TRABAJO FINAL DE GRADO

TÍTULO DEL TFG: Servicios en la nube para procesado de código .NET

TITULACIÓN: Grau en Enginyeria Telemàtica

AUTOR: Maria Campos Pérez

DIRECTOR: Juan López Rubio

SUPERVISOR: Alex Albalá Díaz

FECHA: 4 de Febrero del 2015
Resumen

Normalmente, un proyecto final se supone que tiene que abarcar temas que has aprendido durante la titulación. Este no es mi caso, pues mi idea era encontrar algo nuevo que aprender, una motivación fuera de las aulas que no solo llenara páginas, sino que completara mi formación y mi persona.

Durante las prácticas que realicé en convenio con la universidad, conocí a unos desarrolladores de software y uno de sus proyectos me llamó la atención. Se llamaba AlterNative. La idea fundamental era proporcionar una herramienta para portar de forma completamente automática aplicaciones de lenguajes de alto nivel a bajo nivel.

Más allá de traducir aplicaciones, queríamos ser capaces de ejecutar traducciones online desde una página web, permitiendo probar AlterNative, examinar los resultados y descargarlos. Me gustó poder implicarme en algo nuevo, y quise hacer mi aportación con este proyecto final. Por este motivo no me limité a hacer una simple página web que devolviera un archivo, quise hacer algo más allá.

Este proyecto pretende traducir aplicaciones reales, mostrando los directorios de entrada y salida, y permitiendo editar los archivos necesarios para crear un código compilable que portar a nativo.

Para conseguirlo, se desarrollará un servidor web con Node.js y con una estructura de archivos un poco diferente a la habitual. Gracias a ella se podrán manejar los archivos de cada sesión de usuario, además de llevar un registro con los fallos que puedan surgir durante su uso, así como las funcionalidades que más se utilizan.
Overview

Typically, a final project is supposed to cover topics learned during the engineering degree. This is not my case, because my idea was to find something new to learn, motivation outside the classroom that not only fill pages, but to complete my training and myself.

Through the year of practices I made in agreement with the university I met some software developers, and one of his projects caught my attention. AlterNative was the name, and its essential idea was to provide a tool to port high-level applications to low-level in a fully automatic manner.

Beyond translating applications, we wanted to be able to execute online translations from a web page, allowing AlterNative to be tested, seeing the results and downloading them. I liked being able to get involved in something new, and I wanted to make my contribution with this final project. Therefore I did not limit myself to do a simple website that returned a translated file, I wanted to do something further.

This project aims to translate real applications, showing the input and output directories, and letting the user edit the files needed to create a compilable code ported to native.

To achieve this, a web server will be developed with Node.js, whose files will be structured a little differently than what you would expect. The aforementioned server will handle files of each user session, it will also keep a record of errors, shall any arise during use, as well as the most used features.
ÍNDICE

INTRODUCCIÓN ........................................................................................................... 1

1 CONTEXTO DEL PROYECTO ..................................................................................... 3

1.1 Presentación del proyecto ....................................................................................... 3
  1.1.1 AlterNative ......................................................................................................... 3
  1.1.2 Motivación ......................................................................................................... 6

1.2 Arquitectura del Sistema ......................................................................................... 7

1.3 Tecnologías .............................................................................................................. 8
  1.3.1 HTML5 ............................................................................................................... 8
  1.3.2 CSS3 .................................................................................................................. 8
  1.3.3 Bootstrap ........................................................................................................... 9
  1.3.4 JavaScript .......................................................................................................... 9
  1.3.5 jQuery ................................................................................................................ 9
  1.3.6 Node.js ............................................................................................................. 10

2 PLANIFICACIÓN ........................................................................................................ 12

2.1 Casos de uso ............................................................................................................ 12

2.2 Necesidades ............................................................................................................. 12

2.3 Diseño del proyecto ................................................................................................ 13
  2.3.1 Modelo cliente-servidor ..................................................................................... 13
  2.3.2 Cliente ............................................................................................................... 14
  2.3.3 Servidor ............................................................................................................. 14

2.4 Esquema detallado .................................................................................................. 15
  2.4.1 Frontend ............................................................................................................ 15
  2.4.2 Backend ............................................................................................................ 16

2.5 Mockup ..................................................................................................................... 17

3 IMPLEMENTACIÓN .................................................................................................. 20

3.1 Creación de un servidor HTTP ............................................................................... 20
  3.1.1 Instalación de Node.js ....................................................................................... 20
  3.1.2 Configuración del proyecto: package.json ....................................................... 20
  3.1.3 Servidor HTTP básico ..................................................................................... 21
  3.1.4 Servir un HTML ............................................................................................... 23

3.2 Añadiendo express.js ............................................................................................. 25
  3.2.1 Express ............................................................................................................. 25
  3.2.2 Static ................................................................................................................ 28
  3.2.3 Session ............................................................................................................. 28

3.3 Página web del proyecto ........................................................................................ 30
  3.3.1 Herramientas para la edición del código a traducir ............................................ 31
  3.3.2 Traducción del código ....................................................................................... 33
  3.3.3 Descarga de los resultados .............................................................................. 34

3.4 Estructura de los archivos ....................................................................................... 35
3.5 Funcionalidades ............................................................................................................ 37
  3.5.1 CodeMirror ........................................................................................................ 37
  3.5.2 jQueryFileTree ................................................................................................... 37
  3.5.3 Administración de sesiones .............................................................................. 38
  3.5.4 Edición de archivos .......................................................................................... 39
  3.5.5 Traducción de código ....................................................................................... 43
  3.5.6 Descarga de resultados .................................................................................... 45

4 CONCLUSIONES Y LÍNEAS FUTURAS ................................................................. 46
  4.1 Objetivos cumplidos .............................................................................................. 46
  4.2 Líneas futuras ....................................................................................................... 46
  4.3 Conclusiones personales ...................................................................................... 47

5 BIBLIOGRAFÍA Y REFERENCIAS ........................................................................... 48

6 ANEXO ....................................................................................................................... 49
  6.1 Rutas del servidor ................................................................................................ 49
  6.2 Uso de CodeMirror ............................................................................................. 49
  6.3 Uso de jQueryFileTree ......................................................................................... 50
  6.4 Subida de un archivo ........................................................................................... 50
INTRODUCCIÓN

Este documento recoge los pasos realizados para el desarrollo del Trabajo de Final de Grado (de ahora en adelante TFG) desde el momento de la concepción de la idea, pasando por la definición y estructuración del proyecto, la especificación de los elementos necesarios y la implementación.

El presente TFG se centra en la creación de una página web que permite portar una aplicación programada en un lenguaje de alto nivel a bajo nivel, mostrando las carpetas y archivos que forman cada una de las aplicaciones, y permitiendo descargar la aplicación nativa final.

Hoy en día, todos los sectores de servicios están incluyendo nuevos dispositivos tecnológicos conectados a la red, haciendo que el número de elementos conectados esté aumentando de manera exponencial. Elementos con diferentes sistemas operativos y por consiguiente diferentes lenguajes para su programación. El futuro de estos dispositivos tiende a converger en un entorno multiplataforma, compuesto por diferentes componentes, pero con capacidades y posibilidades similares. Sin embargo, todavía no estamos preparados para que esto suceda. Los desarrolladores se están viendo obligados a diseñar e implementar aplicaciones en todos los diferentes sistemas operativos que requieran utilizarlas.

Con la intención de resolver el problema del rápido desarrollo de aplicaciones en múltiples plataformas surge AlterNative, una nueva plataforma que trata de resolver nuestros problemas.

AlterNative nace a partir del pensamiento de aprovechar al máximo maximizar el Internet de las cosas, proporcionando una herramienta que permite llevar a cabo la traducción del código, permitiendo al usuario tomar ventaja del rápido desarrollo de aplicaciones en lenguajes de alto nivel y el alto rendimiento de los de bajo nivel, así como poder ejecutar el código nativo en varios sistemas operativos.

Durante todo el TFG se analizan cada uno de los pasos llevados a cabo para la finalización del proyecto siguiendo el orden citado más abajo. Para lograrlo, la página web se aloja en un servidor en Node.js que tiene instalado AlterNative.

En el primer capítulo podremos ver el contexto del proyecto. Empezaremos con una pequeña presentación del proyecto, una definición de su componente más interesante, AlterNative, y la motivación por la cual decidí embarcarme en este viaje lleno de retos. Seguidamente se presentará la arquitectura del sistema, las partes que lo componen y cómo interactúan entre ellas para lograr una traducción en manos del cliente. Por último definiremos cada una de las tecnologías utilizadas para el desarrollo de este TFG.
En el segundo capítulo se analizará la planificación inicial del proyecto. Para empezar, los casos de uso nos ayudarán a entender las necesidades de nuestro proyecto, que también definiremos en segundo término. Más tarde hablaremos del diseño del proyecto, para entender los elementos necesarios. En última instancia veremos el plan a seguir para terminar el proyecto.

