada de la extensión, por modularidad y separación de responsabilidades, la carpeta contenedora `src` albergará el resto de módulos o utilidades según crezcan las funcionalidades del proyecto.

Por su parte, el archivo `package.json` declara los comandos disponibles y los eventos que activan la extensión (contribution points), actuando como el manifiesto que describe la estructura y capacidades de la extensión para Visual Studio Code. Por otro lado, también incluye información esencial como el nombre, versión, descripción, dependencias y compatibilidad con versiones de VSCode. Estos metadatos son esenciales para la publicación y correcta integración de la extensión en el Marketplace.

Esta estructura inicial, generada automáticamente, constituye la base sobre la que se desarrolla toda la lógica y funcionalidad de la extensión.

3.2 Conexión con EasyInterface

La arquitectura del servidor según lo expresado en su documentación es la siguiente:

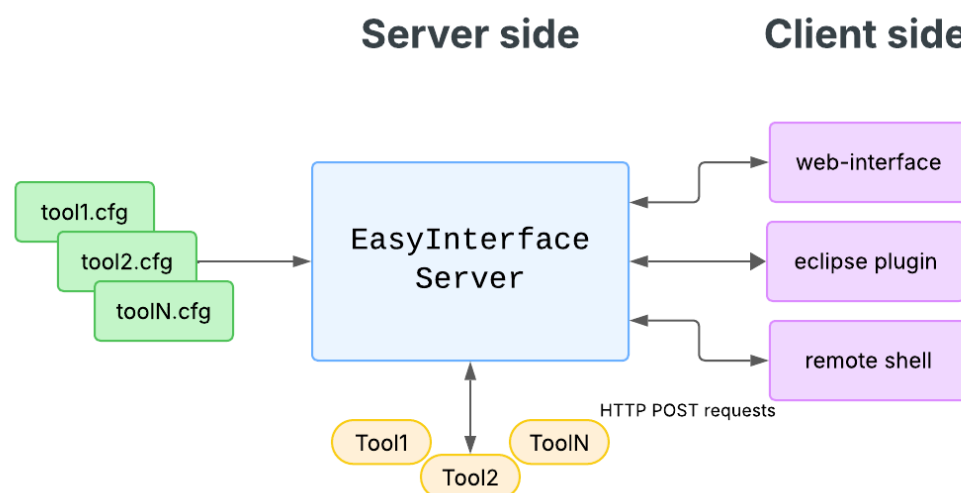


Figura 3.2.1: Arquitectura original EIServer

Existen dos partes principales: el lado del servidor y el lado del cliente. En el lado del servidor, el EIServer se encarga de gestionar la ejecución de las distintas herramientas del sistema. Cada herramienta tiene un archivo de configuración específico que define cómo debe ejecutarse y qué parámetros acepta. Cuando un cliente solicita la ejecución de una herramienta, el servidor utiliza dichos archivos para lanzarla, recoger su salida y enviársela de vuelta al cliente, quien mostrará su salida de forma acorde.

El flujo de ejecución desde un cliente es de la siguiente manera:

1. Conectar al servidor EasyInterface, solicitando la lista de aplicaciones disponibles.
2. Permitir al usuario seleccionar una aplicación a ejecutar y establecer los parámetros correspondientes.

3. Generar la petición correspondiente y enviarla a EasyInterface.
4. Mostrar la salida devuelta al usuario.

El cliente, a su vez, debe contar con distintos elementos para asegurar la experiencia del usuario, como un editor de código, una consola, un outline para seleccionar solo ciertas funciones a enviar al servidor, un manejador de archivos y un menú de parámetros editables. Estas características se pueden apreciar en el siguiente cliente web.

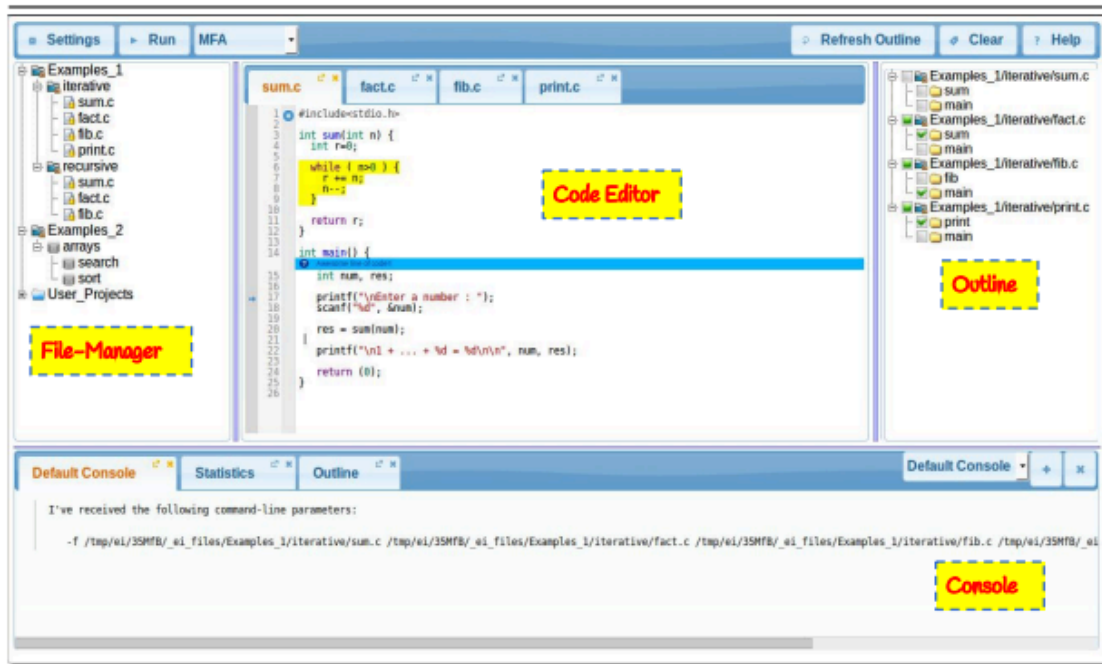


Figura 3.2.2: Ejemplo de cliente de EasyInterface (cliente web).

Este proyecto consiste en la creación de un cliente más en la forma de una extensión de VSCode por lo que el diagrama quedará de la siguiente manera:

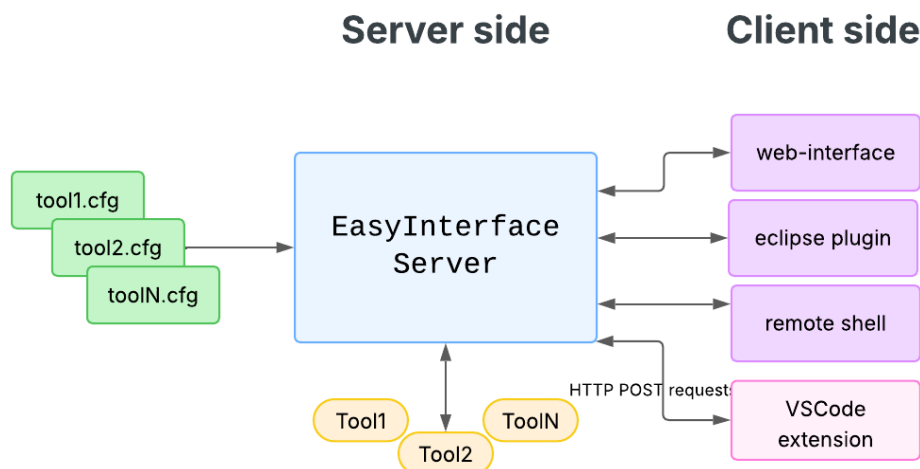


Figura 3.2.3: Arquitectura EIServer con la extensión de VSCode

Dicho cliente debe cumplir las funcionalidades mencionadas y ofrecer los mismos servicios al usuario. VSCode ya cuenta con un sistema de archivos y un editor de código propio, por lo que comenzamos por el primer paso del flujo: “Conectar al servidor EasyInterface, solicitando la lista de aplicaciones disponibles”.

De cara a enviar las peticiones a EasyInterface, creamos una interfaz `EIRequest` basada en la documentación con los siguientes campos:

```
export interface EIRequest {  
    command: "execute" | "app_info" | "app_parameters" |  
    "app_details";  
    app_id: string;  
    parameters?: Parameters;  
}
```

En este caso, es obligatorio incluir el comando que se va a ejecutar, que será “app_details” cuando se desea obtener todos los parámetros de todas las aplicaciones disponibles. Además, se utiliza el valor “ei_all” en el campo `app_id` para indicar que la consulta es global. El campo `parameters` permanece vacío en esta petición y su uso se explicará más adelante.

Siguiendo el principio de separación de responsabilidades, creamos un archivo `apiClient.ts` dentro de la carpeta `core`, encargado exclusivamente de la comunicación con EasyInterface. Este archivo contendrá la siguiente función asíncrona:

```
async function sendPostRequest(data: any): Promise<string> {}
```

Esta función recibe un objeto `EIRequest`, lo convierte al formato adecuado (urlencoded) y realiza una petición POST a EasyInterface, devolviendo la respuesta en formato XML como una cadena de texto.

La estructura de carpetas del proyecto refleja esta organización modular.

```
.  
├── ...  
├── src  
│   ├── interfaces           // interfaces y tipos de la extensión  
│   │   ├── request.ts      // definición de la petición al servidor  
│   │   └── core            // lógica principal y servicios  
│   │       ├── apiClient.ts // comunicación con EasyInterface  
│   │       └── extension.ts // código fuente de la extensión
```

Al activar la extensión, el primer paso consiste en crear un objeto `EIRequest` y realizar una llamada a la función de envío de peticiones, iniciando así la comunicación con el servidor EasyInterface. Una vez recibida la respuesta del servidor, es necesario interpretar los parámetros de herramientas y la lista de aplicaciones mediante un parser específico, que permitirá posteriormente su utilización en el menú de selección de parámetros.

3.3 Menú de selección de parámetros

El segundo paso del flujo de un cliente de EasyInterface es, como ha sido descrito anteriormente: permitir al usuario seleccionar una aplicación a ejecutar y establecer los parámetros correspondientes. Previamente hemos enviado una petición al servidor con el comando “app_details”. Este comando devuelve un xml que incluye todas las aplicaciones disponibles en el servidor junto con sus parámetros a definir y sus valores por defecto. Para poder interpretar esta salida, necesitamos desarrollar un parser específico. Sin embargo, antes de esto es fundamental comprender el formato de la respuesta obtenida.

EasyInterface describe un lenguaje de salida de texto: EasyInterface Output Language. Esto garantiza la representación de los ajustes y distintos tipos de marcadores en todos los clientes de una manera uniforme. En el archivo de configuración, la salida toma la siguiente forma:

EISERVER

```
<eiserver version=[VERSION]?>
  [SETTINGS]?
  [SANDBOX]?
  [EXAMPLES]?
  [APPS]?
</eiserver>
```

Figura 3.3.1: Estructura del archivo de configuración descrito en el manual de EasyInterface.

En la sección SETTINGS se establecen parámetros globales; SANDBOX es utilizado por los servidores de las aplicaciones para establecer límites en los recursos al ejecutar una herramienta; EXAMPLES define los ejemplos disponibles en el servidor; y APPS define las aplicaciones disponibles en el servidor. Para desarrollar nuestra extensión, nos centraremos en esta última sección para mostrarlo al usuario de forma que pueda ajustar los parámetros de forma acorde. El resto de secciones son implementadas por parte del servidor y no del cliente y aunque nuestro parser las analiza, no son relevantes.

APPS es una lista de herramientas, donde cada una tiene que ser definida por un campo APP, que toma la siguiente forma:

APP

```
<app id=[APPID] visible=[BOOL]?>
  [APPINFO]
  [APPHELP]
  [SANDBOX]
  [EXECINFO]
  [PARAMETERS]
  [PROFILES]
</app>
```

Figura 3.3.2: Estructura del campo APP descrito por EasyInterface.

El comando “app_details” devuelve todos los campos de cada APP, con la excepción de SANDBOX y EXECINFO, campos que proporcionan información de cómo ejecutar una herramienta de forma local y que por ende solo serán usados por el servidor. Consecuentemente, nuestro parser ignorará dichas secciones.

APPINFO proporciona información general sobre la herramienta, incluyendo acrónimo, título, logo y/o descripción; APPHELP proporciona información de cómo la herramienta puede ser usada; PARAMETERS es la lista de parámetros aceptados por la herramienta; y PROFILES define el set de valores por defecto para un conjunto de parámetros.

Cada elemento de la lista PARAMETERS puede adoptar diferentes formas como un selector único (elegir una sola opción), un selector múltiple (elegir una o varias opciones), una opción de activación o flag (opción activable/desactivable) o un campo de texto (introducir valores personalizados). Cada tipo de parámetro contiene nombre, descripción, prefijo, y valor (o valores) por defecto. Además, dependiendo del parámetro puede presentar otros campos, los cuales se han contemplados tanto en el parser como en el menú de parámetros de la extensión, como se puede apreciar a continuación:

SELECTONE

```
<selectone name=[PARAMNAME] prefix=[PARAMPREFIX]? check=[BOOL]? >
  [DESC]
  [OPTION]+
  [DEFAULTVALUE]?
</selectone>
```

Figura 3.3.3: Estructura del campo SELECTONE.

SELECTMANY

```
<selectmany name=[PARAMNAME] prefix=[PARAMPREFIX]? check=[BOOL]? >
  [DESC]
  [OPTION]+
  [DEFAULTVALUE]*
</selectmany>
```

Figura 3.3.4: Estructura del campo SELECTMANY.

FLAG

```
<flag name=[PARAMNAME] prefix=[PARAMPREFIX]? check=[BOOL]?
  explicit=[BOOL]? trueval=[PARAMVALUE]? falseval=[PARAMVALUE]? >
  [DESC]
  [DEFAULTVALUE]?
</flag>
```

Figura 3.3.5: Estructura del campo FLAG.

TEXTFIELD

```
<textfield name=[PARAMNAME] prefix=[PARAMPREFIX]? check=[BOOL]?  
    passinfile=[BOOL]? multiline=[BOOL]? type=[TYPE]? >  
    [DESC]  
    <initialtext>[TEXT]</initialtext>  
</textfield>
```

Figura 3.3.6: Estructura del campo TEXTFIELD.

Si se desea obtener información más detallada sobre cada campo, se recomienda consultar el manual de EasyInterface presente en la bibliografía, ya que en este apartado solo es pertinente ofrecer una visión general de la estructura.

Con el objetivo de realizar el parser, se ha creado una interfaz para cada sección descrita con todos sus campos en los archivos `settings.ts` y `common.ts`. En este último se describen tipos comunes de contenido que serán utilizados por otro parser en siguientes secciones. Además, el parser se ha descrito en el archivo `parser/settingsParser.ts` y probado por el archivo `test/testSettingsParser.ts`.

En la sección anterior hemos descrito el procedimiento para obtener el archivo de configuración. Una vez tenemos el xml, utilizamos la función `parseSettingsXML` que toma la siguiente forma:

```
async function parseSettingsXML(xml: string): Promise<Eiserver> {}
```

La función devuelve un objeto `EIServer` en formato JSON con toda la información necesaria, del que crearemos una copia denominada `mutableSettings`. Dicha copia es un archivo mutable con los ajustes de configuración que será modificada por la extensión para gestionar de forma dinámica y centralizada todos los parámetros modificables y los valores por defecto de cada herramienta.

Esta decisión responde a dos necesidades principales: por un lado, facilitar la actualización y persistencia de los cambios de configuración; por otro, mantener sincronizada la configuración entre distintos componentes de la extensión, evitando inconsistencias. Esto sigue la convención de aplicaciones de página única (SPA, Single Page Application), donde es fundamental contar con un sistema centralizado de estado para reflejar cualquier cambio de inmediato en toda la interfaz. Por lo tanto, cada vez que el usuario aplique los cambios o restaure los valores por defecto, `mutableSettings` será modificado, y al volver a mostrar los ajustes, se mostrarán los valores más recientes.

Una vez el parser está creado y la copia del JSON de los ajustes está lista para ser utilizada, registramos el comando “`displaySettings`” en `extension.ts`. Al hacer clic sobre un botón de la extensión, mostrará en una webview los ajustes disponibles y modificables por aplicación, así como un botón para restaurar los valores por defecto y un botón para guardar los valores modificados.

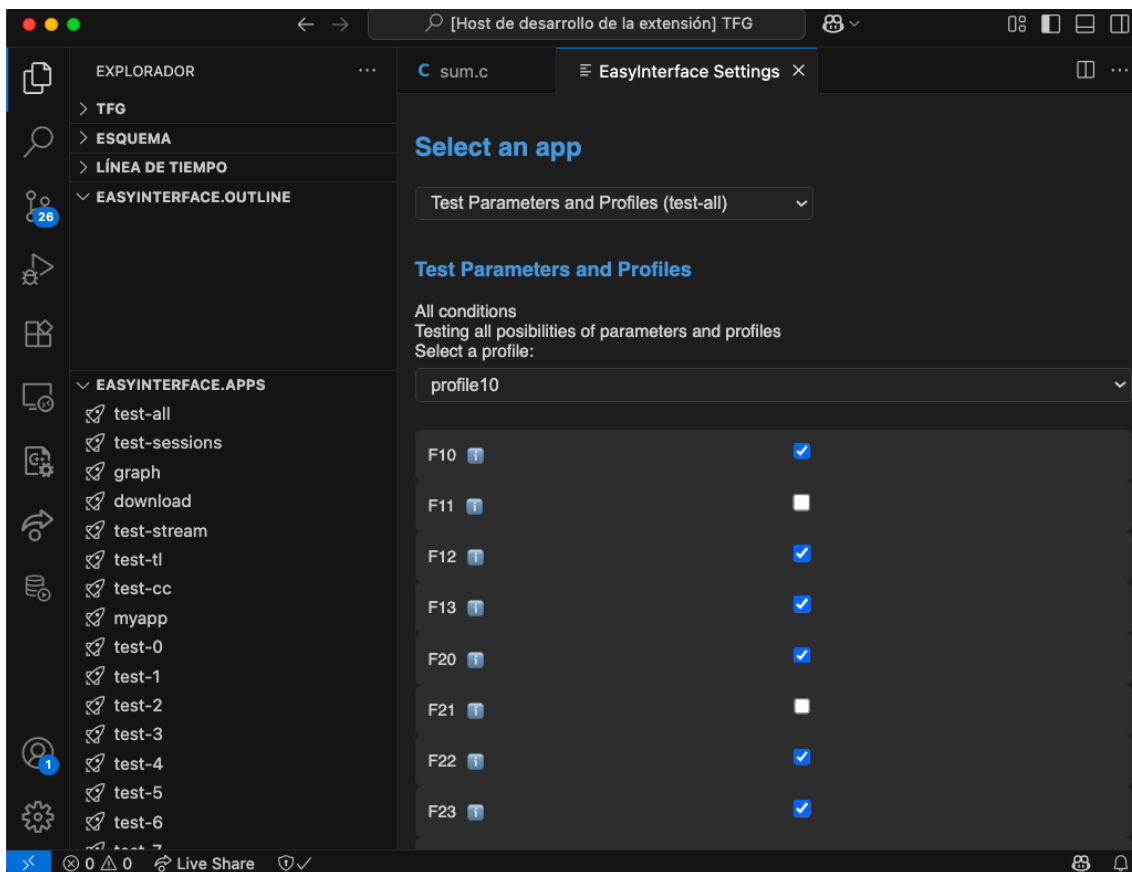


Figura 3.3.7: Webview de los ajustes.

La webview se crea desde la clase `settingsWebview.ts` dentro del directorio `webviews`. Desde ahí, se llama a la función `getWebviewController` que crea una página HTML dinámica en una nueva pestaña del editor a partir de los datos almacenados en `mutableSettings`.

Al hacer clic sobre el botón “Apply settings” en la interfaz gráfica, la webview envía un mensaje interno mediante el mecanismo de mensajería de VSCode. Este mensaje, identificado por el comando `applySettings`, contiene los nuevos valores de los ajustes y el identificador de la aplicación correspondiente. La función `panel.webview.onDidReceiveMessage` captura este mensaje y ejecuta la siguiente función:

```
onApply: (appId: string, data: any) => void
```

Esta función invoca la lógica correspondiente en `extension.ts`, donde se registran los nuevos valores en el archivo mutable de configuración `mutableSettings.json`. Al recibir el identificador de la aplicación y los datos actualizados, busca la aplicación correspondiente y actualiza el valor de cada parámetro modificado.

De este modo, el proceso de actualización de los ajustes queda completamente integrado entre la interfaz gráfica, el backend y el archivo de configuración, asegurando la sincronización y la persistencia de los cambios realizados por el usuario. El siguiente esquema muestra de forma visual todo lo anteriormente descrito con respecto al flujo de gestión de los ajustes.

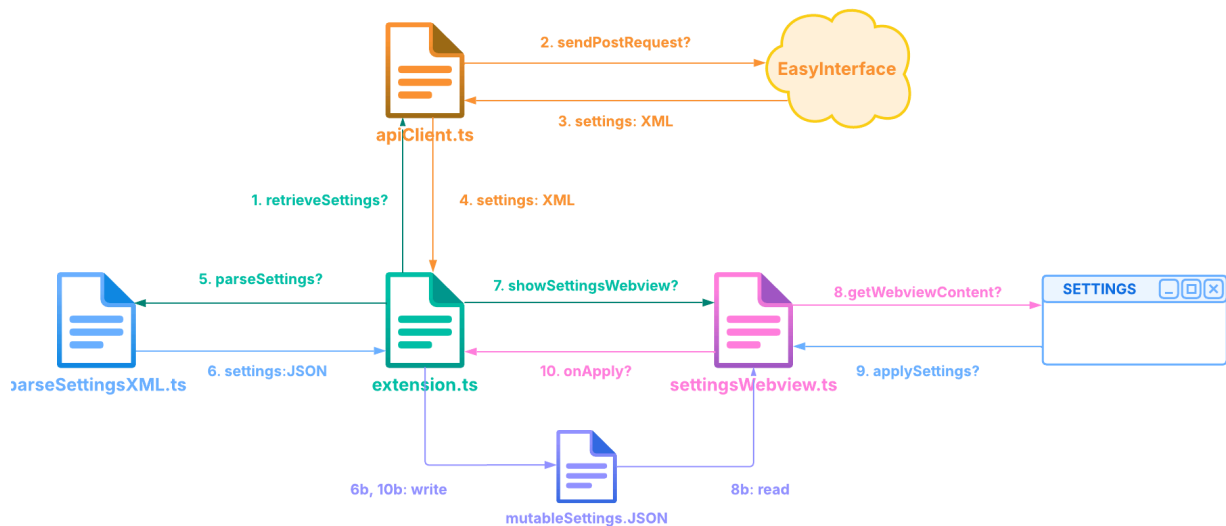


Figura 3.3.8: Esquema de flujo del funcionamiento de los ajustes.

Los directorios y archivos involucrados resultan de la siguiente manera en el proyecto:

```

.
├── ...
├── src
│   ├── interfaces                // interfaces y tipos de la extensión
│   │   ├── settings.ts          // estructura de los parámetros
│   │   └── common.ts            // definición de tipos comunes
│   ├── core                      // lógica principal y servicios
│   │   ├── apiClient.ts         // comunicación con EasyInterface
│   ├── parsers                   // parsers de respuestas de EasyInterface
│   │   ├── settingsParser.ts    // parser de los parámetros
│   ├── webviews                  // webviews HTML interfaz de usuario
│   │   ├── settingsWebview.ts  // renderiza y gestiona de ajustes
│   └── extension.ts              // código fuente de la extensión

```

3.4 Outline del código

Una de las funcionalidades que se requieren de un cliente de EasyInterface es un outline de código, con las funciones del archivo abierto y un selector múltiple para poder decidir qué funciones deben ser analizadas por la herramienta seleccionada. En el cliente web se muestra de la siguiente manera:

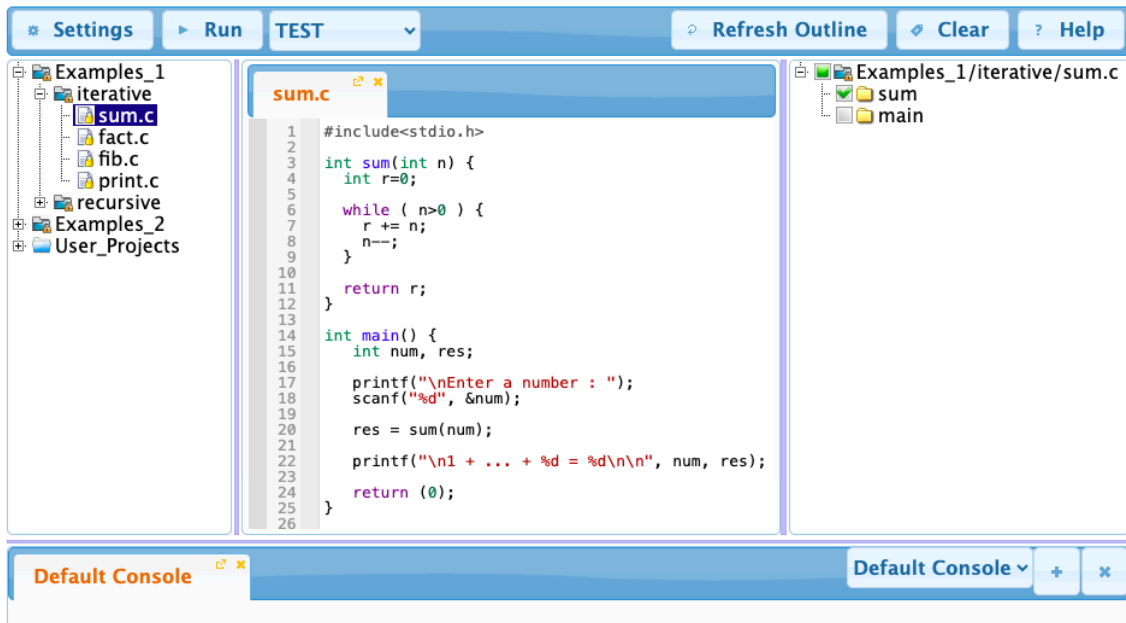


Figura 3.4.1: Outline en el cliente web de EasyInterface

Cuando el usuario hace clic en el botón “Refresh Outline”, el cliente web envía una petición a una herramienta de EasyInterface llamada “outline tool”, adjuntando los archivos de interés. Como en el caso de los parámetros, la salida es un XML con el contenido de la respuesta, que deberá ser procesada por un parser para generar una vista de árbol que muestre todos los métodos incluidos en el archivo.

Sin embargo, VSCode dispone de su propio sistema de generación de outline. Aunque dicho sistema no incorpora un selector múltiple, proporciona toda la información necesaria para visualizar los métodos y símbolos relevantes del archivo abierto. Por este motivo, en vez de enviar una petición a EasyInterface, generar un parser y procesar la respuesta XML, basta con solicitar a Visual Studio Code que ejecute el siguiente comando interno:

```
vscode.commands.executeCommand<vscode.DocumentSymbol[]>('vscode.executeDocumentSymbolProvider', uri: vscode.Uri):
Promise<vscode.DocumentSymbol[]>
```

Este comando consulta al proveedor de símbolos del documento actualmente abierto (identificado por su URI) y devuelve una promesa que, al resolverse, proporciona un array de objetos `vscode.DocumentSymbol`, cada uno representando un símbolo relevante del archivo. Así, la extensión puede obtener y mostrar la estructura del código de manera directa y eficiente, aprovechando las capacidades nativas de Visual Studio Code.

La representación del outline se realiza mediante un `OutlineTreeProvider`, que implementa la interfaz `vscode.TreeDataProvider<OutlineTreeItem>` y gestiona la estructura y actualización de la vista de árbol. Cada nodo del árbol (`OutlineTreeItem`) representa un símbolo del código (clase, función, etc) e incluye suficiente información para representarlo de una forma visualmente atractiva (etiqueta, tipo de símbolo, icono, etc).

Desde dicha clase, cada vez que el usuario cambia de editor o modifica el archivo activo, se invoca el comando interno

`vscode.executeDocumentSymbolProvider` previamente explicado y se transforma cada símbolo y sus hijos en nodos de la vista de árbol mediante el método privado `convertSymbol`.

En el controlador de la extensión `extension.ts` se registra la vista de árbol, permitiendo la selección múltiple de elementos. Dicha selección será fácilmente obtenida mediante `treeViewSelection` y posteriormente transformada al formato deseado por `EasyInterface`, explicado más adelante.

El outline del código en nuestra extensión resulta de la siguiente manera:

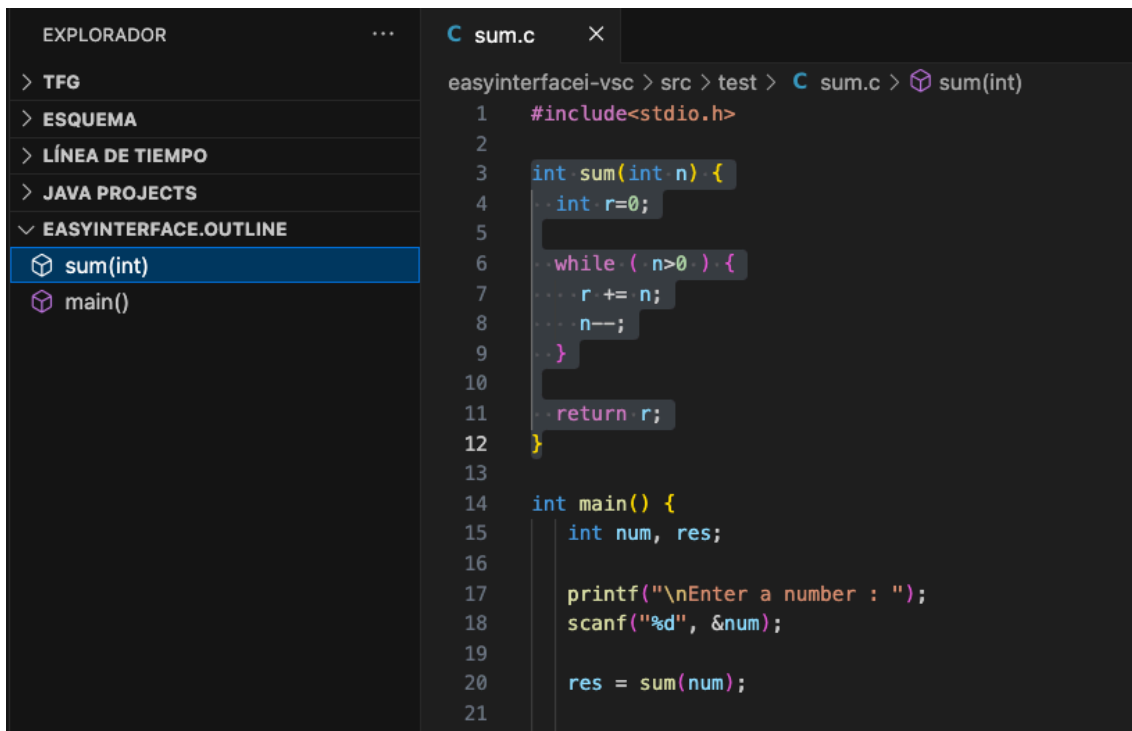


Figura 3.4.2: Outline en la extensión de Visual Studio Code.

3.5 Selector de aplicación

De manera similar al outline, la extensión debe mostrar la lista de aplicaciones disponibles para que el usuario pueda seleccionar una de ellas y realizar el análisis. Para ello, creamos una clase `appSelector` que implementa la interfaz `vscode.TreeDataProvider<AppItem>` y, al igual que en el caso del outline, gestiona la estructura y actualización de la vista de árbol.

El controlador `extension.ts` registra la vista de árbol, proporcionándole al constructor la lista de aplicaciones disponibles. Recordamos que hemos recibido esta lista previamente de `EasyInterface` tras la primera petición de ajustes. Desde `appSelector.ts` se convierte a cada aplicación en nodos de la vista de árbol, que contienen la etiqueta (o nombre de la aplicación) y el valor (o identificador único).

Cuando el usuario selecciona un nodo de la vista de árbol, se ejecuta el siguiente comando:

```
'easyinterface.selectApp', (appId: string) => void
```

El comando recibe el identificador de la aplicación como argumento y actualiza

su valor tanto en el estado global de la extensión como en el almacenamiento de estado de Visual Studio Code (`context.workspaceState`), lo que permite que la selección persista entre sesiones. En el estado global de la extensión, el identificador seleccionado se almacena en una variable, gestionada mediante las siguientes funciones:

```
function getSelectedAppId(): string | undefined {}
function setSelectedAppId(appId: string) {}
```

Estos métodos serán utilizados posteriormente para enviar el fichero a EasyInterface y obtener el identificador de la herramienta que debe ejecutarse.

La vista de aplicaciones queda de la siguiente manera:

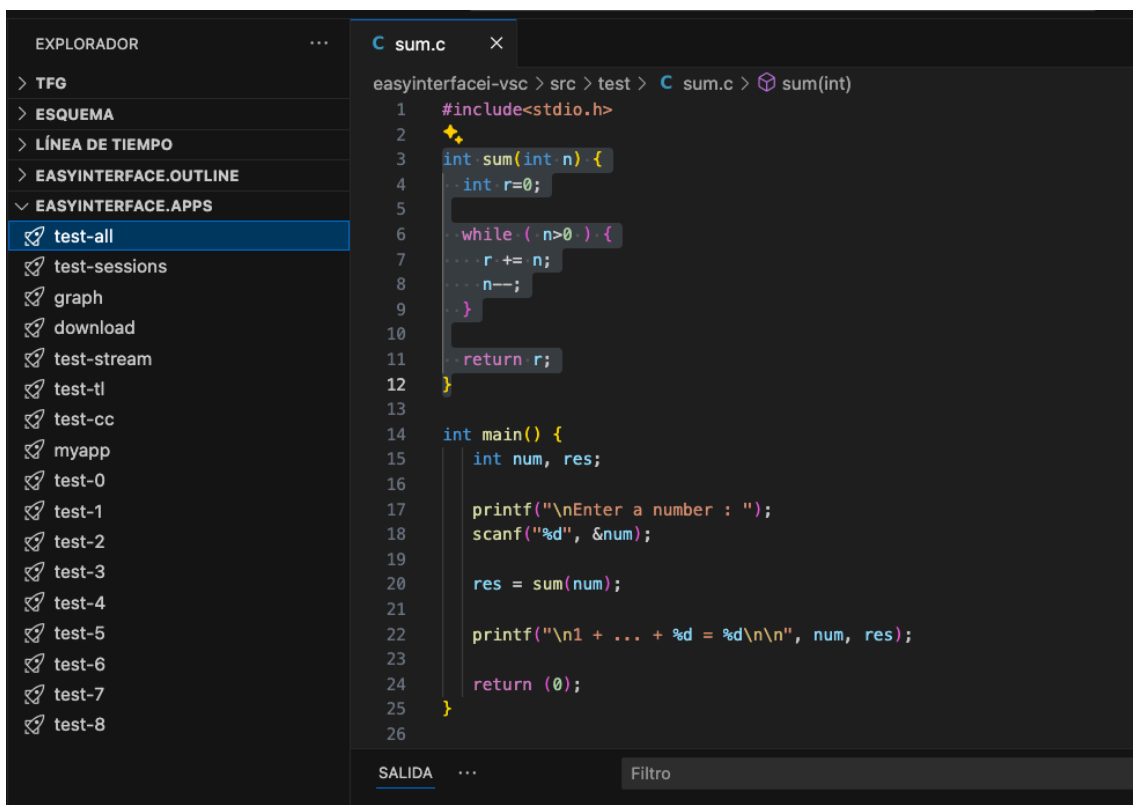
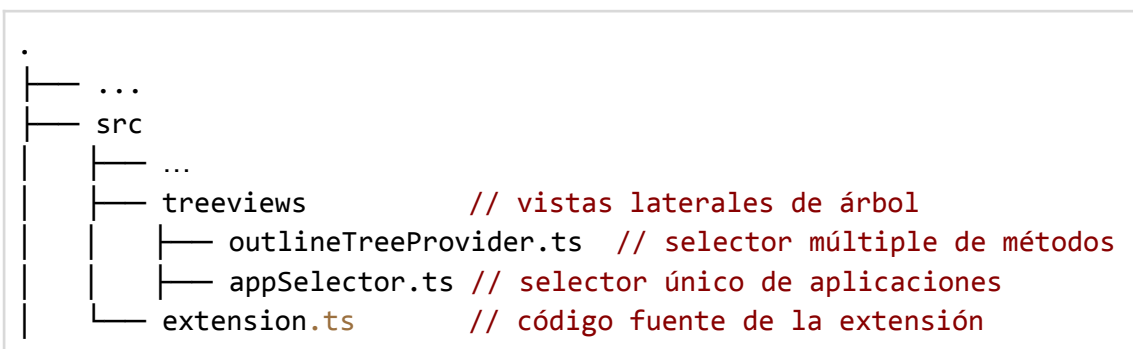


Figura 3.5.1: Selector de aplicación en la vista lateral de VSCode.

La estructura de las clases involucradas en la creación del selector múltiple de métodos y el selector de aplicaciones queda de la siguiente manera:



3.6 Envío a EasyInterface

Para enviar un archivo a EasyInterface de manera que se ejecute una aplicación, el usuario debe hacer clic en el botón “EasyInterface: SendFileToAPI”. Esta acción activa un comando registrado como un evento de activación y punto de contribución en `package.json` y en `extension.ts`, que en primer lugar, obtiene el contenido del editor activo mediante la función `getActiveEditorContent`, devolviendo un error si no encuentra un editor, así como la ruta absoluta del fichero abierto.

Asimismo, utilizando la vista de árbol previamente descrita en la sección 3.4, obtiene los nombres de las funciones seleccionadas y, en caso de que existan, las pone en el formato aceptado por EasyInterface. De la misma manera, obtiene el identificador de la herramienta que el usuario ha seleccionado.

Para enviar los parámetros modificados por el usuario, la extensión obtiene los parámetros de la aplicación seleccionada haciendo uso del archivo mutable. Con estos datos, construye el objeto `Parameters` utilizando los valores más recientes proporcionados por el usuario o, en caso de que no se hayan modificado, los valores por defecto. Este objeto toma la siguiente forma:

```
export interface Parameters {
    settings?: Settings; // array of name and value of each setting
    _ei_files: EIFile[]; // array of files to be processed
    _ei_clientid?: string;
    _ei_outformat: "eiol" | "txt";
    _ei_outline?: EIOutline;
}
```

A continuación se ejecuta la función `normalizeParameters` para convertir cada parámetro al formato requerido por EasyInterface. Finalmente, se construye un objeto `EIRequest` con el comando “execute”, el identificador de la aplicación seleccionada y los parámetros descritos anteriormente.

```
export interface EIRequest {
    command: "execute" | "app_info" | "app_parameters" |
    "app_details";
    app_id: string;
    parameters?: Parameters;
}
```

De la misma manera a la descrita en la sección 3.2, se hace uso de la función `sendPostRequest` de `apiClient.ts` para enviar la petición (`EIRequest`) a EasyInterface y recoger la respuesta en formato XML. Todo este proceso se recoge de forma visual en el siguiente diagrama de flujo:

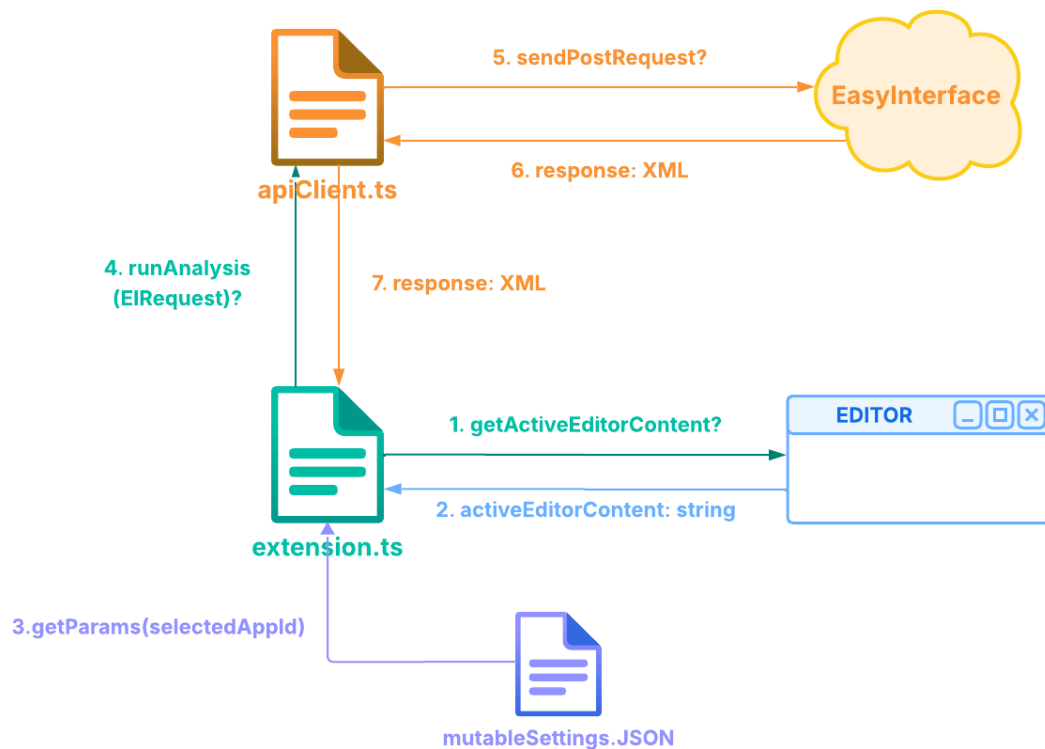


Figura 3.6.1: Diagrama de flujo al enviar el análisis a EasyInterface.