El tercer capítulo abarcará, de inicio a fin, todos los pasos por los que el proyecto ha pasado hasta su finalización; desde la creación del servidor http, la estructura de sus archivos y diseño *frontend*, hasta las herramientas necesarias para crear todas las funcionalidades que proporcionará la página web.

En el cuarto capítulo comentaremos los resultados, todo aquello que en la planificación inicial no se mencionaba y lo que queda por hacer.

En el quinto capítulo encontraremos la bibliografía y referencias utilizadas durante el desarrollo del TFG, todas ellas necesarias para completar la información presente en este documento.

En el sexto y último capítulo hallaremos los anexos, donde se incluirán los detalles del código relacionados con los puntos menos explicados en el resto del documento.
1 CONTEXTO DEL PROYECTO

En este apartado se presenta el proyecto, su estructura y las principales tecnologías necesarias para el desarrollo del mismo.

1.1. Presentación del proyecto

El proyecto se compone de una página web que actúa como enlace entre el usuario y AlterNative, una herramienta que traduce aplicaciones desarrolladas en tecnología .NET a C++, permitiendo probar sus características online y sin necesidad de descargar, instalar, etc…

Fig. 1.1: Vista de la página.

1.1.1 AlterNative

Fig. 1.2: Logotipo de AlterNative.
AlterNative es una herramienta creada por desarrolladores para desarrolladores. Su objetivo es proporcionar una herramienta para portar de forma completamente automática aplicaciones de lenguajes de alto nivel con tecnología .NET, típicamente C#, a una aplicación nativa en lenguaje C++. De esta manera nos beneficiamos del rápido desarrollo de aplicaciones en .NET y del alto rendimiento de los lenguajes de bajo nivel.

La mayoría de los sistemas operativos son compatibles con C++, por lo tanto, si una aplicación se traduce a este lenguaje, podrá ser ejecutada en diferentes plataformas.

1.1.1.1 Suposiciones

Durante el desarrollo de AlterNative se han hecho algunas asunciones iniciales:

1. La teoría computacional dice que todo código fuente puede ser traducido de una máquina a otra (ver [8] y [9]). No obstante, el código de una aplicación puede dividirse en dos partes:

   a. Código idéntico en diferentes sistemas operativos, a menudo con diferencias de sintaxis, pero con la misma base sintáctica. Esto ocurre porque las estructuras lógicas de una misma aplicación es igual en cualquier máquina o lenguaje. Además, la mayoría de algoritmos altamente optimizables también se incluyen dentro de este grupo de código. AlterNative intenta trabajar con este tipo de código.

   b. Código muy difícil de traducir por ser altamente dependiente del sistema operativo subyacente, por ejemplo, las interfaces gráficas. Existen diversas bibliotecas que intentan resolver estos problemas, aún así traducir este tipo de código es una tarea compleja. AlterNative actualmente no intenta traducir este tipo de código y asume que la aplicación inicial ha sido desarrollada utilizando alguna librería portable, por ejemplo Qt o OpenGL.

2. El código debe ser legible y fácilmente editable por los desarrolladores. Por lo tanto, el código traducido debe poderse completar, o incluso mejorar.

1.1.1.2 ¿Cómo funciona?

AlterNative utiliza transformaciones AST (Abstract Syntax Tree o árbol de sintaxis abstracta) y bibliotecas de C++ que convierten las traducciones de AST en códigos compilables. El proceso de traducción se divide en tres partes: decompilación, traducción y recompilación.
En primer lugar, el binario recibido se pasa a través de un descompilador con el propósito de extraer el código fuente. En este caso el código extraído es C#. El código no se extrae en formato de texto, se extrae en un AST, que es una representación abstracta de la jerarquía de los nodos del código.

Esta representación se organiza en un árbol desde el nivel superior (ensamblador) hasta el inferior (instrucciones, tipos y constantes).

Para entender mejor las transformaciones AST aplicadas al descompilar un código, podemos ver el siguiente ejemplo en la Fig. 1.4: Ejemplo AST., donde se representa el AST del método \texttt{void Run(string[] args)}.

En el segundo paso, se lleva a cabo la traducción del código. Este paso consiste en aplicar diferentes conversiones al AST con el fin de obtener un segundo AST que representa el código fuente, pero en otro lenguaje de programación, en este caso, en C++. 

Fig. 1.3: Proceso de traducción de AlterNative.

Fig. 1.4: Ejemplo AST.
Con estas transformaciones, AlterNative es capaz de traducir las características del binario de entrada.

Viendo el ejemplo anterior, y sabiendo que esperamos una salida en C++, el método `void Run(string[] args)` debería traducirse a `void Run(string args[])`. Veamos cómo se ve representado en el AST resultante.

![Diagrama de ANTLR](image)

Fig. 1.5: Ejemplo AST en C++.

El tercer y último paso consiste en compilar el código C++ en un nuevo ejecutable. El binario final mantiene las mismas funcionalidades del primero, con la ventaja de su ejecución en código nativo.

1.1.2 Motivación

Durante los doce meses de prácticas que realicé en convenio con la universidad, conocí a los desarrolladores de AlterNative y pude involucrarme en sus ideas sobre el proyecto.

Había problemas, pues AlterNative es una herramienta con una cierta complejidad para instalar. Necesitaban que otros desarrolladores la probaran y recibir un feedback que hasta la fecha no teníamos, bien porque la funcionalidad que buscaban estaba incompleta, o porque desistían en su intento de probar la herramienta.

Para favorecer su uso y distribución decidimos que la mejor solución era conseguir ser capaces de ejecutar traducciones online desde una página web, permitiendo al desarrollador que la pruebe, examinar los resultados de las
traducciones y descargarlos. Además, buscábamos una forma de ver los fallos que iban surgiendo, así como las funciones que más se emplean, etc.

A raíz de esta decisión, me embarqué en este proyecto, donde no solo tendremos una página web que pueda traducir el código, sino que además será capaz de llevar un histórico de los errores más comunes en las traducciones, para tener alguna referencia sobre los detalles a mejorar de ahora en adelante.

### 1.2 Arquitectura del Sistema

La página web permitirá al usuario introducir un nuevo programa para ser traducido, ya sea subiendo los diferentes ficheros de código fuente o subiendo directamente un binario de .NET. Está implementada utilizando las tecnologías HTML5, CSS3, JavaScript, Bootstrap y jQuery, que forman parte del conjunto de herramientas más habituales actualmente para el desarrollo frontend.

El servidor web recibirá las peticiones del cliente y actuará al respecto. Las tecnologías que lo componen son Node.js y express.

Una vez el cliente inicie una traducción del código, el servidor utilizará AlterNative para realizarla. AlterNative recibirá un binario que traducirá como hemos visto anteriormente en el apartado 1.1.1.2, y su salida serán los archivos generados en dicha traducción.

Con el trabajo realizado, el servidor recogerá los cambios en el directorio de salida de AlterNative y se los devolverá al cliente, mostrándolos en la página web.
1.3 **Tecnologías**

Seguidamente se explican las tecnologías que forman parte del proyecto.

### 1.3.1. HTML5

![Logotipo de HTML5](image)

**Fig. 1.7: Logotipo de HTML5.**

HTML5 es la nueva versión del lenguaje HTML (Hyper Text Markups Language), que hace referencia al lenguaje de marcado utilizado para la creación de páginas web, utilizando una sintaxis de marcas o etiquetas.

Esta tecnología nos permite crear elementos que ayudan a estructurar un documento/página web, y permite la creación de referencias entre ellos, conocidas como enlaces.

### 1.3.2. CSS3

![Logotipo de CSS3](image)

**Fig. 1.8: Logotipo de CSS3.**

CSS3 nos permite aislar la estructura de una página web de su aspecto, de esta manera podemos mantener el código HTML más limpio y legible. En concreto, nos da la oportunidad de dar estilos visuales a los elementos, cambiando el color, familia o estilo de la letra, tamaño, color de fondo, márgenes, etc.
1.3.3. **Bootstrap**

![Bootstrap](image)

Fig. 1.9: Logotipo de Bootstrap.

Bootstrap es un framework desarrollado por twitter que permite crear interfaces con CSS y JavaScript que adaptan automáticamente su tamaño al de la pantalla del dispositivo de visualización. Ofrece un conjunto de estilos, iconos y scripts que permiten dar estilo rápidamente a los elementos.

1.3.4. **JavaScript**

![JavaScript](image)

Fig. 1.10: Logotipo de JavaScript.

JavaScript es un lenguaje de programación ampliamente utilizado en el mundo del desarrollo web por ser muy versátil y potente. Se utiliza principalmente para crear páginas web dinámicas, es decir, páginas que incorporan efectos como texto que aparece y desaparece, animaciones, acciones que se activan al pulsar botones y ventanas con mensajes de aviso al usuario.

Técnicamente, JavaScript es un lenguaje de programación interpretado, por lo que no es necesario compilar los programas, sino que se pueden probar directamente en cualquier navegador sin necesidad de procesos intermedios.

1.3.5. **JQuery**
jQuery es una biblioteca de JavaScript que simplifica la forma de interactuar con los HTML. Permite modificar el árbol DOM de forma más sencilla, adaptándose a cada navegador. Además, permite manejar eventos, desarrollar animaciones y añadir interacción con AJAX a las páginas web.

### 1.3.6. Node.js

Node.js es un entorno de programación en la capa de servidor basado en JavaScript para construir aplicaciones de red fácilmente escalables, como por ejemplo, un servidor web. Es asíncrono, con E/S (entrada y salida) de datos en una arquitectura orientada a eventos.

#### 1.3.6.1. JavaScript en el lado del servidor

JavaScript nació en los navegadores web, en el lado del cliente, ofreciendo mejoras en la interfaz de usuario y en páginas web dinámicas. Pero esto es solo un contexto, vemos lo que podemos hacer con el lenguaje en el *frontend* de un navegador, pero no todo lo que puede llegar a hacer. JavaScript es un lenguaje que se puede usar en muchos otros contextos.

Node.js es otro nuevo contexto. Nos permite ejecutar código JavaScript en el *backend*, es decir, fuera de un navegador, para ello utiliza el motor V8 creado por Google (ver [3]).
1.3.6.2. **Módulos y configuración**

Para sacar el mayor partido a Node.js, se pueden utilizar los módulos que vienen por defecto, o añadir otros desarrollados por terceros. Estos módulos completan y hacen más sencilla la tarea de puesta en marcha de una aplicación, pues no hay que escribir desde cero las funcionalidades que añaden a nuestra aplicación.

Como en todos los tipos de aplicación conocidos, en Node.js habrá un archivo llamado *package.json* que tiene la configuración inicial de la aplicación, los datos más relevantes y las dependencias necesarias para utilizarla.
2 PLANIFICACIÓN

2.1 Casos de uso

Para empezar, pensemos qué podrá hacer un usuario que quiera traducir un código utilizando nuestra página web.

- El usuario debe ser capaz de ejecutar la aplicación con un navegador web.
- El usuario podrá ver ambos lados de la traducción. La entrada y la salida con las carpetas y archivos que componen ambas partes, así como el código asociado a los mismos.
- El usuario debe ser capaz de hacer traducciones de código C# de diferentes maneras:
  - Subiendo un archivo .exe o .dll que la propia aplicación descompilará generando los archivos y carpetas que lo componen.
  - Creando un proyecto desde cero, escribiendo el código de cada archivo nuevo en el editor de código de la propia página.
  - Utilizando ejemplos de diferentes funcionalidades que añadiremos a la página.
- El usuario podrá editar los archivos de entrada, cambiando su nombre y contenido, así como crear otros nuevos o eliminar los obsoletos.
- El usuario verá, en todo momento, el progreso de la traducción y los errores que surjan al utilizar AlterNative.
- El usuario podrá descargar la aplicación resultante de la traducción en un archivo .zip, además del binario correspondiente.

2.2 Necesidades

Viendo los casos de uso mencionados en el apartado anterior, hagamos un desglose de las funcionalidades que debe cumplir nuestra aplicación.

- Queremos que el usuario pueda acceder a una página web, por lo que necesitamos un servidor HTTP.
El usuario podrá ver los archivos y carpetas que componen las aplicaciones de entrada y salida de AlterNative, así que precisaremos un explorador de archivos.

Además, el usuario podrá ver el contenido de los archivos y editar el contenido correspondiente a la aplicación que quiera traducir. Para ello utilizaremos un visualizador y editor de texto que pueda trabajar con código.

Cada usuario será independiente y tendrá sus propios archivos y traducciones, para conseguirlo necesitaremos que el servidor pueda administrar sesiones.

La traducción se lleva a cabo con AlterNative, así que requeriremos que el servidor tenga AlterNative instalado.

Para ver los resultados de la ejecución de la traducción de AlterNative, necesitaremos un registro a modo de consola, y permitiremos que el usuario lo vea en forma de una ventana nueva.

Al descargar los resultados, también necesitaremos que el servidor pueda crear un archivo .zip y el binario de la aplicación, y permita que el usuario lo descargue.

2.3 Diseño del proyecto

2.3.1 Modelo cliente-servidor

Para la realización del proyecto, se seguirá una arquitectura cliente-servidor, que se basa en cubrir la necesidad de comunicar diferentes dispositivos de una red mediante una comunicación bidireccional.

Esta estructura es perfecta para aplicaciones distribuidas, como es el caso de esta. Una aplicación distribuida consta de diferentes componentes que se ejecutan en entornos separados, normalmente en diferentes plataformas conectadas a través de una red.

La arquitectura cliente-servidor es un modelo de aplicación distribuida en el que las tareas se reparten entre los proveedores de servicios o servidores, y los demandantes o clientes. Los servidores proporcionan respuesta a las diferentes peticiones realizadas por los clientes.
2.3.2 Cliente

En la parte del cliente es donde encontramos la página web abierta desde un navegador y es donde se utilizan los datos almacenados en el servidor mediante peticiones AJAX (Asynchronous Javascript And Xml), que permiten comunicarse de forma asíncrona con el servidor para obtener datos en los momentos que se necesitan.

2.3.3 Servidor

En la parte del servidor se gestionan todos los datos que quedarán almacenados para su uso en la parte del cliente.

Como es lógico, habrá datos a los que el cliente podrá acceder directamente como son las diferentes páginas web si las hubiera, o los archivos CSS y JavaScript necesarios para la correcta carga de la página.

Sin embargo, tenemos la necesidad almacenar una serie de datos en el servidor que no van a ser accesibles para el cliente, como son los controladores que dan funcionalidad al propio servidor, o los históricos que almacenaremos para el control de fallos de AlterNative.

Por este motivo, debemos estructurar los archivos de una forma que permita restringir el acceso a los usuarios. En el apartado “3.4 Estructura de los archivos” resolveremos este problema.
2.4 Esquema detallado

Una vez hemos visto las necesidades de nuestra aplicación, podemos hacer un pequeño esquema de la aplicación, con las tecnologías que utiliza y las necesidades plasmadas en diferentes soluciones que vamos a utilizar durante el desarrollo de la aplicación.

2.4.1 Frontend

Además de HTML, CSS, JavaScript, Bootstrap y jQuery, utilizaremos dos herramientas que nos permitirán mejorar la experiencia del usuario.

Como hemos visto anteriormente, el cliente debe ser capaz de ver y editar los archivos que más tarde traducirá. Para lograr este fin necesitaremos:

- **CodeMirror**, un editor de texto que soporta diferentes lenguajes de programación. Permite personalizar diferentes opciones como son la búsqueda dentro del contenido, autocompletado de texto, etc. En el apartado 3.5.1 veremos su uso.
- **jQueryFileTree**, que nos creará un explorador de archivos donde podremos ver los diferentes elementos que componen nuestra aplicación. Es de gran ayuda y se puede configurar con solo una línea de jQuery. Más adelante explicaremos su uso (apartado 3.5.2).

### 2.4.2 Backend

Nuestro servidor no solo se compone de Node.js y *express*, requerirá más elementos para completar su cometido.

Será necesario que nuestro servidor tenga AlterNative y CMake instalados, pues sin ellos será imposible traducir o compilar las aplicaciones. AlterNative nos permitirá traducir los programas que el usuario cargue en nuestra página web. CMake es una herramienta que genera *makefiles* nativos y espacios de trabajo que pueden utilizarse en el entorno de desarrollo deseado, ayudando a la generación de un ejecutable.

Fig. 2.3: Esquema detallado del *backend*. 
Además, el servidor requerirá un conjunto de módulos que harán posible muchas de las funcionalidades que necesitaremos, y utilizará diferentes archivos y carpetas para poder servir contenido y procesar y responder peticiones de los clientes.