3.7 Recepción de la respuesta y salida en pantalla

El primer paso después de recibir la respuesta XML es, al igual que hicimos con el XML de parámetros, construir un parser y obtener los objetos relevantes para poder mostrar los resultados del análisis en pantalla. Según el manual del servidor, la respuesta toma la siguiente forma:

```

<eiout>
  <eicommands>
    [EICOMMAND]*
  </eicommands>
  <eiactions>
    [EIACTION]*
  </eiactions>
</eiout>

```

Figura 3.7.1: Estructura de la respuesta de EasyInterface.

Donde eiout es la etiqueta exterior que encapsula los comandos y acciones, EICOMMAND es la lista de comandos a ejecutar y EIACTION la lista de acciones a declarar. Aquí pues, distinguimos entre ambos conceptos: comando y acción. Un comando se ejecuta inmediatamente después de realizar el análisis, y puede ser imprimir un texto en consola, resaltar las líneas 5-10, crear un marcador, etc. Por otro lado, una acción es un comando o una lista de comandos a ser ejecutada después de una interacción del usuario, por

ejemplo, al hacer clic en la línea 10, resaltar las líneas 12-13 y mostrar un marcador en la línea 5.

Los comandos toman la siguiente estructura:

EICOMMANDS

```
<eicommans dest=[PATH]? outclass=[OUTCLASS]? >  
  [EICOMMAND]*  
</eicommans>
```

Figura 3.7.2: Estructura interna de un comando.

El parámetro dest es la ruta absoluta del archivo destino donde se aplica el comando, y outclass especifica el tipo de información que proporcionan los comandos (error, información, aviso, etc).

Para esta extensión, hemos implementado los siguientes tipos de comandos:

PRINTONCONSOLECOMMAND

```
<printonconsole outclass=[OUTCLASS]? consoleid=[CONSOLEID]?  
  consoletitle=[STRING]?>  
  [CONTENT]+  
</printonconsole>
```

Figura 3.7.3: Estructura de PRINTONCONSOLECOMMAND.

HIGHLIGHTLINESCOMMAND

```
<highlightlines outclass=[OUTCLASS]? dest=[PATH]?>  
  [LINES]*  
</highlightlines>
```

Figura 3.7.4: Estructura de HIGHLIGHTLINESCOMMAND.

DIALOGBOXCOMMAND

```
<dialogbox outclass=[OUTCLASS]? boxtitle=[STRING]? boxwidth=[INT]?  
  boxheight=[INT]?>  
  [CONTENT]+  
</dialogbox>
```

Figura 3.7.5: Estructura de DIALOGBOXCOMMAND.

ADDMARKERCOMMAND

```
<addmarker outclass=[OUTCLASS]? dest=[PATH]? boxtitle=[STRING]?  
          boxwidth=[INT]? boxheight=[INT]?>  
  [LINES]  
  [CONTENT]*  
</addmarker>
```

Figura 3.7.6: Estructura de ADDMARKERCOMMAND.

ADDINLINEMARKERCOMMAND

```
<addinlinemarker outclass=[OUTCLASS]? dest=[PATH]?>  
  [LINES]  
  [CONTENT]*  
</addinlinemarker>
```

Figura 3.7.7: Estructura de ADDINLINEMARKERCOMMAND.

La modularidad que ha sido tomada en cuenta para implementar cada comando permite de manera sencilla añadir más tipos de comandos en un futuro.

Por otro lado, las acciones toman la siguiente estructura:

EIACTIONS

```
<eiactions dest=[PATH]? autoclean=[BOOL]?>  
  [EIACTION]*  
</eiactions>
```

Figura 3.7.8: Estructura interna de la lista de comandos.

La única acción implementada en nuestra extensión es la siguiente:

ONCODELINECLICKACTION

```
<oncodelineclick dest=[PATH]? autoclean=[BOOL]? outclass=[OUTCLASS]?>  
  [LINES]  
  [CONTENT]*  
  [EICOMMANDS]  
</oncodelineclick>
```

Figura 3.7.9: Estructura de ONCODELINECLICKACTION.

Si se desea obtener información más detallada sobre cada campo, se recomienda consultar el manual de EasyInterface presente en la bibliografía, ya que en este apartado solo es pertinente ofrecer una visión general de la estructura.

Con el objetivo de realizar el parser, se ha creado una interfaz para cada sección descrita con todos sus campos en los archivos `actions.ts`, `commands.ts`, y `common.ts`. En este último se describen tipos comunes de

contenido que han sido utilizados previamente. Además, el parser se ha descrito en el archivo `parser/analysisXMLParser.ts` y probado por el archivo `test/testAnalysisParser.ts`.

El diagrama final de cómo resultan las interfaces y sus dependencias es el siguiente:

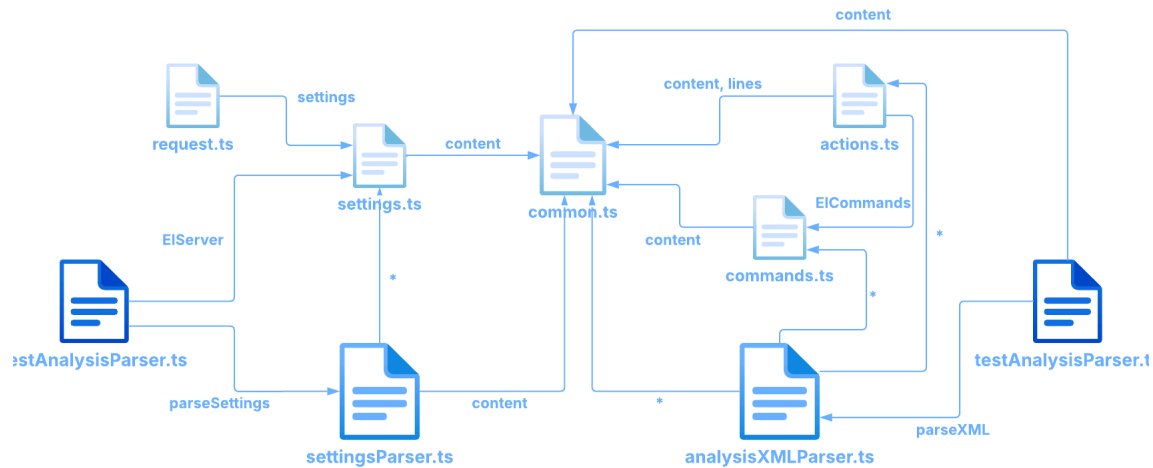


Figura 3.7.10: Dependencias e importaciones entre interfaces y parsers.

Al ejecutar el parser se devuelven dos objetos: `EICommands` y `EIActions`. Estos objetos corresponden a la lista de comandos a ejecutar y acciones a registrar respectivamente. Para ejecutarlas y mostrar los resultados del análisis, hemos creado un registro de comandos en `core/commandRegistry.ts`, que registra un mapa denominado `commandHandler`:

```
commandHandlers = new Map<string, CommandHandler>();
```

Al iniciar la extensión, incluso antes de enviar el archivo a analizar, se inicializan todos los comandos en dicho mapa, con el objeto del comando como clave y el manejador como valor. Un manejador es un objeto o función que en este caso representa la función a ejecutar.

Dada la lista de comandos, esta se recorre mediante un `for/each`, obteniendo del mapa descrito su manejador y ejecutando el comando correspondiente (o error si no está registrado). Para cada tipo de comando, su manejador y función de inicialización se encuentran en el directorio `commands/nameOfTheCommand.ts`. La función de inicialización registra el comando en el mapa `commandHandlers`, mientras que cada función de manejador ejecuta la lógica correspondiente al comando: imprimir en consola, mostrar una caja de texto, resaltar líneas, etc.

De esta manera, en caso de querer añadir un nuevo comando, la serie de pasos a seguir sería:

1. `interfaces/commands.ts`
 - a. Añadir nuevo tipo de objeto `EICommand`.
 - b. Crear la nueva interfaz del nuevo comando con los parámetros necesarios.
2. `parser/analysisXMLParser.ts`

- a. Crear nueva función de parser del comando: `function parseNewCommand`.
 - b. En la función parser de los comandos `parseCommands`, añadir el caso de que el nombre del comando sea el mismo del nuevo comando y añadir ese objeto a la lista de comandos resultante.
3. `commands/newCommand.ts`
 - a. Definir el manejador con la lógica correspondiente: `handler NewCommand (command: NewCommand)`.
 - b. Definir la función de inicialización: `registerCommand` en el mapa con el objeto del nuevo comando que acabamos de definir en la interfaz.
4. `extension.ts`
 - a. Importar la función de inicialización del comando y añadirla a la función de inicialización de todas las funciones.

La misma lógica se aplica para registrar las acciones en `core/actionRegistry.ts`. Sin embargo, en el caso de las acciones interactivas hemos de registrar cuáles han sido ya ejecutadas para poder revertir sus efectos si se vuelven a ejecutar. Es decir, al hacer un clic en una acción, debe mostrar los comandos pertinentes; pero si vuelve a hacer clic en la misma línea, la extensión debe deshacer sus efectos.

Para implementar este comportamiento creamos un mapa de las acciones ejecutadas en `onCodeLineClick.ts`:

```
const executedActions = new Map<string, Set<number>>>();
```

En este mapa, la clave es la URI del editor abierto y el valor es el conjunto de líneas en las que se han ejecutado acciones y aún están activas. De esta manera, la extensión puede identificar si una línea ya ha sido procesada y decidir si debe ejecutar o revertir los efectos asociados.

Además, para mejorar la experiencia de usuario, la extensión resalta visualmente las líneas interactivas mediante una decoración específica en el editor, facilitando la identificación de las líneas que tienen acciones asociadas y su estado actual. La gestión de estas decoraciones se actualiza dinámicamente según el estado de las acciones, asegurando que la interfaz refleje siempre la situación real de cada línea de código.

4 Manual de usuario

Para ejecutar la extensión partiendo del repositorio, debemos clonar el repositorio en nuestro directorio local y abrirlo desde Visual Studio Code. Abrimos el archivo `src/extension.ts` y ejecutamos el siguiente comando:

```
npm run compile
```

Este es un script personalizado definido en `package.json` que utiliza el compilador de TypeScript y el archivo de configuración `tsconfig.json` para compilar todos los archivos a JavaScript y que VSCode pueda ejecutarlo. Esto se hace ejecutando el comando:

```
tsc -p ./.
```

Después de compilar la extensión, presionamos F5 o seleccionamos “Debug: Start Debugging” y el host de extensiones de Visual Studio Code.

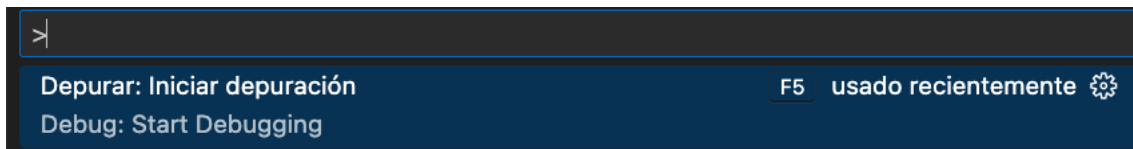


Figura 4.1: Comando a utilizar para probar la extensión.

Esto abrirá una nueva ventana llamada “Extension Development Host” o “Host de desarrollo de la extensión” donde podremos usar nuestra extensión como si estuviera instalada. Desde la consola de depuración de nuestro editor original podremos encontrar los logs de ejecución.

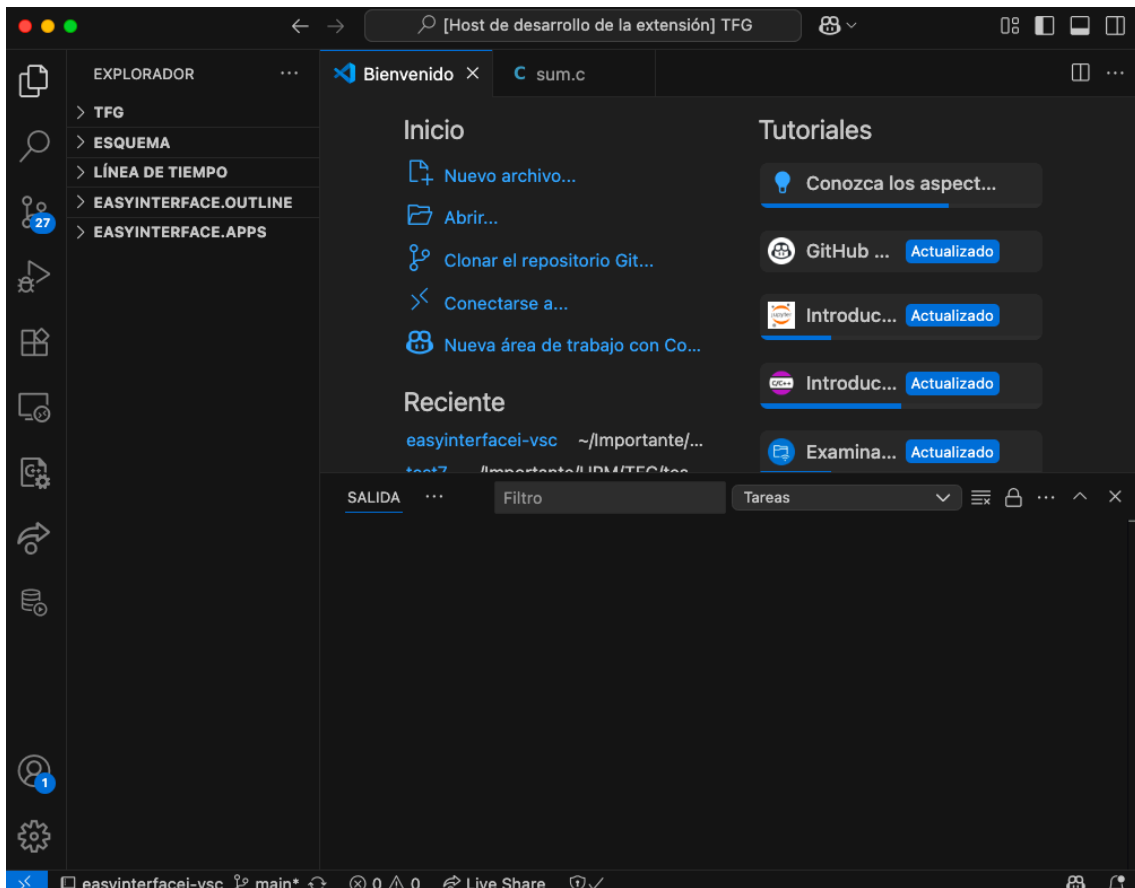


Figura 4.2: Snapshot de la extensión recién ejecutada.

Como se puede observar, en el explorador nos sale la carpeta abierta actualmente (en nuestro caso TFG), el esquema y línea de tiempo (nativos de VSCode) y dos menús de EasyInterface: Outline y Apps. Para esta demostración usaremos el archivo `src/test/sum.c`, extraído de los archivos de ejemplo de EasyInterface. Abrimos el archivo y vemos lo siguiente:

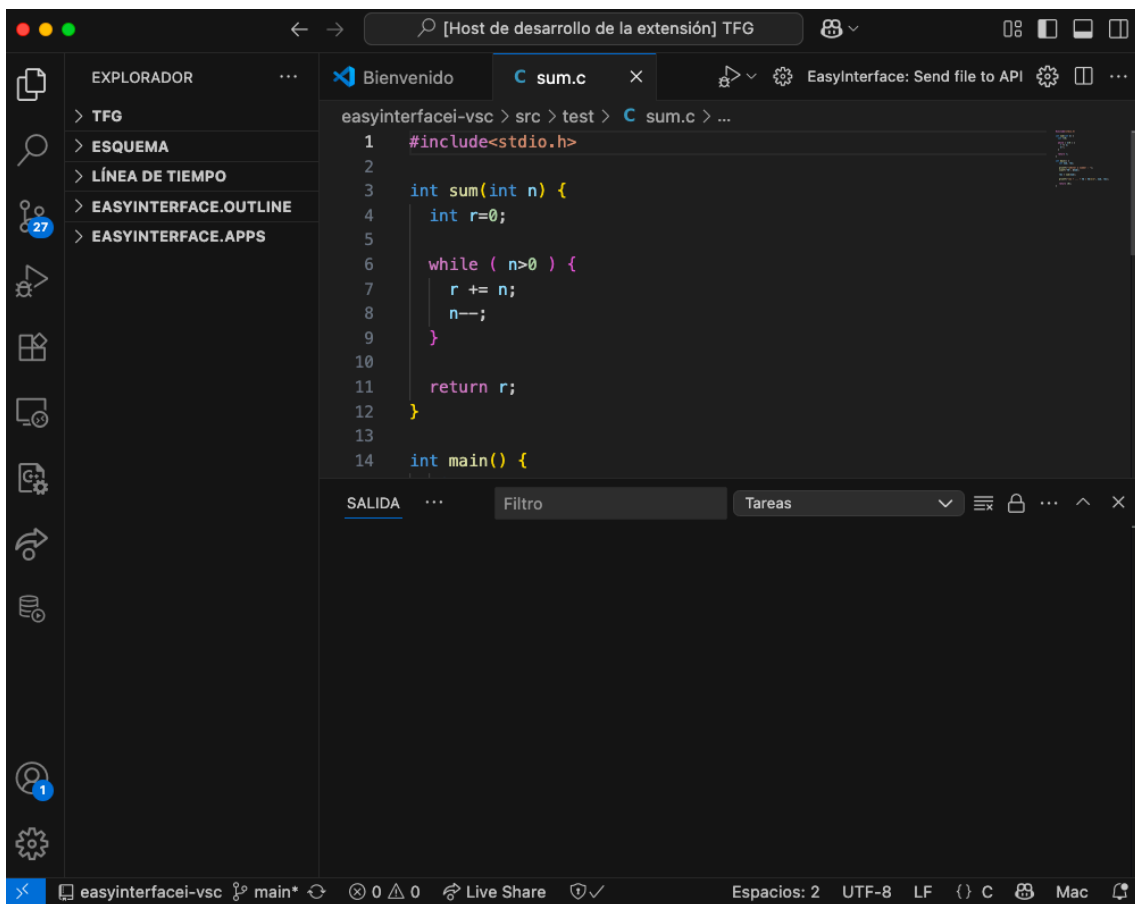


Figura 4.3: Extensión al abrir un archivo.

Si hacemos clic en la vista EasyInterface.Outline veremos el outline de las funciones del archivo de la siguiente manera:

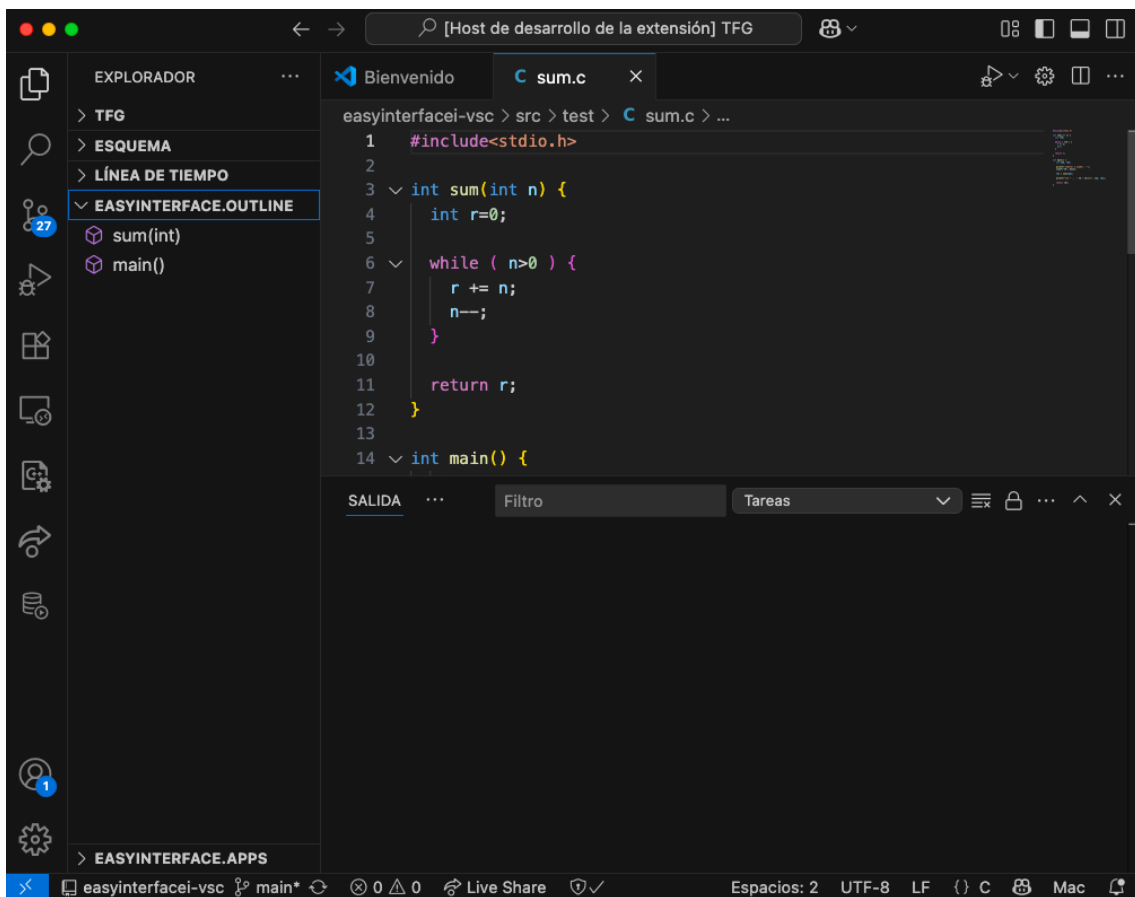


Figura 4.4: Outline de funciones y métodos de un archivo.

Y si hacemos clic en la vista EasyInterface.Apps veremos las aplicaciones de EasyInterface disponibles para enviar el análisis.

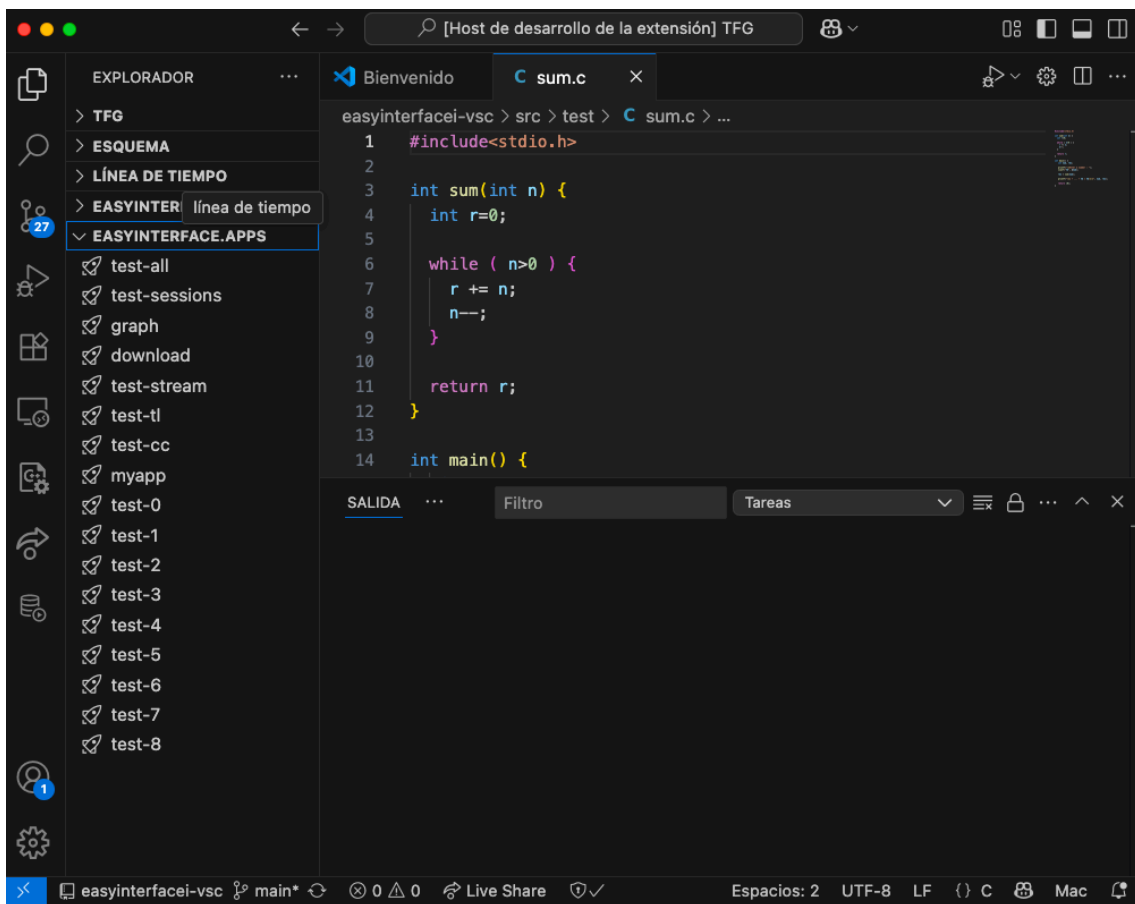


Figura 4.5: Listado de aplicaciones disponibles.

Si volvemos a hacer clic en el editor de sum.c, encontraremos los iconos de las funcionalidades de EasyInterface: ajustes y enviar al servidor.

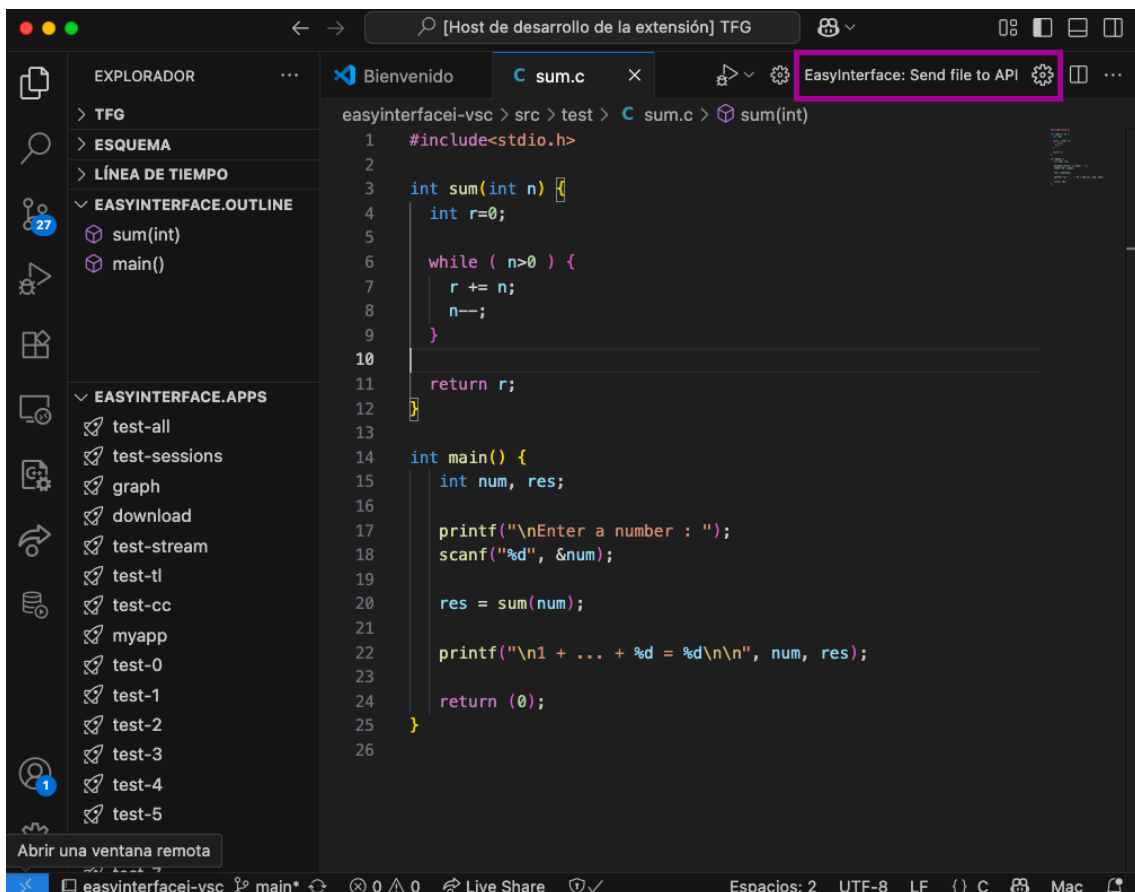


Figura 4.6: Iconos de EasyInterface en nuestra extensión.

Para acceder a los ajustes de cada aplicación de EasyInterface hacemos clic en la rueda de la derecha, que muestra el texto EasyInterface Settings cuando pasamos el ratón por ella.

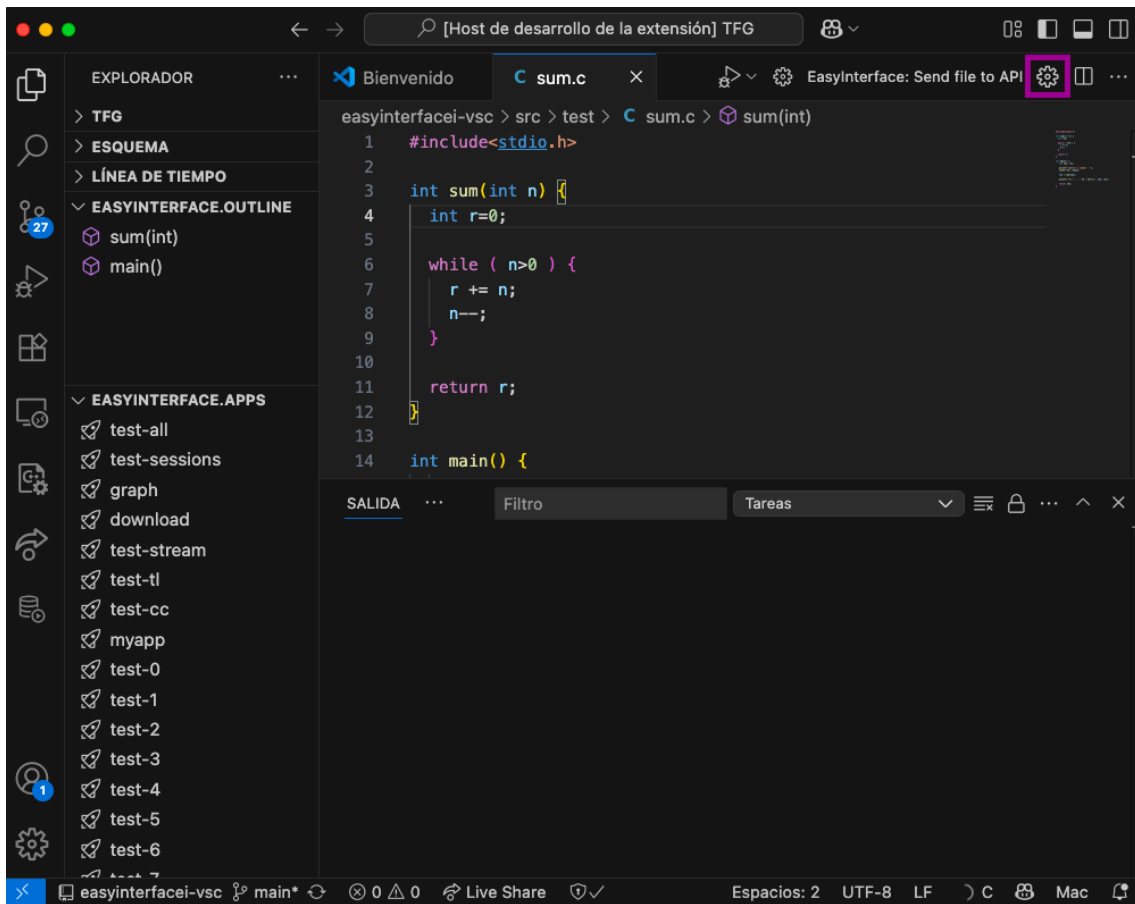


Figura 4.7: Botón de ajustes de EasyInterface.

Una ventana nueva se abre y muestra la siguiente información:

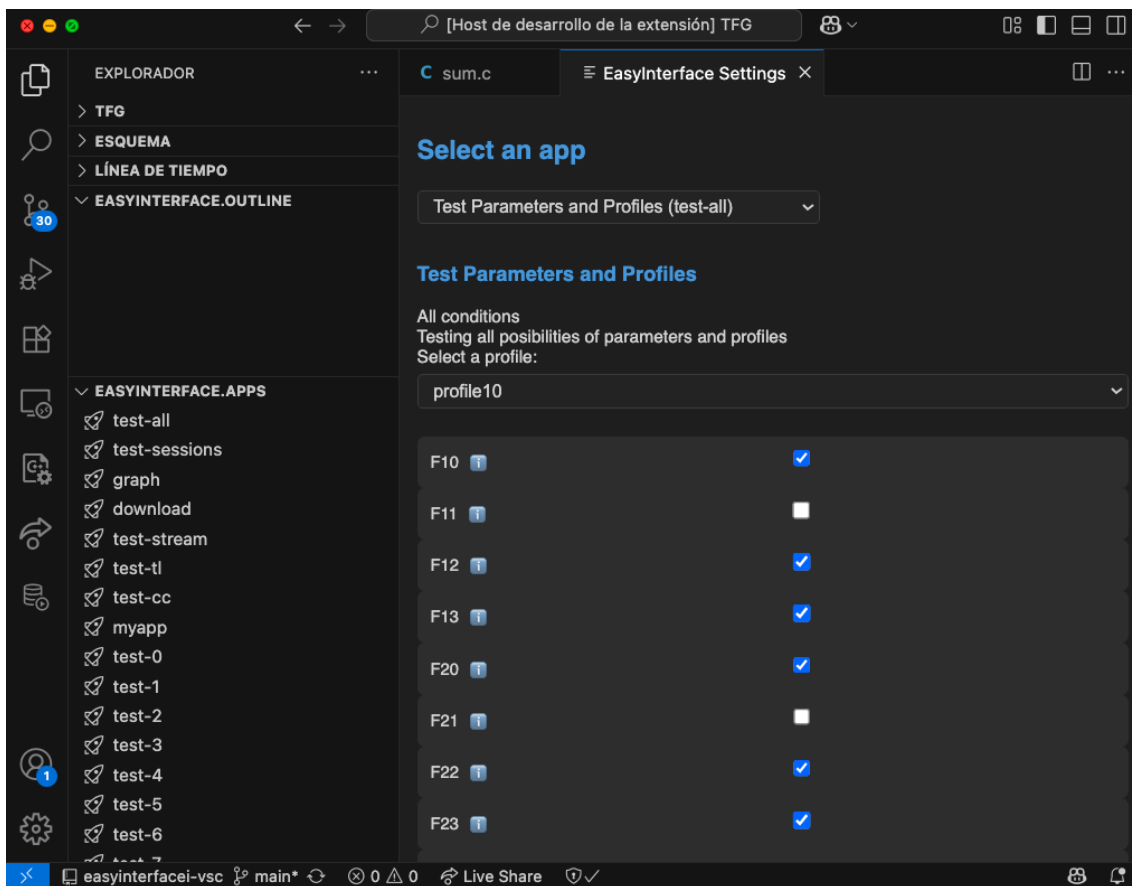


Figura 4.8: Pestaña EasyInterface Settings.

Si hacemos clic en el nombre de la primera app disponible, en este caso “Test Parameters and Profiles (test all)” podemos ver las demás aplicaciones y cambiar:

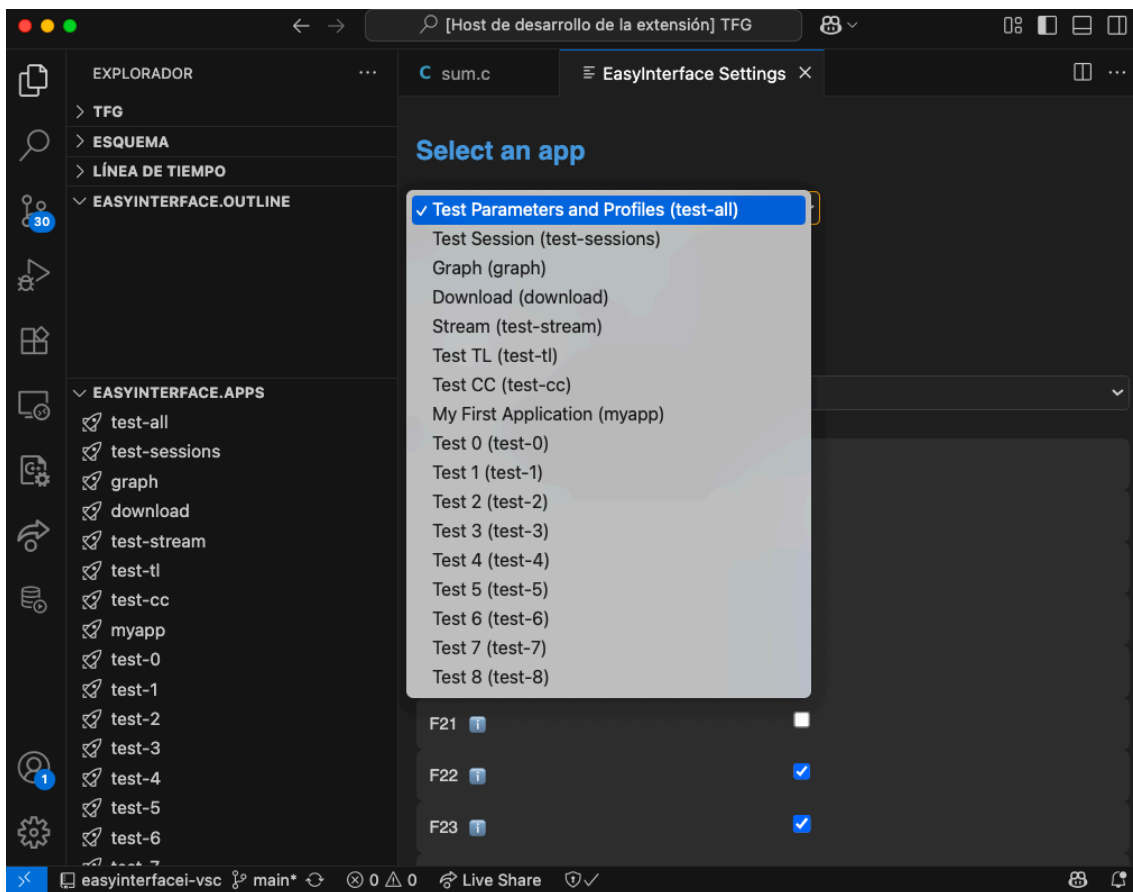


Figura 4.9: Desplegable con aplicaciones disponibles.
Cambiamos de aplicación y escogemos “My first application”.