Como hemos visto anteriormente, algunos módulos son propios de Node y otros son desarrollados por terceros. En la Fig. 2.3: Esquema detallado del backend. vemos diferentes módulos; todos aquellos desarrollados por terceros estarán dentro de la carpeta node_modules del servidor, excepto los desarrollados exclusivamente para esta aplicación, que se encontrarán en hidden.

La estructura de archivos es muy sencilla. En la raíz tendremos los archivos app.js y package.json que contendrán la inicialización del servidor y su configuración, respectivamente. Además de ellos tendremos tres carpetas: una para los archivos “escondidos”, otra para los módulos, y la última para la página web.

La carpeta hidden almacenará un backup de diferentes usos que los usuarios den a la aplicación con el objetivo de mejorar aquellos puntos en los que falle y ver las utilidades más demandadas por ellos. Además, contendrá una carpeta default donde estarán todos los programas de ejemplos que los clientes podrán probar en cualquier momento. En último término, users llevará la administración de las sesiones de cada usuario, con los archivos que quiera traducir (en la carpeta input) y los traducidos (carpeta output), además, la carpeta log tendrá un registro de la salida y los posibles errores durante la ejecución de la traducción.

Las carpetas node_modules y web son autoexplicativas. La primera contendrá los módulos de Node desarrollados por terceros, y la segunda todo el contenido visible de la página web, así como los archivos JavaScript y CSS necesarios para su correcto funcionamiento.

2.5 Mockup

Nuestra aplicación es básicamente un traductor. Cuando empezamos a darle forma a esta idea, lo primero que se nos ocurre es algo parecido al traductor de google. Es una herramienta intuitiva y sencilla, y cualquiera querría eso para su aplicación. Por este motivo surgieron varias ideas para la estructuración de la página web, y también para las funcionalidades que ésta tendría.

Obviamente no nos podemos limitar a dos campos de texto y un botón para traducir. En nuestro caso necesitamos algo más elaborado, que nos permita explorar los archivos y carpetas y editarlos. Además, deseamos traducir y también descargar los resultados, por ello harán falta dos botones para estos cometidos.
Como podemos ver en la figura anterior, el diseño inicial estaba formado por dos exploradores de archivos y dos editores de texto. La parte izquierda correspondería al código que introduce el usuario y que pretende traducir, y la parte derecha con el ya traducido. Además hay dos botones, uno para iniciar la traducción y otro para descargar los resultados.

Es un diseño sencillo y práctico, parece que tiene todo lo que cabría esperar. Sin embargo, al empezar a desarrollar la aplicación, te das cuenta de que no es suficiente. Necesitamos una idea de las partes que va a tener y las herramientas que tendrá el usuario para enviar peticiones al servidor. Por este motivo, nos replantearemos la parte izquierda de la aplicación.

Fig. 2.4: Mockup inicial.

Fig. 2.5: Rediseño de la parte izquierda del mockup inicial.
Como podemos ver, este nuevo diseño tiene nuevos botones encima de las dos secciones; el explorador de archivos y el editor de texto.

Si recordamos los casos de uso, el usuario debía poder traducir un código a partir de los ejemplos, de un archivo subido a la web, o de un proyecto creado desde cero. Los botones encima del navegador de archivos pretenden solucionar esta necesidad.

- **Load** permitirá seleccionar un programa de ejemplo y cargarlo en el explorador de archivos.
- **Upload** permitirá subir un .exe o un .dll que se decompilará y cargará en el mismo lugar.
- **New** creará una carpeta nueva, vaciando el contenido actual del explorador.

Cuando pensamos en editar un archivo, nos referimos a crear, cambiar su nombre y contenido, además de borrarlo. Por ello, el usuario dispondrá de botones para crear (**New** de la derecha), renombrar (**Rename**), guardar los cambios (**Save**), y otro para eliminar (**Delete**) los archivos que tengamos abiertos en el editor de texto.

Poco a poco el proyecto va tomando forma. A partir de aquí se explicará el desarrollo de todas las funcionalidades del proyecto, desde la creación del servidor hasta la descarga de los resultados de la traducción.
3 IMPLEMENTACIÓN

3.1 Creación de un servidor HTTP

3.1.1 Instalación de Node.js

Para hacer funcionar Node.js, lo primero que necesitamos es el instalable que podemos descargar de la página oficial (ver [1]). Después de descargar e instalar Node, verificaremos que esté correctamente instalado utilizando el símbolo de sistema de la siguiente manera:

![Símbolo del sistema]

Fig. 3.1: Verificación de la instalación.

3.1.2 Configuración del proyecto: package.json

Cualquier proyecto en cualquier lenguaje debe tener un archivo de configuración. En Node este archivo se llama package.json, y debe compartir directorio con la aplicación Node que vayamos a crear.

El archivo package.json nos da el nombre de la aplicación, la versión y las dependencias necesarias para el funcionamiento del mismo, es decir, los módulos de terceros que va a necesitar. Además puede darnos más información, pero los campos que vamos a utilizar son únicamente estos.

```json
{
    "name": "Alter-Native",
    "version": "1.0.0",
    "dependencies": {
        "nombre_paquete": "X.X.X"
    }
}
```

El nodo más importante es el de dependencias, que indica el paquete y la versión utilizada.
Para incluir las dependencias en nuestro proyecto volveremos a utilizar el símbolo de sistema, llamando esta vez al npm (Node Package Manager o manejador de paquetes de Node.js) de la siguiente forma:

![Fig. 3.2: Comando para añadir las dependencias al proyecto.](image)

Este comando leerá las dependencias del archivo package.json y las instalará, copiándolas automáticamente en una carpeta llamada “node_modules” dentro del directorio del proyecto.

### 3.1.3 Servidor HTTP básico

El código para la creación de un servidor HTTP es el siguiente:

```javascript
//paso 1) requerimos los módulos que necesitamos
var http = require('http');

function requestHandler(request, response) {
    console.log("Petición recibida.");
    response.write("Hola Mundo");
    response.end();
}

//paso 2) creamos el servidor
http.createServer(requestHandler)

//paso 3) escuchamos peticiones HTTP por el puerto 3000
.listen(3000);

console.log("Servidor iniciado y funcionando en localhost:3000.");
```

#### 3.1.3.1 Prueba

Para iniciar el servidor guardaremos el código anterior en un archivo llamado app.js y lo ejecutaremos utilizando de nuevo el símbolo de sistema. Para probar su funcionamiento abriremos el navegador y escribiremos la url “localhost:3000”.
Implementación

Fig. 3.3: Ejecución del servidor.

El cliente podrá ver en su navegador la frase “Hola Mundo” escrita.

Fig. 3.4: Respuesta en el cliente.

3.1.3.2 Funcionamiento

Analicemos paso a paso lo que está sucediendo.

```javascript
var http = require('http');

http.createServer(requestHandler).listen(3000);
```

La primera línea de código `require`, requiere el módulo `http` que ya viene incluido en los módulos que tiene Node.js por defecto, haciéndolo visible mediante la variable a la que llamamos `http`.

Más adelante, llamamos a una de las funciones que nos ofrece el módulo requerido, en este caso la función `createServer`, que nos creará un servidor y devolverá el objeto asociado a este servidor. Este objeto tiene un método llamado `listen` que nos permite indicar el número de puerto por el que nuestro servidor HTTP escuchará.
Cuando nosotros llamamos al método `http.createServer`, no sólo queremos que el servidor escuche por un puerto, también queremos que haga algo cuando reciba peticiones HTTP. Por este motivo, el módulo `http` nos permite, además, pasar una función de `callback`, que en nuestro caso es `requestHandler`.

![Function](https://via.placeholder.com/150)

```javascript
function requestHandler(request, response) {
  console.log("Petición recibida.");
  response.write("Hola Mundo");
  response.end();
}
```

La función `requestHandler` tiene dos parámetros, `request` y `response`, que en realidad son objetos formados por una serie de métodos que permiten trabajar con las características de la petición HTTP y responder a la misma. El `response.end` finaliza la respuesta.

```
console.log("<<TEXTO>>");
```

El `console.log` nos permite recibir un texto en el terminal donde ejecutamos la aplicación cada vez que sucede un evento. En el caso de nuestro código, sucederá cuando iniciamos el servidor y cada vez que se ejecuta la función `requestHandler`.

### 3.1.4 Servir un HTML

Hasta ahora el servidor devolvía un “Hola Mundo” cada vez que recibía una petición HTTP. La idea ahora es permitir que el servidor pueda servir una página web alojada en una carpeta llamada “web” que comparte directorio con la ubicación de la aplicación de servidor que estamos ejecutando.