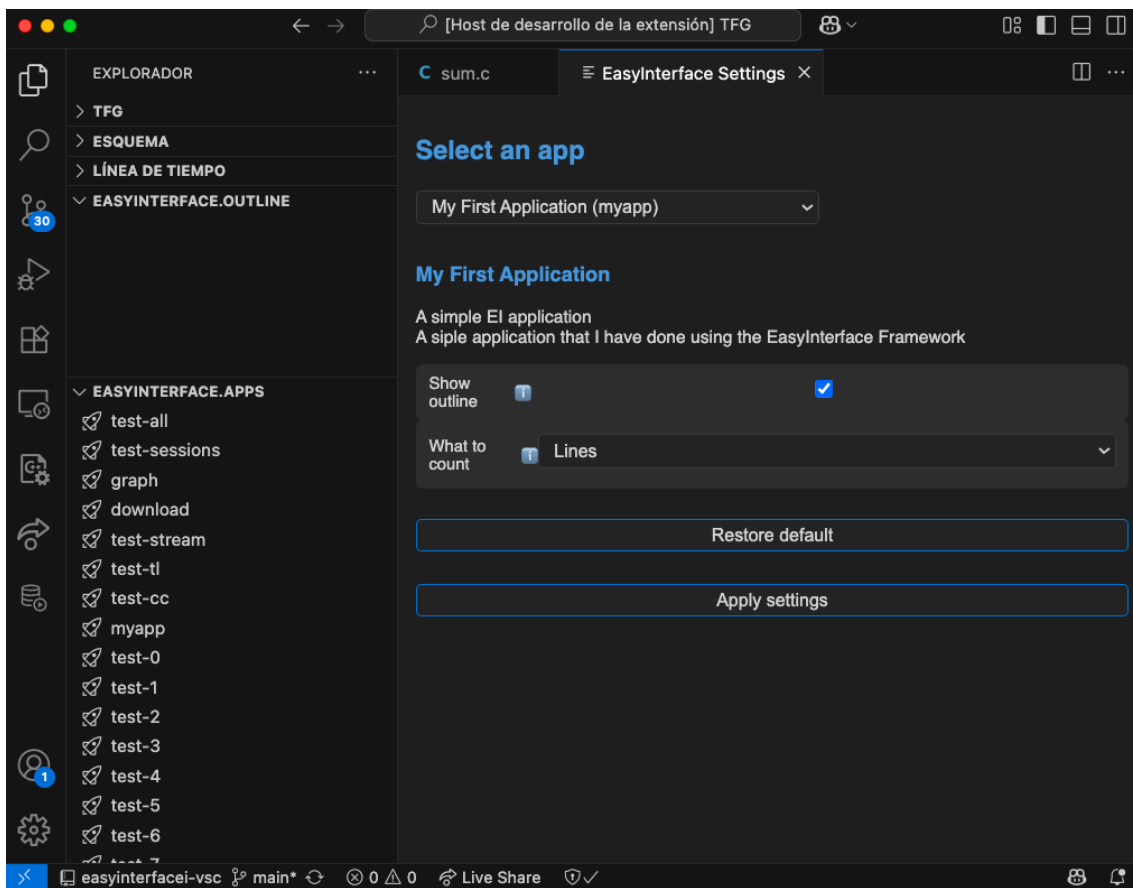


Figura 4.10: Información disponible de My First Application (my app).

Asimismo, si seleccionamos una aplicación sin parámetros se muestra de la siguiente manera, junto con las descripciones proporcionadas:

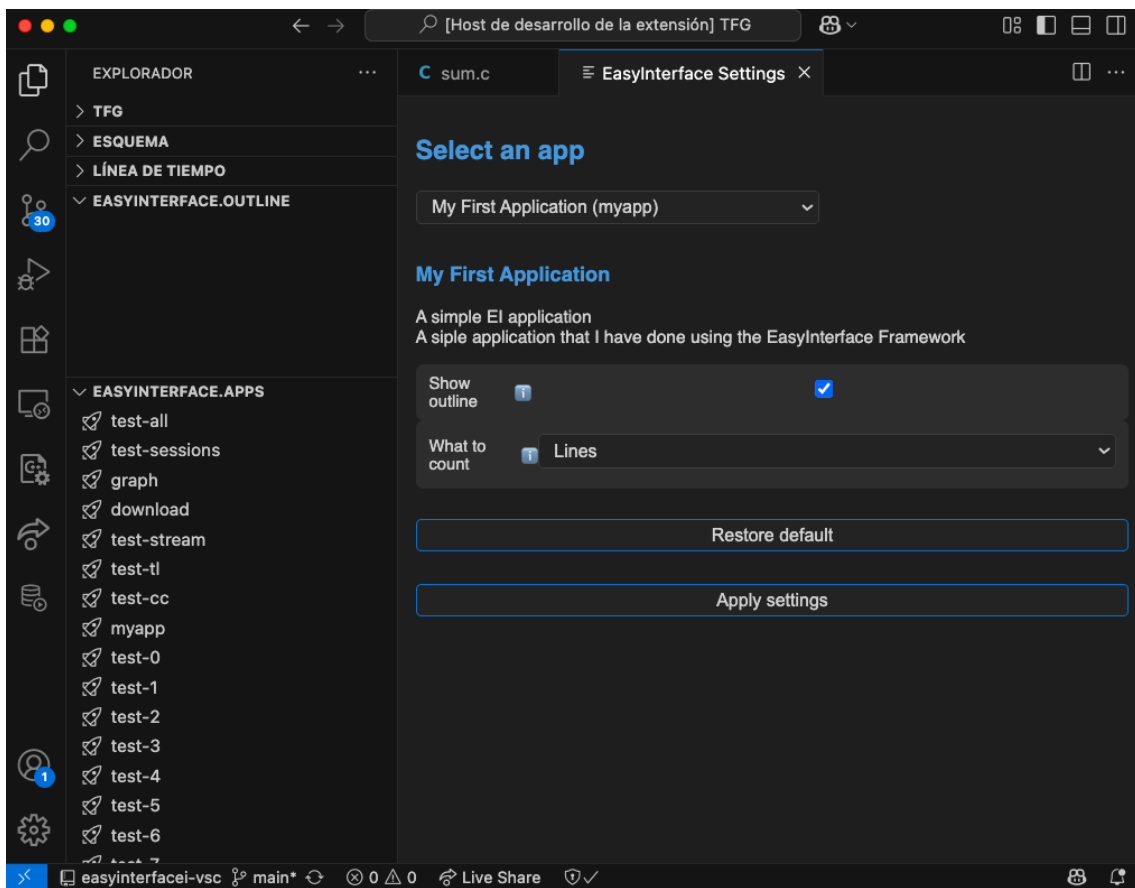


Figura 4.11: Información disponible de una aplicación sin parámetros (test-0). Volviendo a la primera aplicación, ya que podemos observar un mayor número y tipos de parámetros, nos desplazamos hacia abajo:

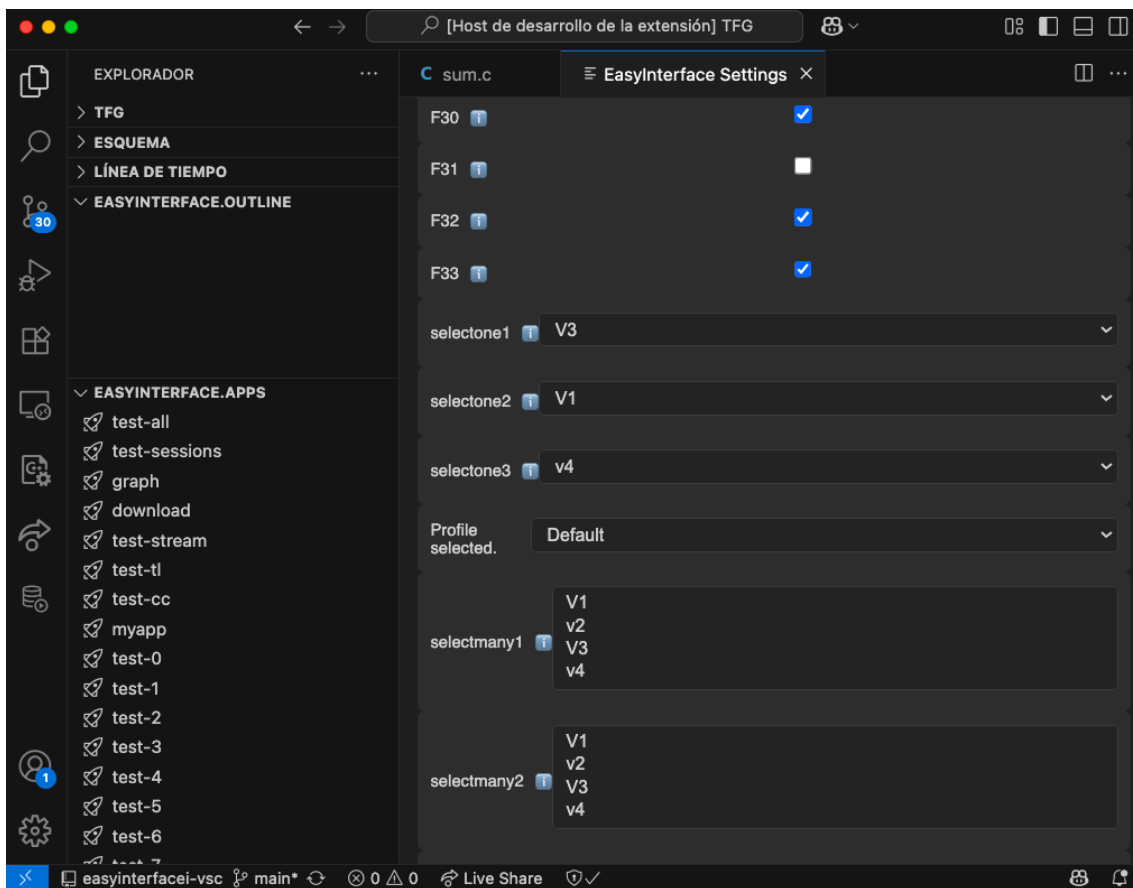


Figura 4.12: Información disponible de la primera aplicación.

Si pasamos el ratón por el botón de información, se muestra la descripción del parámetro (si existe). Asimismo, después de todos los ajustes encontramos dos botones: “Restore default” y “Apply Settings”.

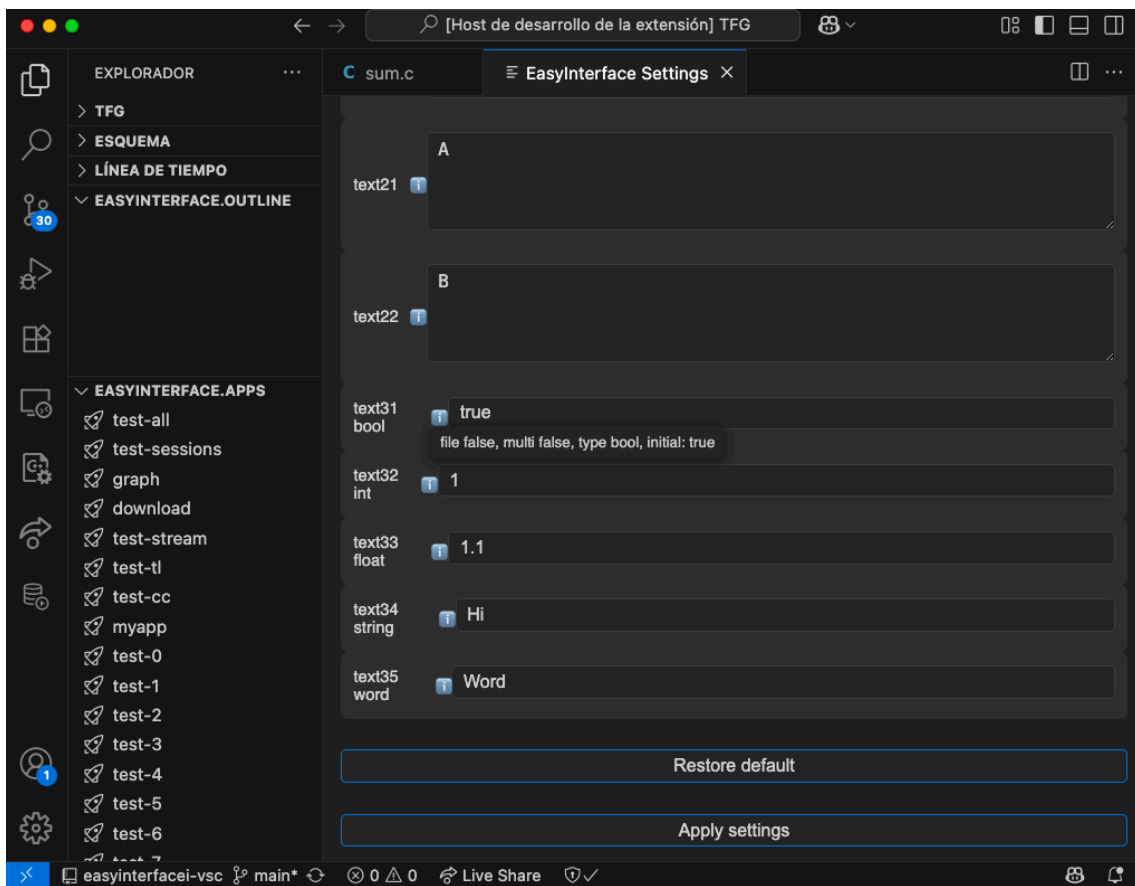


Figura 4.13: Cursor encima de un icono de información.

Si cambiamos de perfil, los valores por defecto o el texto inicial cambiarán, como por ejemplo, para el perfil 12, los campos de texto son:

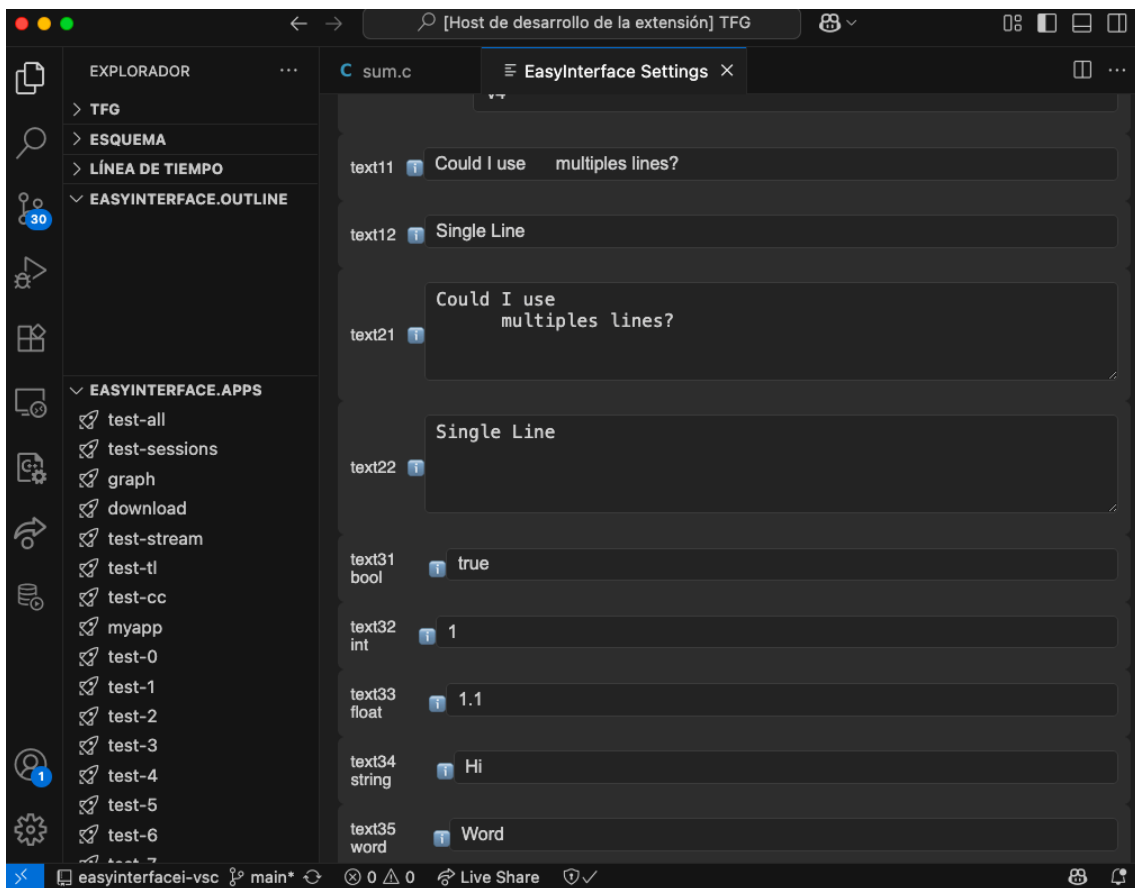


Figura 4.14: Campos de texto del perfil 12, cambian los valores por defecto.

Cambiamos los valores de la forma que veamos oportuna. En nuestro caso, para facilitar la demostración quitamos la selección de la mayoría de checkboxes menos F10, en los selectores escogemos V1, seleccionamos todo en el primer selector múltiple y cambiamos los campos de texto de la siguiente manera:

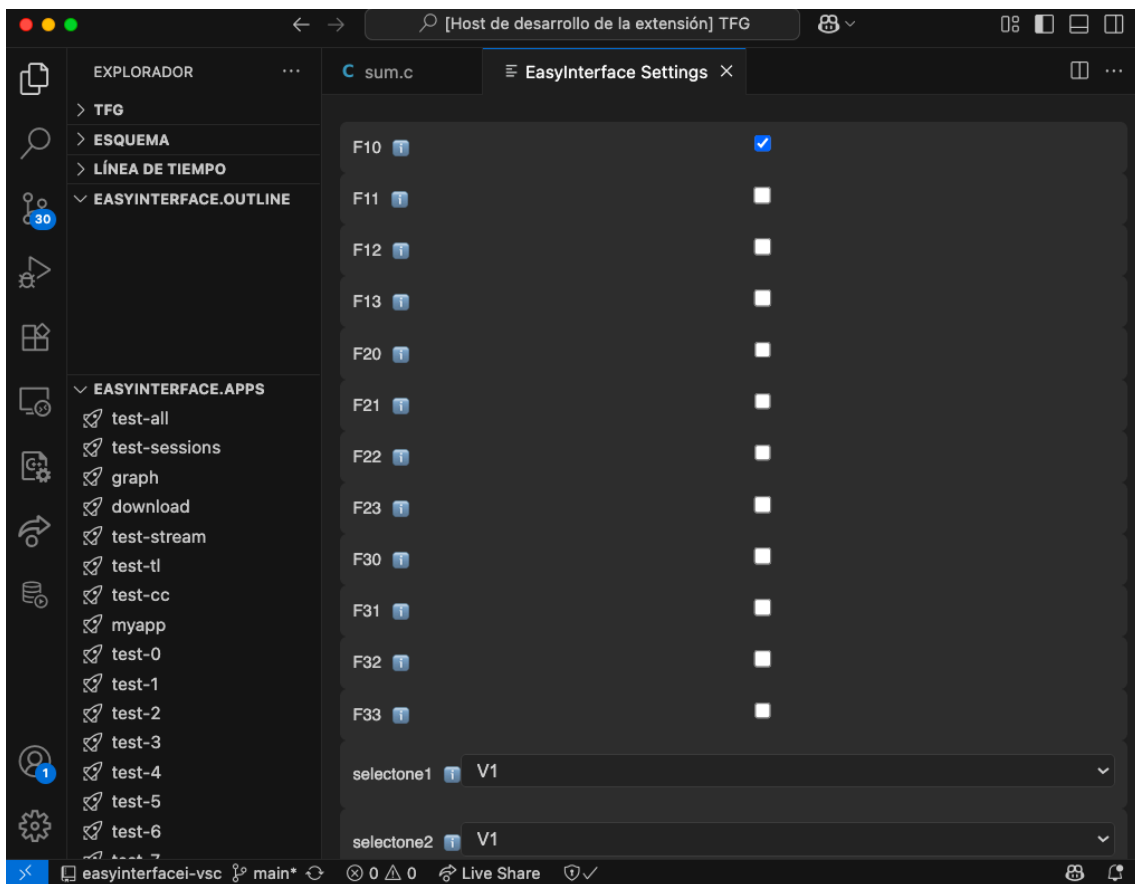


Figura 4.15: Campos modificados de ajustes de la primera aplicación.

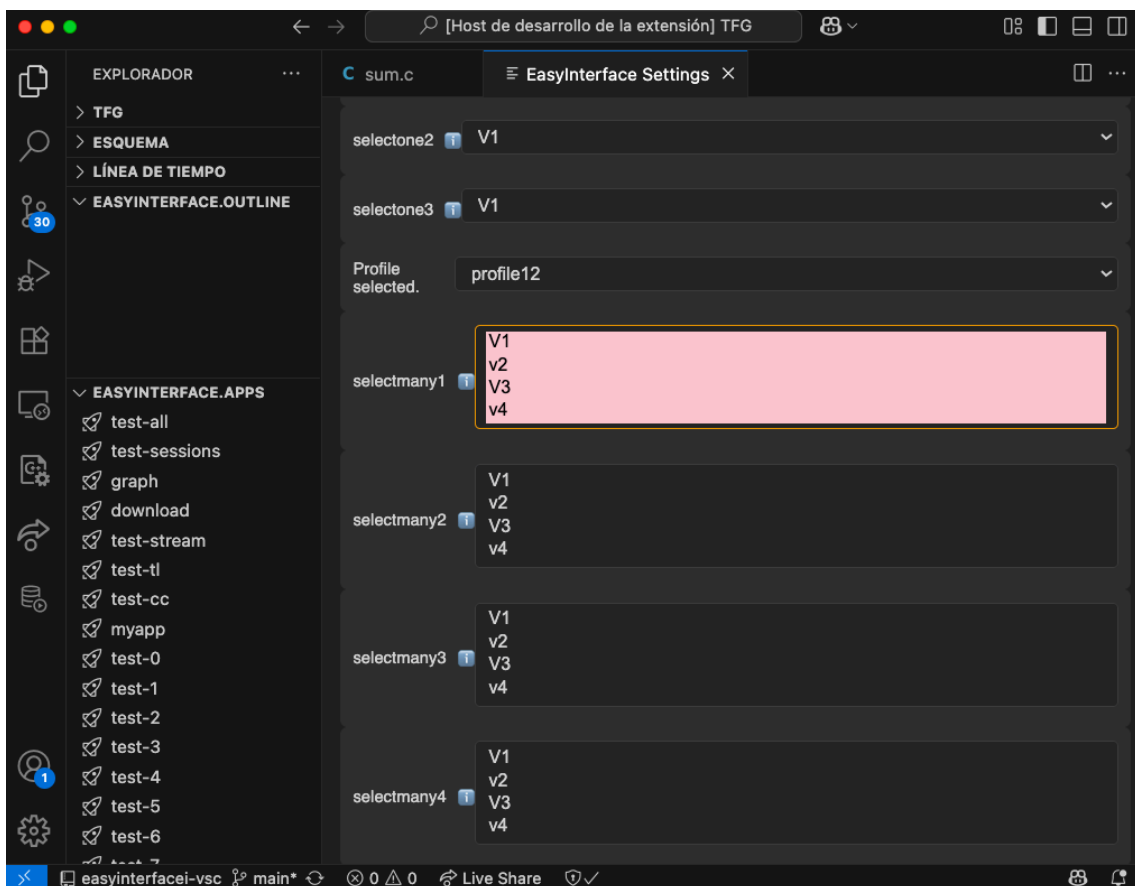


Figura 4.16: Campos modificados de ajustes de la primera aplicación.

Por último, hacemos clic en el botón de “ApplySettings”, como se muestra a continuación:

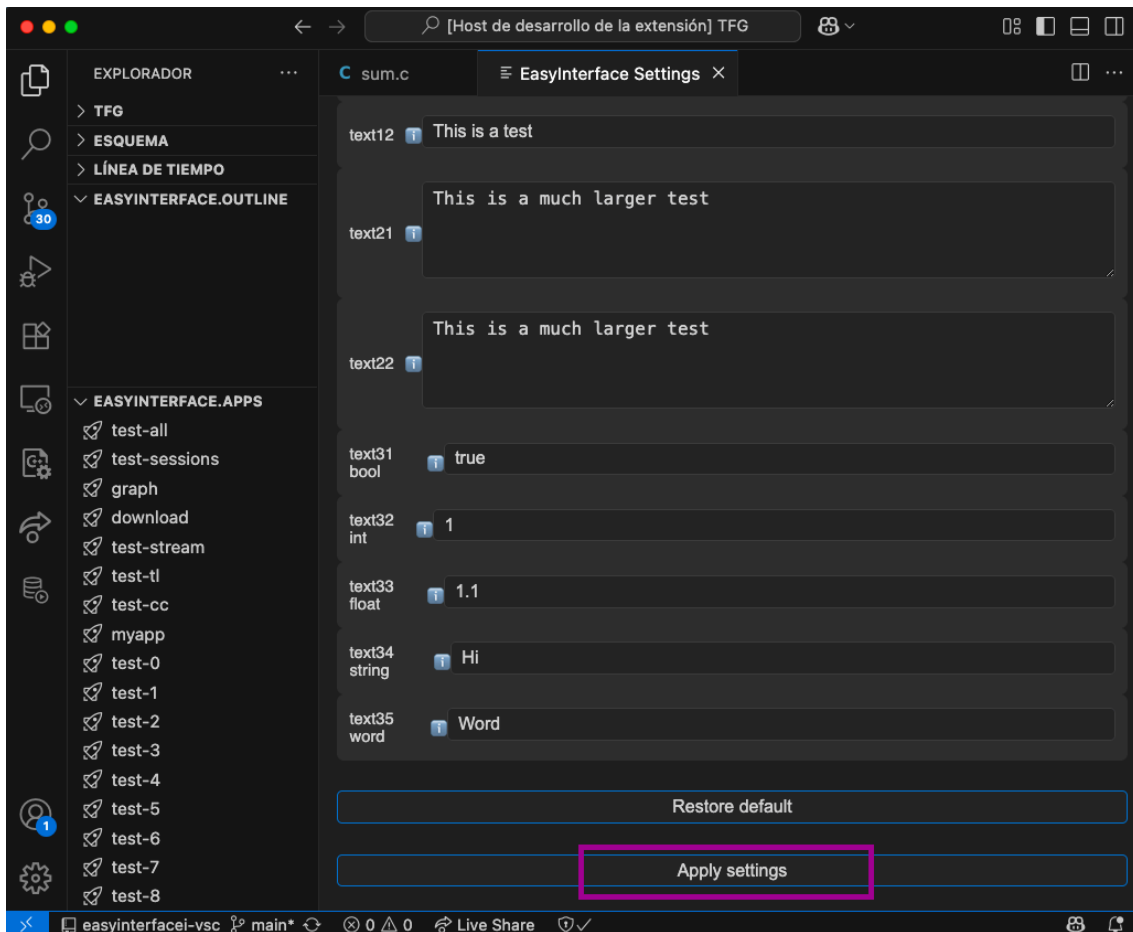


Figura 4.17: Campos modificados de la primera aplicación y botón “Apply”.

Al hacer clic, se muestra el siguiente mensaje y caja de diálogo de confirmación.

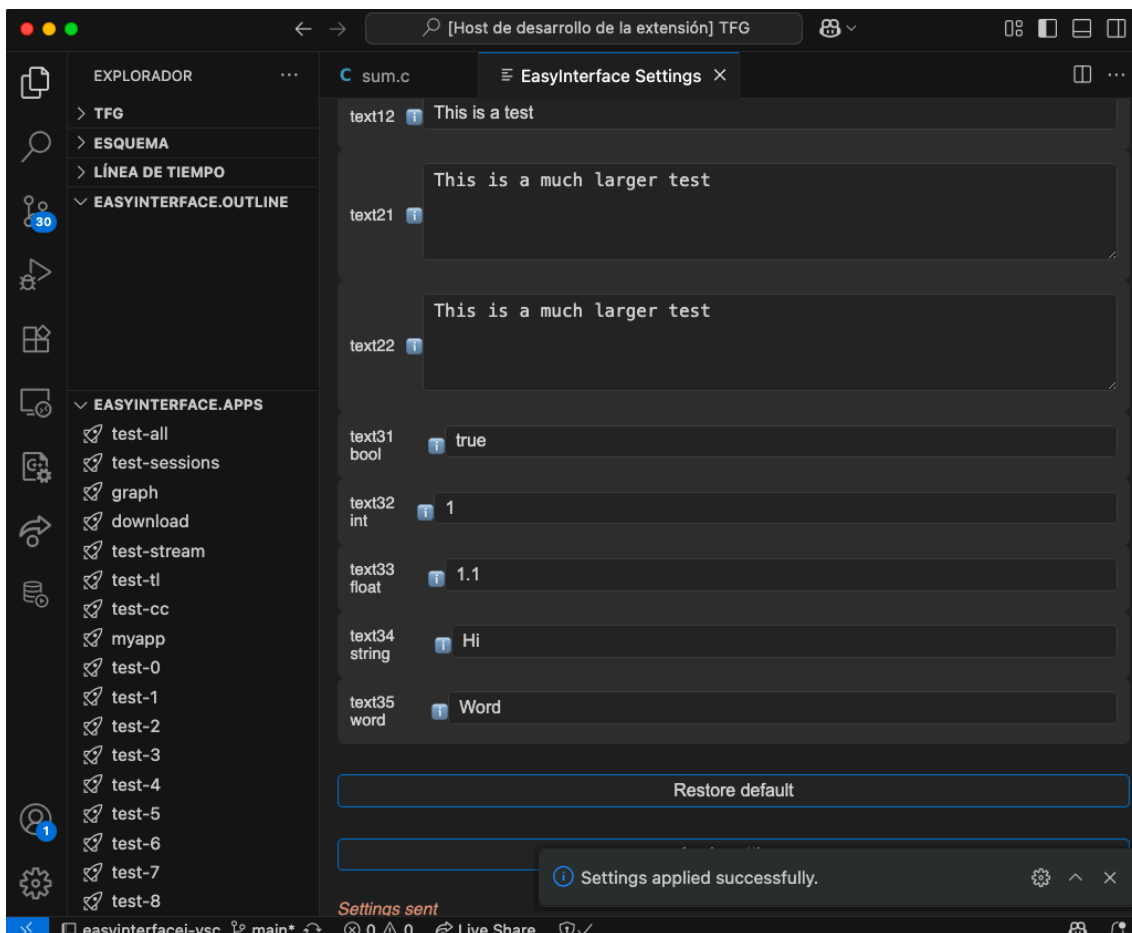


Figura 4.18: Mensaje y caja de diálogo de confirmación al guardar ajustes.

Una vez guardados los ajustes, si se cierra y abre la ventana veremos, como se ha explicado previamente, los valores guardados. En caso de querer deshacer nuestra selección, podremos hacer clic en el botón “Restore default” y volver a guardar nuestros ajustes.

Para mostrar los distintos efectos implementados vamos a aplicar los tests 0-6. Para ello, volvemos a nuestro archivo `sum.c`, seleccionamos en el panel de la izquierda `test-0`, hacemos clic de nuevo sobre el editor y presionamos en “EasyInterface: Send file to API”.

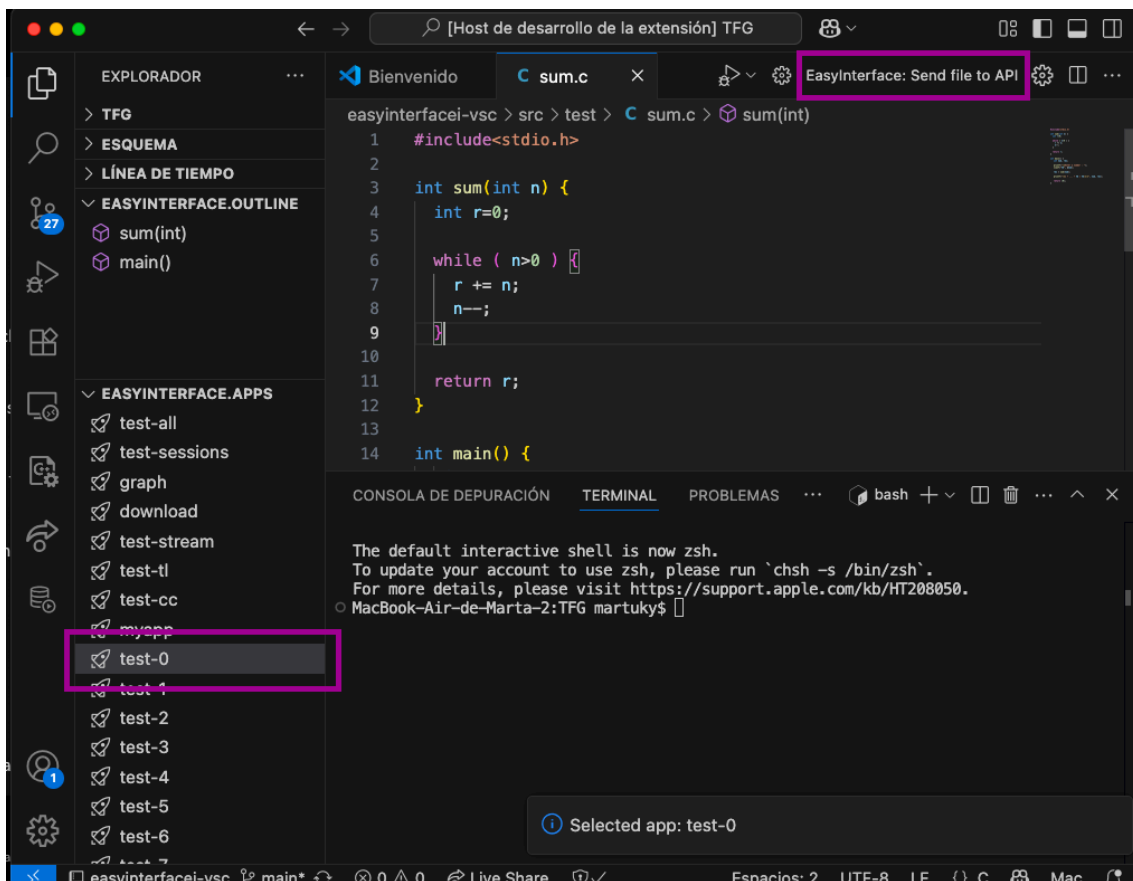


Figura 4.19: Enviar un archivo a EasyInterface con “test-0”.

La salida del primer test es la siguiente:

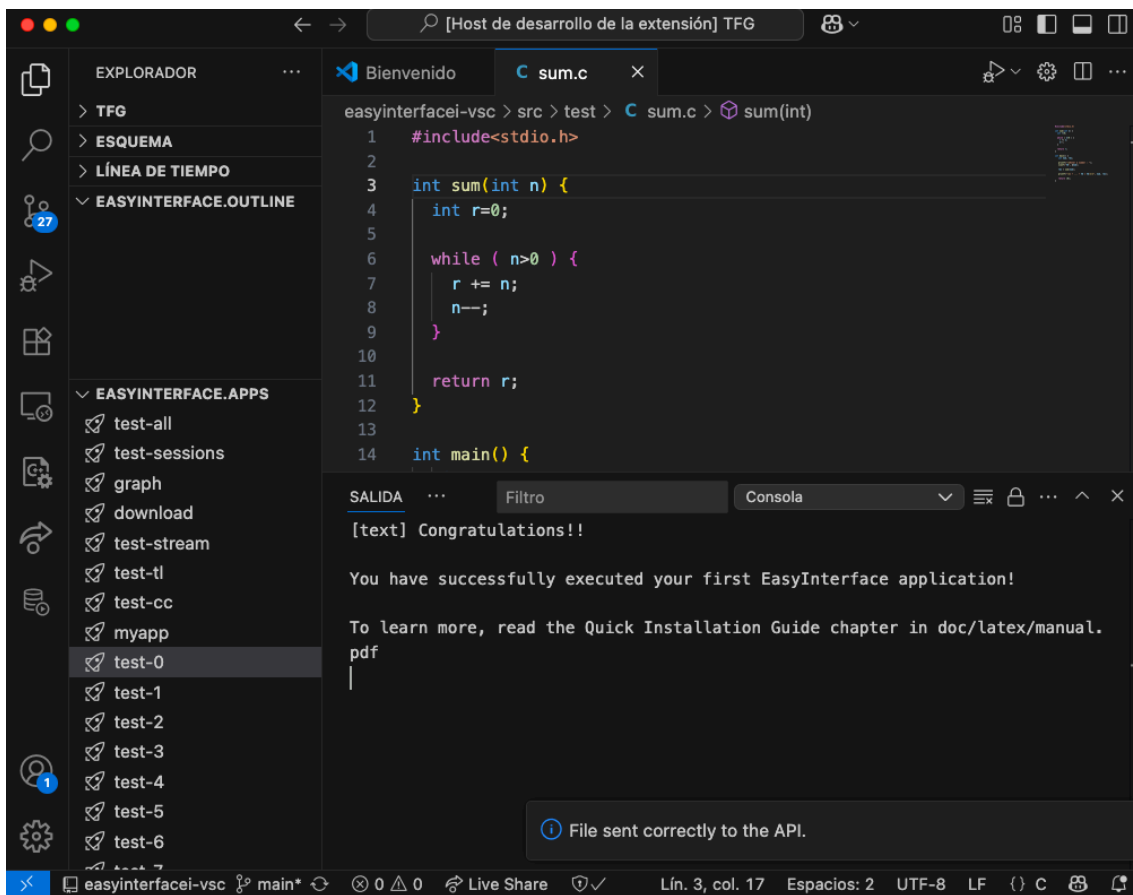


Figura 4.20: Salida test-0.

Siguiendo el mismo procedimiento, las siguientes capturas de pantalla muestran las salidas de los siguientes tests disponibles.

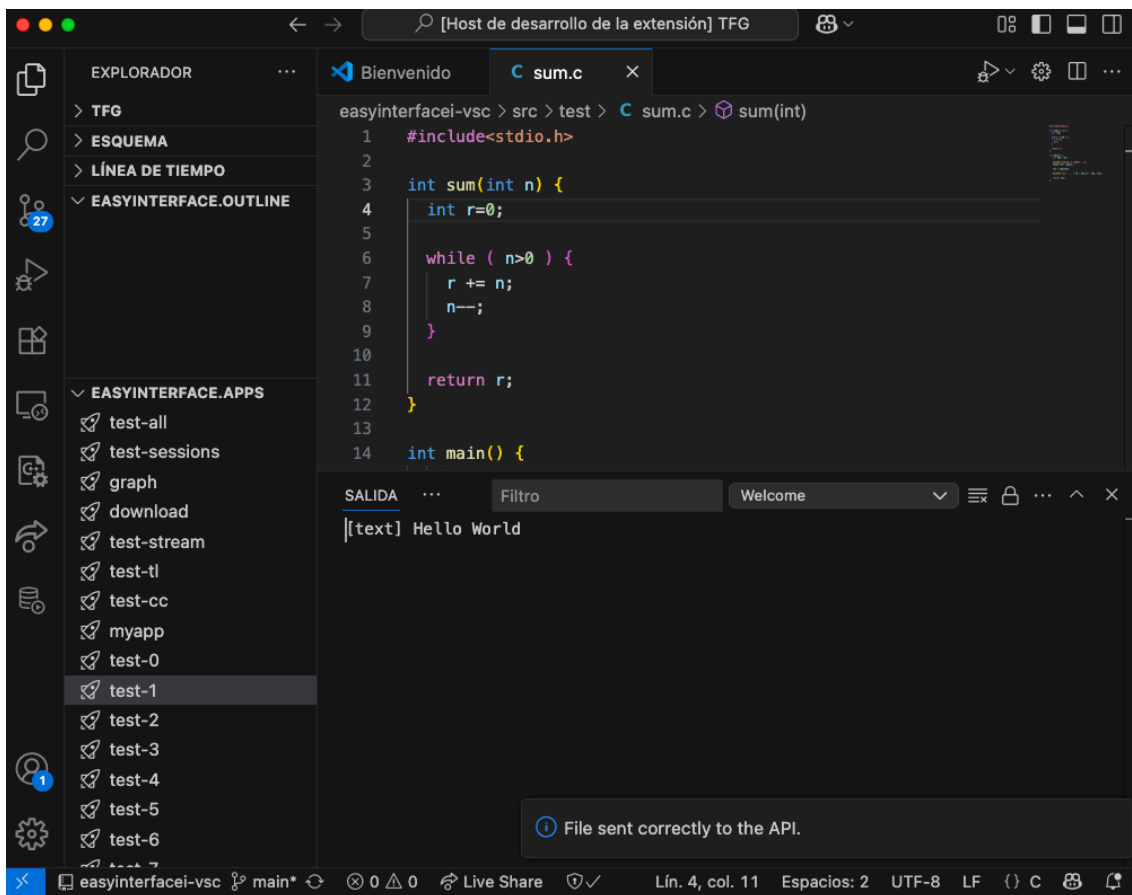


Figura 4.21: Salida test-1, imprimir en consola.