Para ello, añadiremos dos módulos necesarios para el tratamiento de archivos:

```
path = require('path'),
fs = require('fs');
```

Además, añadiremos varias líneas a la función de `callback`:
3.1.4.1 Funcionamiento

Volvamos a analizar lo que sucede.

```javascript
var fileName = path.basename(request.url);
```

El módulo `path` nos permite manejar rutas de archivos específicas. Así, cuando hacemos un `path.basename(request.url)` estamos cogiendo la ubicación y el nombre del archivo que estamos pidiendo, eso sí, eliminando la ruta inicial “localhost:3000”, de esta manera obtendremos el `fileName` o nombre de archivo.

```javascript
var localFolder = __dirname + '/web/';
```

La variable `__dirname` nos devuelve la ruta completa en que se encuentra el servidor que estamos ejecutando. Cuando añadimos “/web” estamos buscando archivos en la carpeta llamada `web` dentro de ese mismo directorio.

```javascript
var content = localFolder + filename;
```

De esta manera creamos la variable `content`, que contiene la ruta hasta la carpeta y el nombre del archivo que está pidiendo el cliente mediante la petición HTTP.
El segundo módulo requerido, fs, nos facilita el manejo de archivos del sistema de archivos (“file system” en inglés, de ahí el nombre fs). Este módulo tiene una función llamada readFile, que sirve para leer el contenido de un archivo. Como podemos ver, controlando si el archivo se lee correctamente podemos devolver el contenido al cliente web mediante un request.end(contents).

3.2 Añadiendo express.js

Express es un framework que se añade a Node.js como un módulo, proporcionando un robusto conjunto de características para aplicaciones web y móviles.

Pero además, express ofrece diferentes funcionalidades divididas en módulos que también podremos añadir a Node.js pero que a su vez no son más que sub-módulos de express.

3.2.1 Express

Para añadir el módulo principal de express a nuestro servidor necesitaremos realizar diferentes pasos.

3.2.1.1 Instalación de express.js

Como hemos hecho anteriormente, utilizaremos el npm para instalar express, de esta manera se añadirá a nuestra carpeta node_modules.
Obviamente, añadiremos *express* al archivo de configuración (*package.json*) con la versión que requiera nuestro servidor.

### 3.2.1.2 Uso de *express.js*

Nuestro servidor web hasta la fecha era muy sencillo. Se iniciaba escuchando por un puerto determinado y realizaba la función de *callback* definida cada vez que recibía una petición HTTP.

Express nos permite crear el servidor definiéndole, en vez de una única función, un conjunto de rutas que utilizaremos para cada uno de los tipos de petición HTTP que podrá hacer el cliente.

Por este motivo, el archivo llamado *app.js* que tenía toda la configuración del servidor, puede cambiarse por este otro:

```javascript
var http = require('http'),
    express = require('express');

app = express();

// rutas

http.createServer(app).listen(3000);
console.log("Servidor iniciado y funcionando en localhost:3000.");
```

### 3.2.1.3 Creando rutas

Las rutas nos permiten direccionar peticiones a los controladores correctos. Para definir una ruta debemos ver qué tipo de petición HTTP es y entonces llamar a la función del controlador que pueda llevarla a cabo.
Por ejemplo, en nuestro servidor vamos a necesitar que cuando el cliente lo solicite, se compile el código y se genere la traducción. Para este caso, cuando el cliente clique el botón “Compilar”, se generará una petición GET “/compile-csharp”, y el servidor podrá manejar esta petición utilizando una ruta de express.

```javascript
app.get('/compile-csharp',require(__dirname+'/compiler.js').compile);
```

Analizando la línea anterior, podemos ver que en las peticiones de tipo /compile-csharp se llamará a la función compile situada en el archivo compiler.js.

Para más detalles del funcionamiento de las rutas de la aplicación, podemos ver con más detalle todas las rutas del servidor en el anexo (apartado 6.1).

### 3.2.1.4 Exportar métodos

Con todo lo que acabamos de ver en el apartado anterior surge una pregunta. ¿Cómo es el archivo compiler.js y porqué se ejecuta así la función compile?

Muy sencillo. El archivo compiler.js es un controlador que utilizamos cuando el servidor lo requiere, llamando a las funciones necesarias en cada momento. De esta manera, únicamente es necesario exportar dichas funciones como si de módulos se tratara.

Para entender un poco mejor esto, podemos ver el formato de compiler.js:

```javascript
var _compile = function(req, res){
    //se encarga de procesar la petición de traducción del código
}
var _consola = function(req, res){
    //permite leer los resultados de la traducción
}
var _zip = function(req,res){
    //permite descargar un .zip con la traducción
}

module.exports.compile = _compile;
module.exports.consola = _consola;
module.exports.zip = _zip;
```

Como hemos podido apreciar en el código anterior, el archivo compile.js exporta tres métodos diferentes. Sin embargo, en esta petición solamente utiliza uno. Esto significa que habrá otras peticiones realizadas por el cliente que precisaran su llamada.
3.2.2 Static

Express nos ofrece una función para servir archivos desde un directorio raíz determinado. Como hemos visto anteriormente dicho directorio será una carpeta llamada web dentro del directorio donde se encuentra el servidor.

Esta función es `express.static(root)`, y se puede utilizar como en el código citado más abajo.

```javascript
app.use(express.static(__dirname + '/web'));
```

Por defecto se enviará el archivo "index.html" si existe. Cuando no se encuentra un archivo, en lugar de enviarse un 404 como respuesta, llama a `next()`, pasando a la siguiente petición.

3.2.3 Session

Cuando trabajamos con una aplicación en nuestro ordenador, la abrimos, hacemos algunos cambios y después la cerramos, estamos utilizando algo parecido a una sesión. El ordenador sabe quién eres y también cuándo abres y cierras la aplicación.

En Internet, sin embargo, tenemos un problema. El servidor web no sabe quién eres ni lo que haces, porque HTTP no mantiene un estado.

Para resolver esto, `express` nos ofrece un módulo llamado `express-session` que nos permite hacer exactamente lo que necesitamos. Creará un objeto Sesión que almacenará información sobre la sesión y los cambios realizados por el usuario.

Por lo tanto, habrá un objeto Sesión para cada nuevo usuario. La Sesión se eliminará una vez haya expirado.

3.2.3.1 Uso de express-session

Para habilitar el uso de sesiones en nuestro servidor necesitaremos añadir un nuevo módulo en el proyecto, añadiéndolo en el nodo dependencias del archivo "package.json".

```json
"dependencies": {
  "express": "~4.9.8",
  "express-session": "~1.9.3"
}
```
Una vez editado el archivo, utilizaremos el comando “npm install” para añadir el módulo al directorio *node_modules*, permitiendo así que el servidor lo encuentre y pueda trabajar con él.

![Imagen de instalación de express-session](image)

**Fig. 3.7:** Instalación de *express-session*.

Además, en el archivo JavaScript de nuestro servidor añadiremos el requerimiento.

```javascript
var session = require('express-session');
```

Para conseguir que funcione correctamente, almacenaremos la sesión de cada usuario en un objeto al que llamaremos “*uCookie*” con la siguiente línea de código:

```javascript
app.use(session({name: "uCookie", secret: "test", resave: true, saveUninitialized: true}));
```

Analizando esta última línea podemos ver que el método *session* recibe como parámetro varias opciones como son:

- **name**: nombre de la del objeto Sesión o *cookie*, en nuestro caso se llamará *uCookie* (“user cookie”).
- **secret**: clave secreta con la que se firma la *cookie* para evitar su manipulación.
- **resave**: puede ser verdadera o falsa, forzando a guardar nuevamente la *cookie* incluso cuando todavía no se ha modificado.
- **saveUninitialized**: puede ser verdadera o falsa, obligando a una sesión sin inicializar a guardarse.
3.3 Página web del proyecto

Cuando el cliente entre en nuestra página web para probar el funcionamiento de AlterNative, podrá ver la siguiente página:

![Fig. 3.8: Vista de la página web principal.](image)

Como se puede apreciar en la Fig. 3.8: Vista de la página web principal, en el diseño de la página web encontramos dos secciones diferentes, una a cada lado de la pantalla, separadas por el eje de mitad de página.

La mitad izquierda contendrá el código fuente a partir del cual se realizará la traducción. Además, habrá una vista de las carpetas y archivos correspondientes al código, así como las herramientas necesarias para tratarlo y editarlo antes de iniciar la prueba de AlterNative.