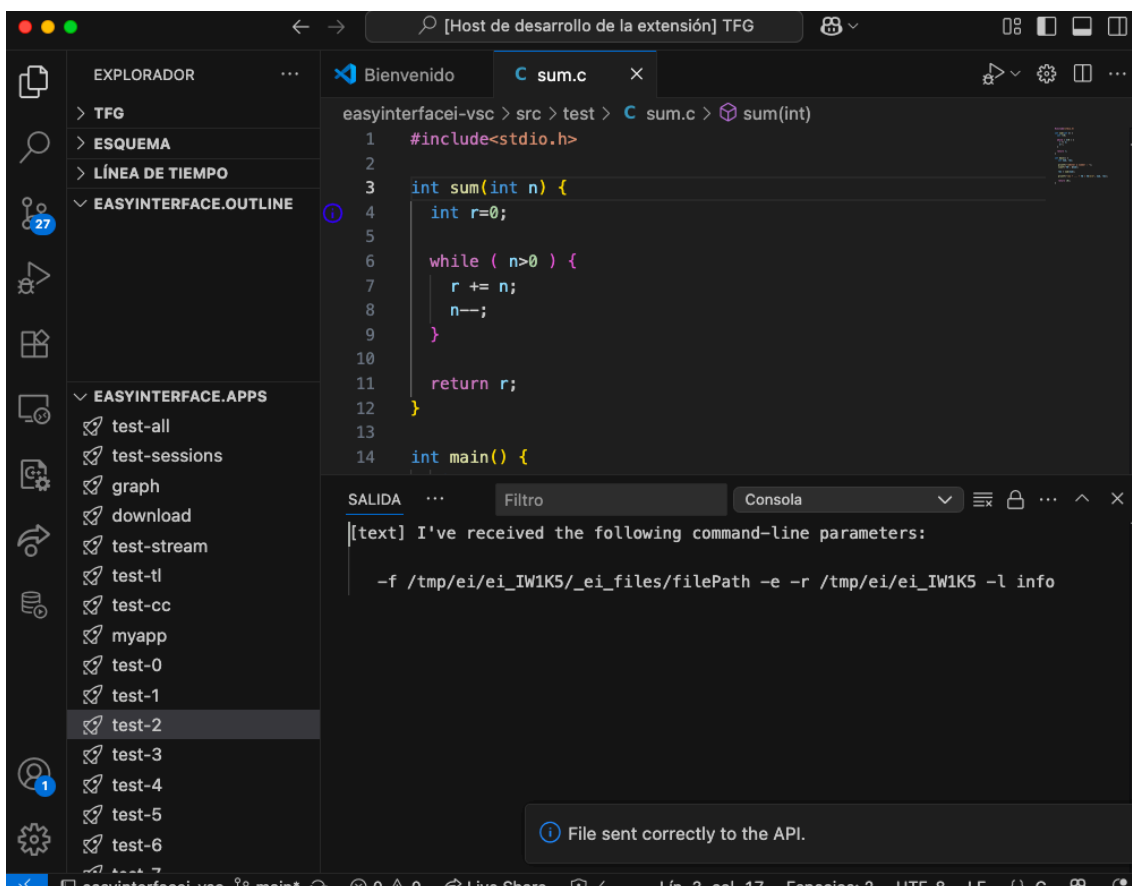


Figura 4.22: Salida test-2, marcador de información.

Al pasar por la línea resaltada número 4, se muestra el siguiente mensaje.

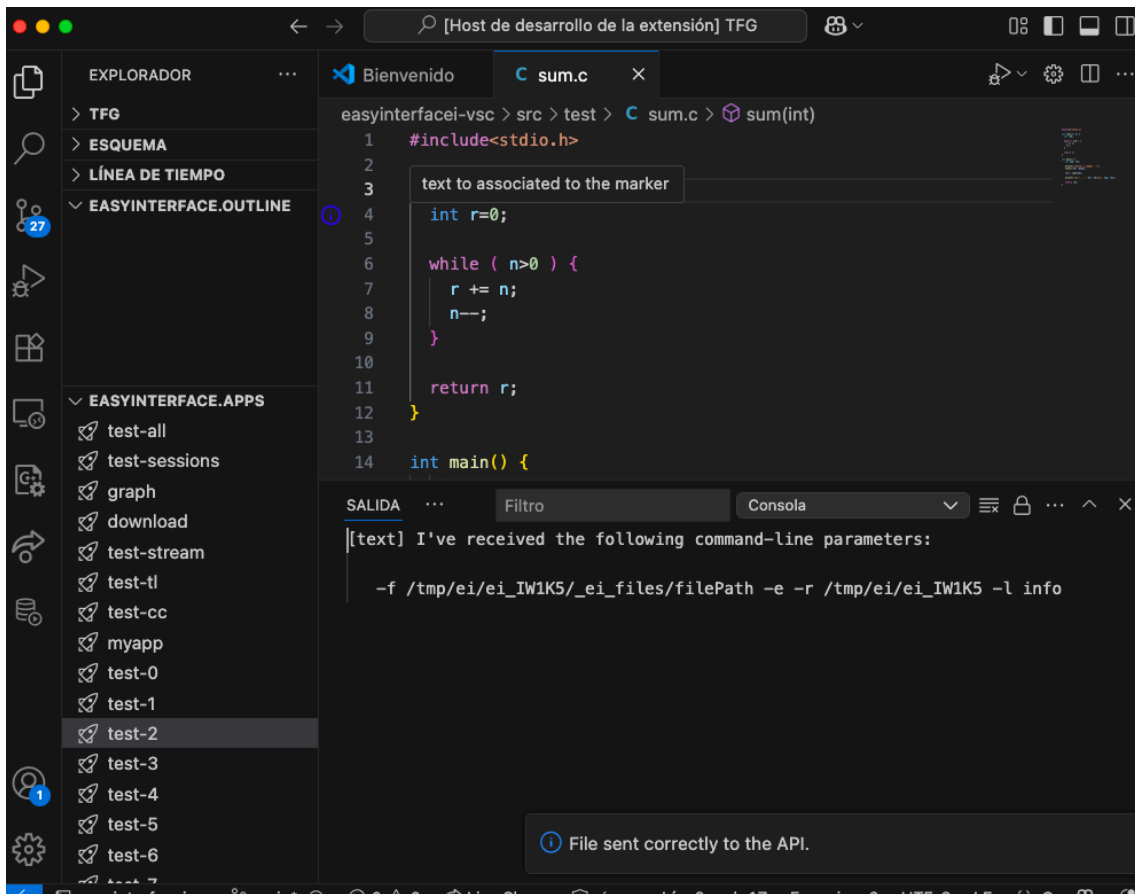


Figura 4.23: Salida test-2, marcador de información con texto asociado.

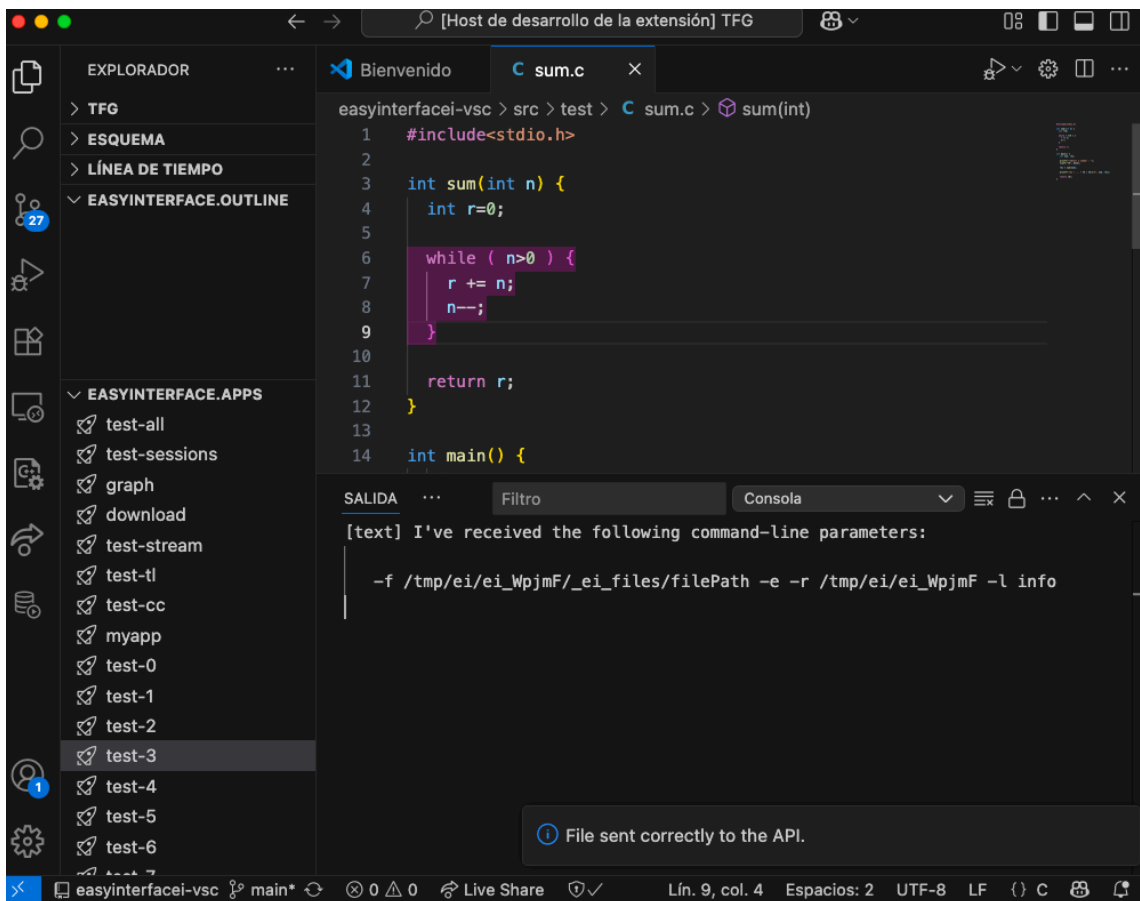


Figura 4.24: Salida test-3, resaltar líneas.

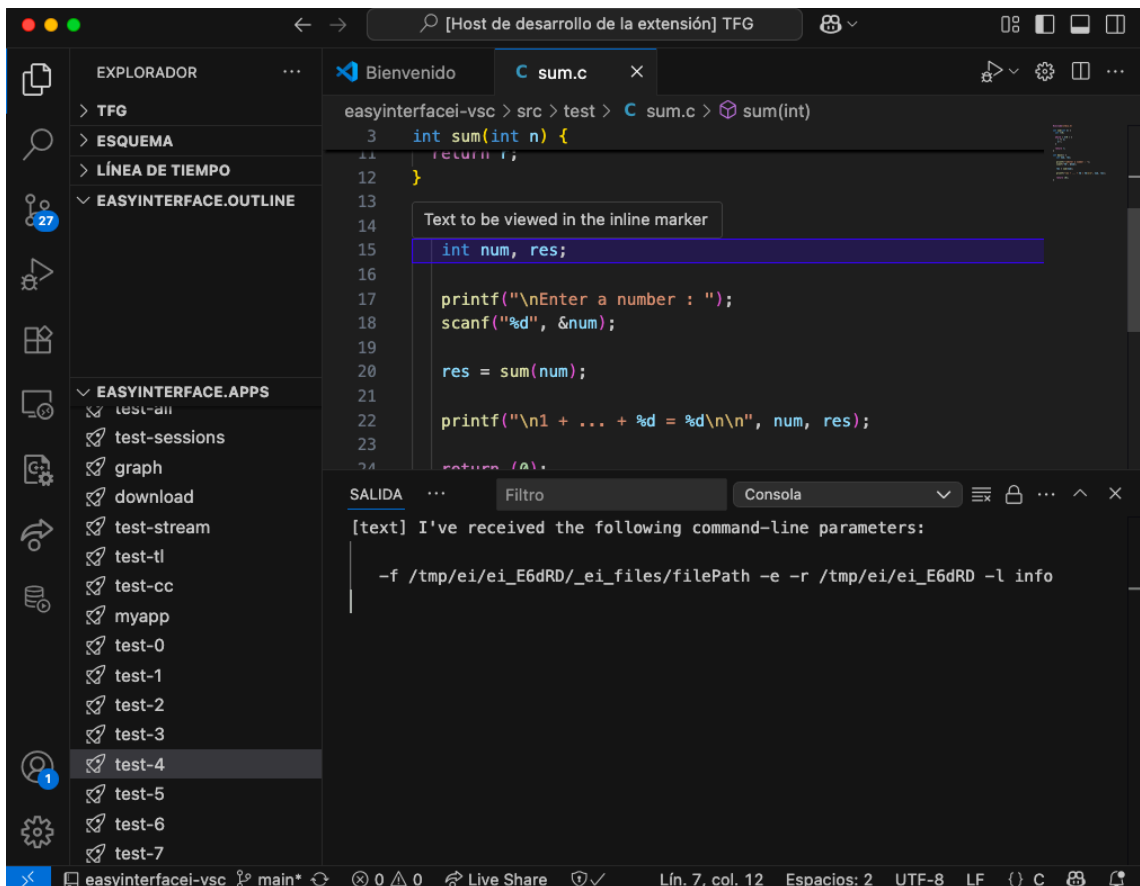


Figura 4.25: Salida test-4, marcador de línea con texto asociado.

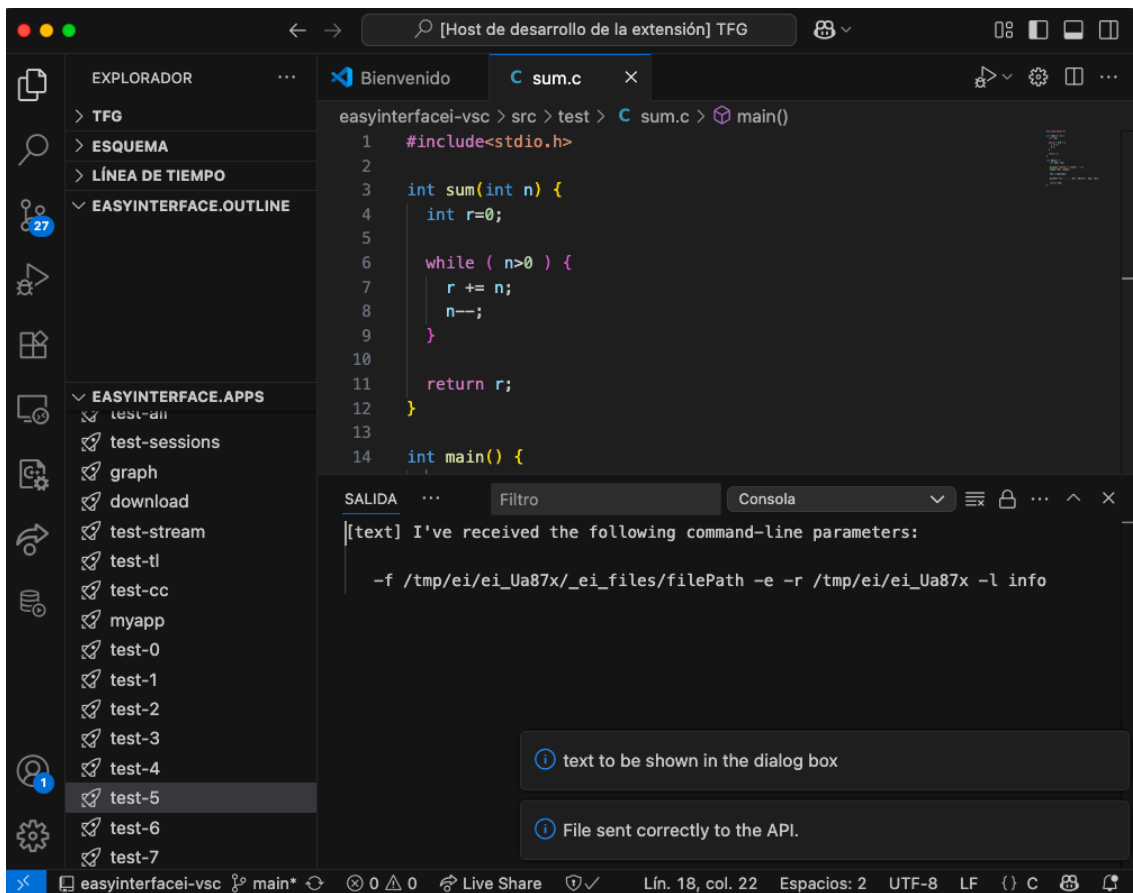


Figura 4.26: Salida test-5, cuadro de diálogo.

El siguiente test muestra una acción en la línea 20, con un marcador al lado.

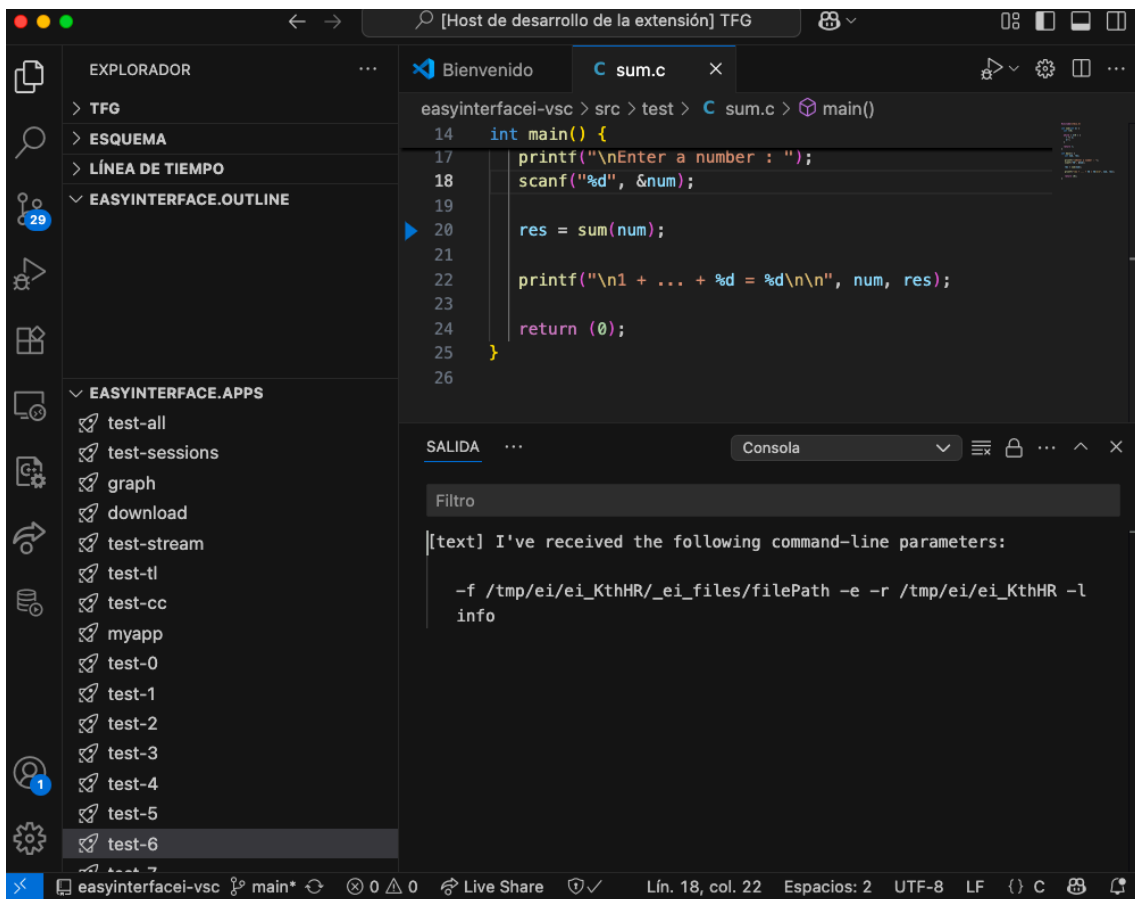


Figura 4.27: Salida test-6, acción en línea 20.

Al pasar el ratón por encima de dicha línea (señalada por el triángulo azul) vemos el texto “Right click to execute action”, como se puede apreciar a continuación.

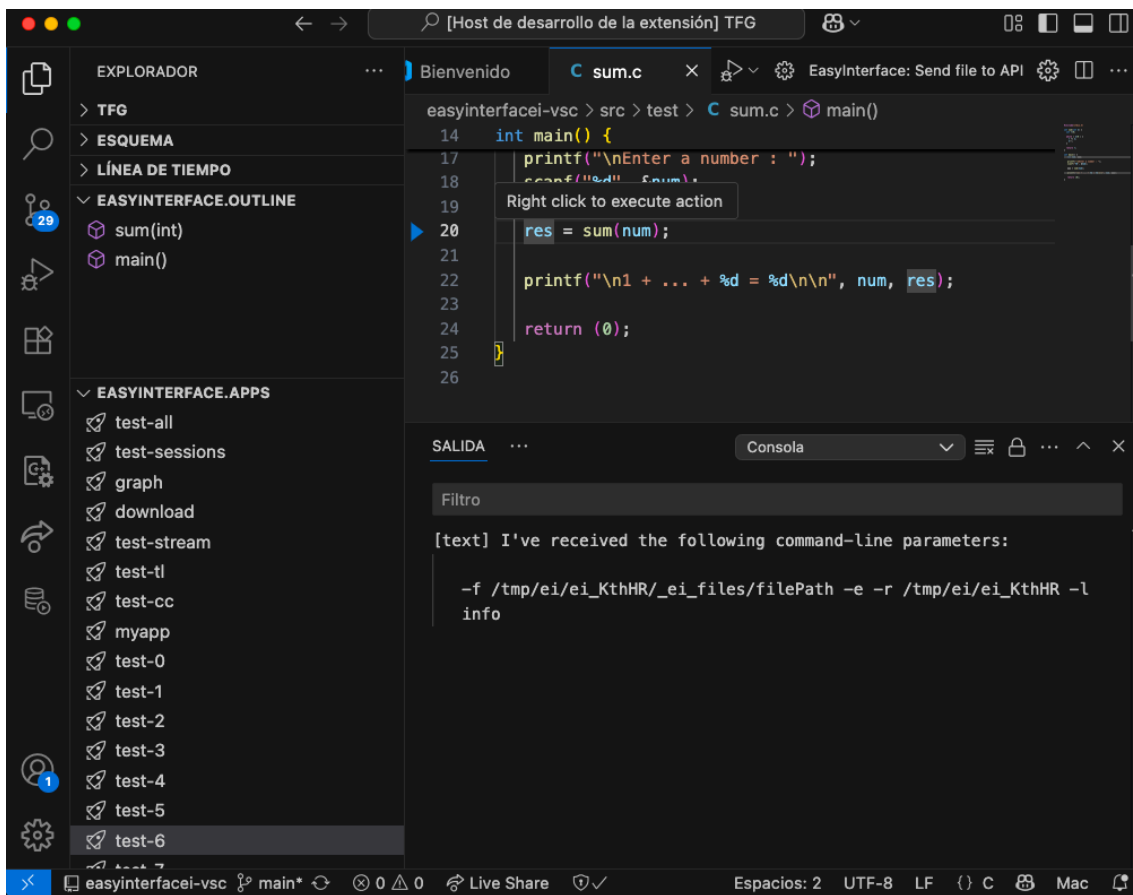


Figura 4.28: Salida test-6, acción en línea 20 con texto asociado.
Al hacer clic derecho se muestra la siguiente opción:

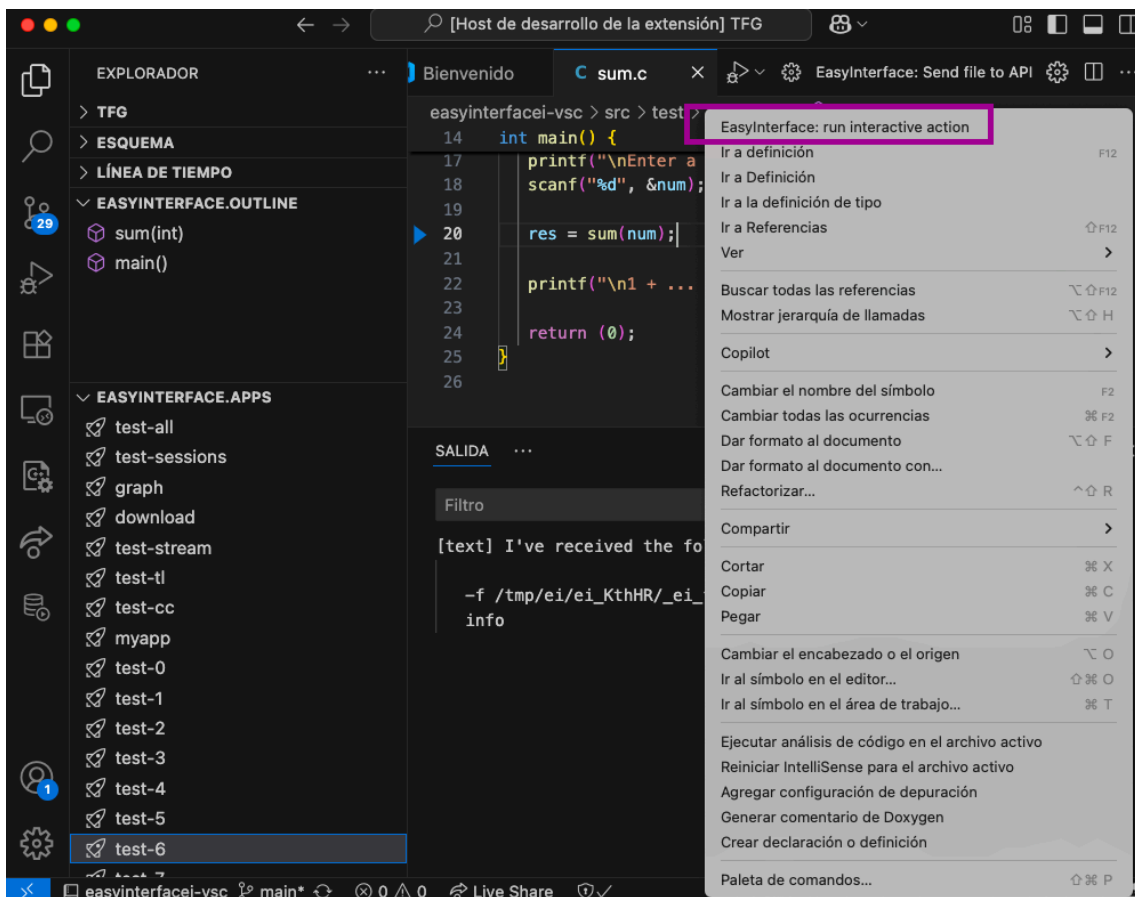


Figura 4.29: Salida test-6, acción en línea 20 al hacer clic derecho.

Cuando hacemos clic en “EasyInterface: run interactive action”, podemos comprobar que se ejecutan dos comandos: resaltar líneas y cuadro de diálogo.

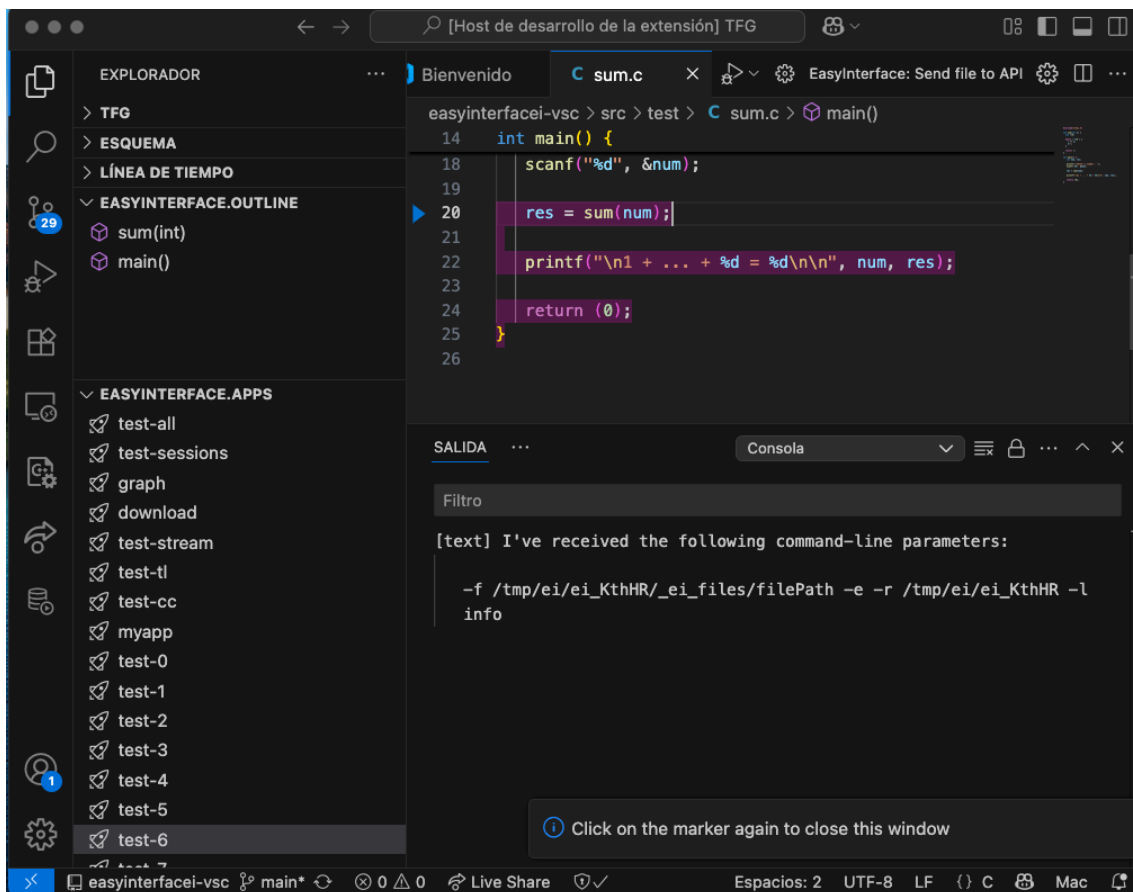


Figure 4.30: Salida test-6, acción en línea 20 ejecutada.

Cuando volvemos a hacer clic derecho en la línea 20 y a volver a seleccionar “EasyInterface: run interactive action”, los comandos se deshacen y se muestra otro cuadro de diálogo.

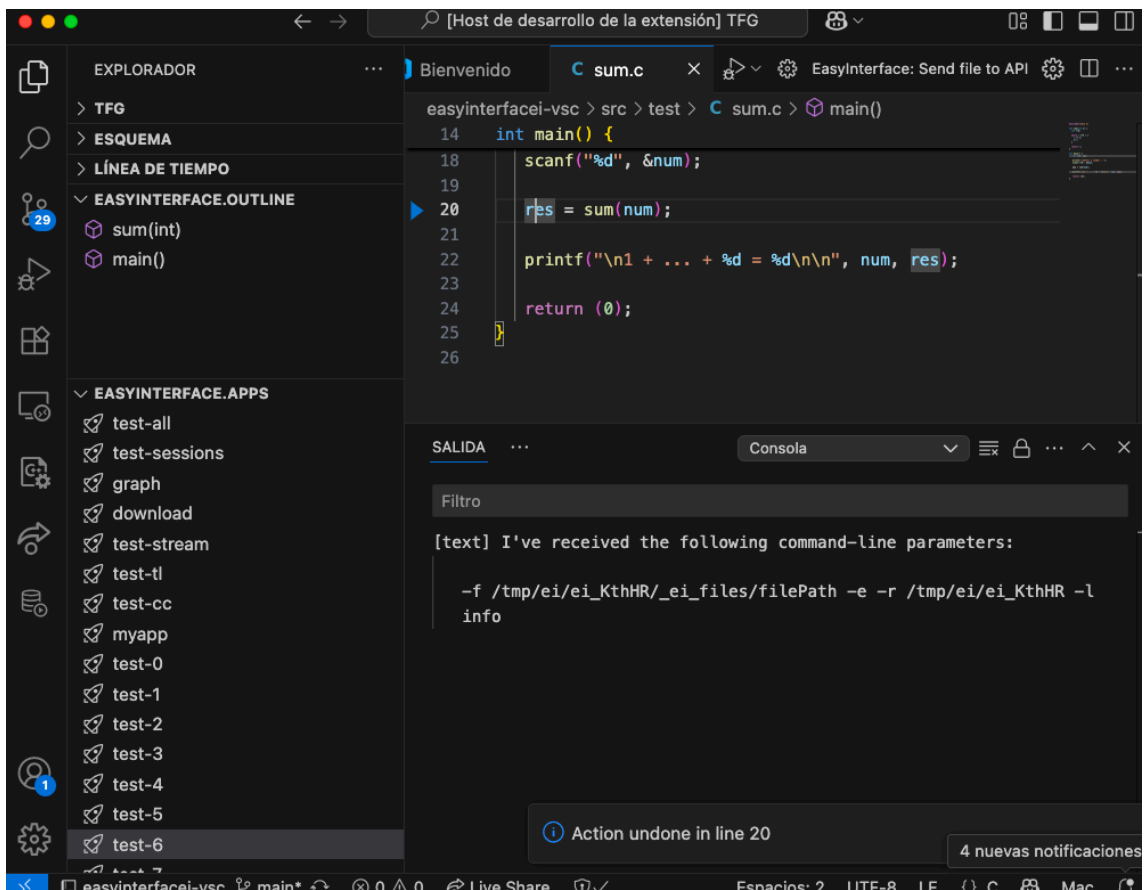


Figura 4.31: Salida test-6, acción en línea 20 revertida.

En caso de que quisiéramos enviar funciones concretas, seleccionaremos una (o varias) en el Outline y enviaremos.

5 Resultados y conclusiones

Al comienzo de nuestro proyecto nuestro principal objetivo era crear una extensión de Visual Studio Code que lograra conectarse con el servidor de EasyInterface, proporcionar una manera de modificar los parámetros de las aplicaciones disponibles, enviar y recoger el resultado del análisis del usuario y mostrarlo de forma visual, atractiva e intuitiva. Tras terminar la implementación de todas las funcionalidades y realizar los tests oportunos, podemos concluir que dicho objetivo ha sido conseguido de forma satisfactoria.

Las dificultades encontradas no han sido pocas. Durante este proyecto he tenido que enfrentarme a diversos lenguajes de programación que desconocía, lo que ha hecho que mi conocimiento de HTML, TypeScript y JavaScript aumente considerablemente. Asimismo, entender EasyInterface desde cero, averiguar cómo construir las peticiones al servidor y procesar las respuestas no ha sido tarea fácil, pero en mi opinión, esto me ha proporcionado las herramientas necesarias para enfrentarme a problemas similares en el futuro.

Puedo concluir de manera satisfactoria que este proyecto me ha permitido adquirir y consolidar una gran variedad de conocimientos técnicos y prácticos. He logrado integrar el código de forma modular, aplicando de manera efectiva los conceptos y metodologías aprendidos a lo largo de la carrera.

Además, la integración de APIs externas y la gestión de comandos me han permitido profundizar en la arquitectura de software y el diseño orientado a eventos, reforzando mi capacidad para estructurar proyectos complejos de manera eficiente. Considero que la organización del código, el uso de ejemplos prácticos y la documentación clara han sido claves para el éxito de la solución desarrollada.

Gracias a este proceso, me siento preparada para afrontar nuevos retos profesionales y avanzar hacia la siguiente fase de mi formación o carrera, con la confianza de haber desarrollado las competencias necesarias en aspectos tan diversos como desarrollo, trabajo autónomo u organización.

La extensión está lista para ser utilizada tanto por investigadores como por usuarios en general, y gracias a su estructura modular, se encuentra preparada para futuras ampliaciones que permitan mejorar código o incorporar nuevas funcionalidades.

6 Análisis de impacto

La realización de este proyecto y el desarrollo de esta extensión de EasyInterface para Visual Studio Code ha supuesto un avance significativo en mi formación profesional y personal, permitiéndome aplicar conocimientos técnicos avanzados. Esto ha reforzado mi capacidad para diseñar soluciones modulares, integrar servicios externos y documentar de manera rigurosa, preparándome para afrontar retos de mayor envergadura en el ámbito tecnológico.

Desde la perspectiva de impacto social y alineación con los Objetivos de Desarrollo Sostenible de la Agenda 2030, este proyecto contribuye a los siguientes ODS:

- **Educación de calidad (ODS 4):** facilita el acceso a herramientas de análisis avanzadas para estudiantes e investigadores, promoviendo así el aprendizaje práctico y el desarrollo de competencias técnicas en entornos educativos y científicos.
- **Trabajo decente y crecimiento económico (ODS 8):** impulsa la productividad y la empleabilidad al facilitar el uso eficiente de herramientas avanzadas, mejorando así las competencias profesionales de los usuarios.
- **Industria, innovación e infraestructura (ODS 9):** fomenta la innovación tecnológica y la modernización de infraestructuras digitales al integrar soluciones de análisis avanzadas en un entorno de desarrollo ampliamente utilizado.
- **Alianzas para lograr los objetivos (ODS 17):** promueve la colaboración entre investigadores, instituciones y usuarios, promoviendo el trabajo conjunto hacia metas comunes.

7 Bibliografía

- [1] Cuadrado, Gustavo Cimas. “Visual Studio Code: Editor de Código Para Desarrolladores: OpenWebinars.” *OpenWebinars.Net*, 17 Sept. 2024, openwebinars.net/blog/que-es-visual-studio-code-y-que-ventajas-ofrece/.
- [2] Doménech, Jesús, and Samir Genaim. “Abstools/Easyinterface: A Web Frontend for the ABS Toolchain.” *GitHub*, github.com/abstools/easyinterface. Accessed 20 Apr. 2025.
- [3] “ESLint - Visual Studio Marketplace.” *Marketplace*, marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint. Accessed 20 Apr. 2025.
- [4] “Github/Vscode-Codeql: An Extension for Visual Studio Code That Adds Rich Language Support for CodeQL.” *GitHub*, github.com/github/vscode-codeql. Accessed 20 Apr. 2025.
- [5] “SonarQube for IDE - Visual Studio Marketplace.” *Marketplace*, marketplace.visualstudio.com/items?itemName=SonarSource.sonarlint-vscode. Accessed 20 Apr. 2025.
- [6] Doménech, Jesús, Samir Genaim. *EasyInterface User Manual*, costa.fdi.ucm.es/papers/costa/eiusermanual.pdf.
- [7] “The Starting Point for Learning Typescript.” *TypeScript*, www.typescriptlang.org/docs/.

8 Anexos

El código de nuestra extensión estará disponible en el siguiente repositorio de GitLab:

<https://costa.ls.fi.upm.es/gitlab/groman/easyinterfacei-vsc.git>

Este documento esta firmado por



Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
Fecha/Hora	Wed Jul 02 17:02:10 CEST 2025
Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
Numero de Serie	561
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)