La mitad derecha, por el contrario, contendrá el código ya traducido. También la vista de los archivos y carpetas generados por AlterNative para la traducción. Sin embargo, en este caso no se podrá modificar el contenido. Por otro lado, nos permitirá descargar el resultado en un archivo .zip y el binario generado.

Para ver el código HTML asociado a la página, podemos visitar el anexo.
3.3.1 Herramientas para la edición del código a traducir

Como podemos ver en la Fig. 3.9, la sección izquierda de nuestra página web consta de dos partes, el explorador de archivos y el editor de archivos. Cada una de ellas nos ofrece herramientas para personalizar el código que más adelante compilaremos. Además, el botón compile nos permitirá iniciar la traducción.

Fig. 3.9: Herramientas de gestión y edición de archivos.

Analicemos estas herramientas según la numeración que siguen en la imagen anterior:

1. **Load** contiene un desplegable con todos los ejemplos de programas sencillos para poder probar las posibilidades de AlterNative y los diferentes tipos de código de entrada.
2. **Upload** nos ofrece la posibilidad de subir un archivo *exe* o *dll* que será descompilado. Para ello se abrirá una ventana nueva que permitirá subir un archivo desde el explorador de archivos del ordenador del cliente. Los archivos y carpetas resultantes de esta descompilación se cargarán en el explorador de la parte izquierda de nuestra web, pues serán archivos que podremos modificar antes de traducir.

3. **New Project** nos crea un nuevo proyecto, añadiendo una carpeta con el nombre que queramos.

4. **Clic en el explorador**: cuando clicamos en las carpetas del explorador, se abrirán y mostrarán los archivos que contengan. Además, al seleccionar un archivo, éste se nos descargará en el editor de código de la izquierda.

5. **Create file** nos añade un archivo en el directorio en el que estamos.

6. **Rename**: Esta herramienta tiene dos elementos: un campo de texto editable donde se muestra el nombre actual del archivo abierto, y un botón que guardará los cambios de nombre que realicemos en dicho campo.
7. Save guarda los cambios realizados en el archivo abierto actualmente en el editor de código, permitiendo añadir nuevas funciones o eliminar fragmentos de código no deseados.

8. Delete elimina el archivo abierto.

3.3.2 Traducción del código

Una vez tenemos el código que queremos traducir, ya sea hecho por nosotros, descompilado de un archivo que hemos subido, o editado de los ejemplos existentes en la web, el siguiente paso es darle al botón Compilar.

Fig. 3.12: Clicar en el botón “Compilar” para iniciar la traducción.

Una vez pulsemos el botón, el servidor comprobará que el programa no tenga errores e intentará compilarlo. Si todo va bien, iniciará su traducción utilizando AlterNative. En ese momento aparecerá una consola que mostrará el progreso.

Fig. 3.13: Consola de AlterNative.
Al finalizar el proceso de traducción aparecerán todos los archivos generados por AlterNative en la sección derecha de nuestra página web. En este punto, podremos ver las carpetas y abrir los archivos que seleccionemos, igual que pasaba en el lado izquierdo, sin embargo no se podrán editar.

Fig. 3.14: Resultado de la traducción.

3.3.3 Descarga de los resultados

Una vez finalizado el proceso y con los resultados obtenidos, podremos descargar un archivo .zip que contendrá el programa final en C++ y la aplicación nativa.

Fig. 3.15: Descarga de resultados.
3.4 Estructura de los archivos

En la raíz del servidor encontramos tres carpetas; `node_modules`, `hidden` y `web`, y dos archivos; `app.js` y `package.json`. Detallamos:

![Fig. 3.17: Contenido de node_modules.](image)

En `node_modules` es donde almacenamos las entrañas del servidor, pues contiene aquellos archivos esenciales para el correcto funcionamiento de Node.js sin los cuales no funcionaría nada.

Cada una de las subcarpetas que encontramos en este directorio nos proporciona una funcionalidad determinada:

- **express**: es el módulo explicado anteriormente en el apartado 3.2.1, recordemos que nos permite automatizar el funcionamiento de las peticiones del servidor añadiendo las rutas de los métodos necesarios.

- **express-session**: también explicado (apartado 3.2.3), con este módulo conseguimos gestionar la sesión de cada usuario por separado.

- **body-parser y multer**: son también sub-módulos de `express`, gracias a ellos se pueden leer y responder peticiones POST. Anteriormente estaban dentro de `express`, pero en las últimas versiones deben añadirse por separado.
- **jquery**: es un módulo que permite usar jQuery en el lado del servidor.

- **node-zip**: genera y extrae archivos .zip para que el usuario después pueda descargarlos.

![Fig. 3.18: Contenido de hidden.](image)

*Hidden* contiene aquellos archivos internos que no son accesibles directamente por el usuario, pues contienen información sensible, y es el servidor el que distribuye discriminadamente sus contenidos según corresponda.

Entre sus subcarpetas encontramos *default*, que contiene los programas de ejemplo accesibles a todos los usuarios. *Users* será la encargada de almacenar los archivos que usa cada usuario durante su sesión, en el apartado 3.5.3 lo veremos con más detalle. En *backup* guardaremos registros del uso que los usuarios dan a *AlterNative* para su posterior análisis.

Los archivos que encontramos, *compiler.js* y *fileStorage.js*, son los controladores del servidor. Como hemos visto anteriormente, los controladores exportan diferentes métodos como si fueran módulos, así cada petición del cliente podrá resolverse de forma particular.

De esta manera, *compiler.js* contendrá todos los métodos relacionados con la compilación y traducción del código, además de la consola asociada a los comandos que se ejecutan durante el proceso. Por otro lado, *fileStorage.js* se encargará de todo lo que tenga que ver con la gestión de archivos, pudiendo crear, renombrar, editar, copiar y eliminar. Además, se encargará de actualizar el árbol de archivos contenido en la página web.
3.5 Funcionalidades

Después de ver en los apartados anteriores la página web y la estructuración de los archivos en el servidor, tenemos una idea más o menos clara de lo que este proyecto consigue hacer. Pero, ¿cómo lo hace? En este apartado vamos a entrar en detalle y profundizar en cada una de las funcionalidades.

3.5.1 CodeMirror

Una de las partes más importantes del proyecto es el editor de texto. CodeMirror es un editor de textos versátil implementado en JavaScript especializado para la edición de código, y viene con una serie de modos de lenguaje y addons que implementan funcionalidades de edición más avanzadas. En el anexo está el código utilizado (ver 6.2).

3.5.2 jQueryFileTree

jQueryFileTree es un plugin de jQuery que facilita la creación de una vista de directorios y archivos, totalmente personalizable, con tan solo una línea de código JavaScript. Funciona utilizando un script que contiene las peticiones AJAX necesarias para cargar la vista de archivos.

Es ideal para lo que necesitamos; el siguiente código es el responsable de su implementación:

```javascript
$('#container_id').fileTree({[options]}, function(file) {
    openFile(file);
});
```

Dónde container_id es el id del elemento html que contendrá el visor de archivos, [options] las opciones que definimos para personalizarlo, y openFile la función que cargará el archivo escogido en el editor de texto.

Entre las opciones encontramos:

- root, que especifica la carpeta a mostrar.
- script, que especifica la ruta al archivo donde está el script AJAX del servidor.
- folderEvent, nos define el evento necesario para mostrar/esconder los elementos de una carpeta. En este caso definiremos el evento “clic”, para que al seleccionar una carpeta se vean los archivos de su interior.

De esta manera podemos crear un explorador de archivos. El código utilizado podemos verlo en el anexo (apartado 6.3).
3.5.3 Administración de sesiones

Como hemos visto anteriormente, para la administración de la sesión de un usuario utilizamos una cookie llamada uCookie.

Además, como cada uno de los usuarios que se conecte necesitará un entorno de trabajo, el servidor alojará una nueva carpeta durante la sesión del usuario. Esta carpeta recibe el nombre a partir del contenido de la cookie.

Dentro de la carpeta de cada usuario habrá diferentes carpetas, cada una con un cometido específico:
La carpeta *input* nos almacenará el *workspace* del código inicial del usuario, es decir, todo lo que se mostrará en la parte izquierda de la página web, los archivos y carpetas que después se traducirán.

La carpeta *log* nos almacenará los resultados de la consola cuando intente compilar la aplicación y traducir el código utilizando AlterNative.

Por último, la carpeta *output* se compondrá del resultado de la traducción, que se mostrará en la parte derecha de la página web.

De esta manera, cada sesión de cada uno de los usuarios tendrá su propio directorio con los archivos que esté utilizando. Cuando la sesión expire, deberán eliminarse.

### 3.5.4 Edición de archivos

Para la edición de archivos, utilizaremos el módulo *fs* (del que hemos hablado anteriormente en el apartado 3.1.4). Éste módulo de Node.js nos va a permitir tratar con archivos y carpetas.

Con unos ejemplos de su uso será más fácil ver de qué se trata. Pero antes, recordemos que al inicio del controlador deberemos requerir el módulo *fs* de la siguiente manera:

```javascript
var fs = require('fs');
```

De esta manera, cada vez que usemos la variable *fs*, nos estaremos refiriendo al módulo de almacenamiento de archivos.

Vemos los ejemplos:

#### 3.5.4.1 Creación de un archivo

Para crear un nuevo archivo necesitamos llamar al siguiente método de *fs*:

```javascript
fs.writeFile(filename, data, [options], callback);
```

Necesitamos *filename*, que contendrá el nombre y la ubicación del archivo. Además, el contenido de dicho archivo, que es *data*. Y por último *callback*, que es una función que se ejecutará después del método. Su primer argumento se reserva para una excepción, y si la operación ha sido satisfactoria este argumento será nulo o indefinido.
 Debemos tener en cuenta que si la ruta donde queremos crear el archivo no existe, nos dará error. Por este motivo más abajo se explican formas de comprobar si existe un directorio o archivo.

3.5.4.2 Eliminación un archivo

Para eliminar un archivo, sin embargo, únicamente necesitaremos `filename`.

```javascript
fs.unlink(filename, callback);
```

Como podemos ver, `callback` es un argumento necesario siempre. Definiremos un `callback` específico para cada tipo de función.

3.5.4.3 Edición de un archivo

Para editar el contenido de un archivo, eliminaremos el archivo y volveremos a crearlo con el contenido nuevo.

```javascript
fs.unlink(filename, callback);
fs.writeFile(filename, newdata, [options], callback);
```

3.5.4.4 Lectura de un archivo

Definiremos una variable que nos almacenará el contenido de un archivo cuando lo leamos con la función `readFile`.

```javascript
var data = fs.readFile(filename, [options], callback);
```

3.5.4.5 Copia de un archivo

Para copiar un archivo, lo leeremos y crearemos un archivo con el mismo contenido y un nombre nuevo.

```javascript
var data = fs.readFile(filename, [options], callback);
fs.writeFile(newfilename, data, [options], callback);
```

3.5.4.6 Renombra de un archivo

Existe la función `rename` para renombrar un archivo.
En este caso necesitaremos `oldfilename` que contendrá la ruta y el nombre del archivo, y `newfilename`, que se formará por la ruta y el nuevo nombre del archivo.

### 3.5.4.7 Existencia de un archivo o carpeta

Hay veces que necesitamos saber si un archivo o carpeta existe antes de abrirlo o leer su contenido. Por este motivo, `exists` es muy útil, ya que nos devuelve `true` o `false` en función de si existe o no.

```javascript
fs.exists(path, callback);
```

El parámetro necesario `path` será la ruta del archivo o carpeta cuya existencia necesitemos comprobar.

### 3.5.4.8 Comprobación del estado

Además, también podemos ver algo parecido al estado de un elemento con la siguiente función:

```javascript
var stat = fs.stat(path, callback);
```

Si queremos comprobar si el elemento es una carpeta, podremos hacerlo de la siguiente manera:

```javascript
stat.isDirectory();
```

El resultado volverá a ser `true` o `false` en función de si es o no una carpeta.

Si por el contrario, queremos comprobar si es un archivo, podemos suponer que si `isDirectory()` nos ha dado falso tenemos un archivo, o simplemente comprobarlo así:

```javascript
stat.isFile();
```

### 3.5.4.9 Creación de una carpeta

La creación de una carpeta es similar a la de un archivo, pero esta vez la función es `mkdir`, y en vez de `filename` con ruta y nombre únicamente necesitamos la
ruta en la que irá la carpeta (con su nombre incluido), y en este caso el argumento se llamará path.

```javascript
fs.mkdir(path, [mode], callback);
```

Debe tener en cuenta que si la carpeta donde queremos ubicar otra carpeta no existe, nos dará error. Por este motivo a veces vamos a necesitar comprobar la existencia de algún elemento al que queramos acceder.

### 3.5.4.10 Lectura del contenido de una carpeta

Para leer el contenido de una carpeta utilizaremos a función readdir.

```javascript
var content = fs.readdir(path, callback);
```

El *content* o contenido que nos devuelve, es un array de nombres de carpetas y archivos dentro de ducho directorio.

### 3.5.4.11 Copia de una carpeta

Para copiar una carpeta, como es lógico, leeremos su contenido y después lo copiaremos en el directorio que escojamos. Pero para empezar, crearemos el directorio destino de nuestra copia de la siguiente manera;

```javascript
fs.mkdir(newfolderpath, [mode], callback);
recursiveCopy(oldfolderpath, newfolderpath);
```

Como podemos ver, para copiar todo el contenido, llamamos a la función `recursiveCopy`. Esta función no existe dentro de *fs*, por lo que habrá que crearla:

```javascript
var recursiveCopy = function(src, dst){
    content = fs.readdir(src, callback);
    content.forEach(function (f){
      if (fs.stat(src+f, callback).isDirectory()){
        if(!fs.exists(dst+f, callback))
          fs.mkdir(dst+f, callback);
        recursiveCopy(src+f+'/'+dst+f+'/'+');
      }
      else{
        if(fs.exists(dst+f, callback))
          fs.unlink(dst+f, callback);
        fs.writeFile(dst+f, fs.readFile(src+f, callback), callback);
      }
    });
}
```
Básicamente, *recursiveCopy* lee el contenido de la carpeta origen y va copiando los archivos y creando carpetas nuevas, usando todos los elementos vistos anteriormente.

### 3.5.4.12 Renombra de una carpeta

De la misma manera que renombramos un archivo, también podemos hacerlo con una carpeta:

```javascript
fs.rename(oldpath, newpath, callback);
```

### 3.5.4.13 Eliminación de una carpeta

Para eliminar una carpeta utilizaremos la función contraria al `mkdir`, que es `rmdir`. Únicamente necesitaremos el `path` o ruta de la carpeta.

```javascript
fs.rmdir(path, callback);
```

### 3.5.5 Traducción de código

Para la traducción de código ejecutaremos AlterNative desde el símbolo de sistema. El comando a utilizar es el siguiente:

```bash
alternative MyExe.exe ./output/
```

Dónde `MyExe.exe` es la entrada binaria a traducir y `.output/` es la carpeta dónde se crearan todos los archivos y carpetas resultantes.

### 3.5.5.1 Uso del terminal desde Node.js

Para utilizar el terminal o símbolo de sistema desde el servidor de Node.js, vamos a necesitar un proceso hijo, que inicializaremos como lo hacemos con los módulos:

```javascript
var terminal = require('child_process').spawn('cmd');
```

Para escribir en el terminal, utilizaremos la función `write`, y para que se ejecute el comando necesitaremos un cambio de línea (su sintaxis es “\n”).

```javascript
terminal.write('command\n');
```
3.5.5.2 Creación del archivo .exe

Para crear el archivo .exe que necesita AlterNative para traducir, utilizaremos el csc (CSharp Compiler en inglés, es decir, el compilador de lenguaje C#). El comando a utilizar es:

```
csc /optimize /out:MyExe.exe /recurse:input/*.cs
```

Como podemos apreciar, detrás del comando csc tenemos diferentes opciones:

- La opción `optimize` habilita/deshabilita optimizaciones añadiendo +/- detrás, respectivamente. Cuando no se especifica nada detrás, significa que queremos construir un módulo para ser utilizado como ensamblador, en ese caso usa los mismos ajustes que para un binario.
- La opción `out` define dónde se almacenará el archivo resultante del compilado, además de su nombre.
- La opción `recurse` busca archivos fuente a compilar en los subdirectorios de una carpeta, en este caso de la carpeta `input`, que es donde se encuentra todo el código que queremos traducir. Cuando definimos `*.cs` hacemos que coja cualquier archivo con esa extensión.

Una vez sabemos el comando a utilizar, veamos cómo se ve llamado desde el servidor como hemos explicado anteriormente:

```
terminal.write('csc /optimize /out:MyExe.exe /recurse:input/*.cs
');
```

3.5.5.3 Traducción

Al principio del apartado hemos visto el comando que utilizamos para realizar la traducción con AlterNative.

Sin embargo, necesitamos registrar la salida de la consola para ver si hay algún error. Para ello escribiremos en un archivo que guardaremos en la carpeta `log` que hemos visto anteriormente (apartado 3.5.3, después de la Fig. 3.21: Carpetas del usuario.). Necesitaremos redirigir el resultado del terminal de la siguiente manera:

```
terminal.write('alternative MyExe.exe ./output/ > out.log 2>> err.log &
');
```

De esta forma hacemos que la primera salida del terminal (>) se redirija al archivo `out.log` y la segunda salida (2>>, que es la de los errores, se redirija al archivo
err.log. La & nos indica que el comando se ejecutará en segundo plano, de esta manera el funcionamiento del resto de comandos no dependerán de este.

3.5.6 Descarga de resultados

Al descargar los resultados descargaremos un archivo .zip con la traducción de AlterNative y un binario de la aplicación resultante. Como en la carpeta output del usuario ya tenemos los resultados, únicamente nos falta construir el binario para poder adjuntar todos los archivos en un .zip.

Para crear el binario que nos falta, utilizaremos el comando make de AlterNative. Para ello necesitaremos hacer los siguientes pasos:

```
cd output
mkdir build
cd build
cmake ..
cd ..
cd ..
alternative make /output
```

El último paso es crear el zip, llenarlo y enviarlo al cliente para que se lo descargue directamente. Con el siguiente código lo conseguiremos:

```
var zip = new require('node-zip')();
var zipFilled = fillZip(zip, sourcepath, destpath);
var data = zipFilled.generate({base64:false, compression:'DEFLATE'});
fs.writeFile(path, data, 'binary', callback);
response.download(path);
```

La función `fillZip` se define a continuación:

```
var fillZip = function(zip, src, dst){
    content = fs.readdirSync(src);
    content.forEach(function (f){
        if (fs.statSync(src+f).isDirectory()){
            zip.folder(dst+f);
            fillZipFile(zip, src+f+'/', dst+f+'/');
        } else{
            zip.file(dst+f, fs.readFileSync(src+f));
        };
    });
    return zip;
}
```

Como podemos ver, `fillZip` guarda mucha relación con la función `recursiveCopy` utilizada para copiar archivos. En este caso estamos llenando un archivo zip, por lo que usaremos `zip.folder` o `zip.file` en vez de `fs.mkdir` y `fs.writeFile`. 
4 CONCLUSIONES Y LÍNEAS FUTURAS

4.1 Objetivos cumplidos

El objetivo principal de este proyecto era la traducción online de aplicaciones utilizando AlterNative. Para conseguirlo, hemos visto el desarrollo de un servidor http con Node.js y el uso de todos los módulos necesarios para completar las funcionalidades que marcábamos en un primer momento.

Otro requisito principal era la subida de un archivo y la decompilación del mismo. El trabajo necesario para realizarlo fue modificar AlterNative para recuperar la primera parte de su traducción, la decompilación de binario a C#.

Además, necesitábamos descargar un archivo .zip con los archivos traducidos y un ejecutable. CMake nos ayuda en este cometido, facilitando la creación del ejecutable.

4.2 Líneas futuras

Como cosas a mejorar podemos hablar de algunos puntos que han quedado pendientes y otros que han ido surgiendo durante el desarrollo y que no ha habido tiempo de implementar.

- **Eliminación de archivos de sesiones caducadas.** Lo normal cuando una sesión expira es eliminar los datos. Nuestro servidor elimina los datos de sesión y las cookies, pero no elimina todavía los archivos de sesión.

- **Aviso de uso de cookies al usuario.** Con la nueva Ley de Cookies en España, habría que implementar un aviso a los usuarios de que utilizamos cookies para mantener abierta una sesión. Sobretodo habrá que hacerlo antes de subir la web a un servidor público.

- **Ejecutar AlterNative a tiempo real.** Estaría interesante comprobar si la aplicación es compilable mientras el usuario la está editando. Cada vez que el servidor ve que puede compilar, ejecutará la traducción de código para que puedan verse los cambios a tiempo real.

- **Portabilidad multiplataforma.** Aunque todos los elementos usados son multiplataforma (AlterNative, CMake, Node.js), las convenciones de nombres de los sistemas operativos cambian y el servidor no está preparado para ello, por ejemplo, en Windows el terminal es “cmd” y en los sistemas basados en Unix como Linux o MacOS es “bash”. La idea es adaptar el servidor para que funcione en cualquier plataforma.
4.3 Conclusiones personales

En el desarrollo del proyecto he adquirido conocimientos nuevos sobre el mundo de las páginas web.

Al principio apenas sabía hacer una página web con estilos y scripts, y actualmente tengo bastante soltura a la hora de hacerlo. Además, utilizar JavaScript en el lado del servidor fue un concepto completamente nuevo, y todo lo relacionado con Node.js me resultaba desconcertante antes de empezar.

Puedo afirmar que he utilizado tecnologías y herramientas que jamás había tocado, y que me siento muy orgullosa de los resultados obtenidos. Pero lo más importante es que he mejorado en la elaboración de proyectos y he ganado en capacidad de adaptación y aprendizaje de nuevas tecnologías.
5 BIBLIOGRAFÍA Y REFERENCIAS


6 ANEXO

6.1 Rutas del servidor

Como podemos ver, para gestionar las peticiones de los clientes, el servidor llama a funciones que están dentro de los dos controladores principales: `fileStorage.js`, es el controlador que lleva toda la parte de gestión de archivos; y `compiler.js` en cambio, lleva todo lo que es compilación y traducción del código.

En el apartado “3.5.4 Edición de archivos” encontramos todo lo necesario para la gestión de los archivos, en “3.5.5 Traducción de código” vemos el uso de la consola de comandos, la compilación y traducción del código, y en el “3.5.6 Descarga de resultados” se explica cómo se consigue la creación de un archivo .zip. Con toda esta información se han creado los controladores del servidor.

6.2 Uso de CodeMirror

```javascript
// CONFIGURACIÓN DE CODEMIRROR
var codeConsole = document.getElementById("code");
CODE = CodeMirror.fromTextArea(codeConsole, {
    lineNumbers: true, extraKeys: {
        "Ctrl-Space": "autocomplete"}, readOnly: false,
    mode: {name: "text/x-csharpe", globalVars: true}});
codeConsole = document.getElementById("code2");
CODE2 = CodeMirror.fromTextArea(codeConsole, {
    lineNumbers: true, readOnly: true, mode: "text/x-c++src"});
```

Fig. 6.1: Rutas del servidor.

Fig. 6.2: Código para la creación y configuración de CodeMirror.
El CODE se corresponde con el editor de texto de la izquierda, por este motivo tiene el autocompletado activado y C# como lenguaje de programación predeterminado.

CODE2 se corresponde con el visor de texto de la derecha. Como vemos no se puede editar y tiene C++ configurado.

### 6.3 Uso de jQueryFileTree

```javascript
//CARGAR LOS EXPLORADORES DE ARCHIVOS
var loadFolders1 = function() {
    $('.folders1').fileTree({
        root: '../uFolder/input/', script: 'uFolder', folderEvent: 'click', function(file) {
            openFile(file, 'input');
        }
    });
}

var loadFolders2 = function() {
    $('.folders2').fileTree({
        root: '../uFolder/output/', script: 'uFolder', folderEvent: 'click', function(file) {
            openFile(file, 'output');
        }
    });
}
```

Fig. 6.3: Código para la creación de los exploradores de archivos.

Como pasaba antes, en este caso el 1 corresponde con la parte izquierda y el 2 con la derecha. Para inicializarlos, el JavaScript de la web llamará a las funciones `loadFolders1` y `loadFolders2` una vez cargada la página.

### 6.4 Subida de un archivo

Es quizás una de las partes menos triviales del controlador `fileStorage.js`, por este motivo existe este apartado en el Anexo.

Como hemos visto en las rutas del servidor, cuando el cliente haga la petición /upload significará que estará subiendo un archivo al servidor. El proceso que tendrá que hacer el servidor en este momento es recuperar el archivo del cliente y decompilar el ejecutable, almacenándolo en la carpeta de sesión del cliente.
Para decompilar el binario subido, llamamos a la función `decompile` del otro controlador. Esta función ejecutará un comando de AlterNative que permitirá volcar el decompilado en la carpeta del usuario. El código es el siguiente:

```javascript
var _decompile = function(filename, sessionID)
    terminal = require('child_process').spawn('cmd');
    terminal.stdout.on('data', function (data) {
        console.log('stdout: ' + data);
    });
    t = terminal.stdin;
    //iniciamos alternative
    t.write('cd C:/Alter-Native/\n');
    t.write('alternative-init.bat\n');
    t.write('cd '+__dirname+'/'+users+'/'+sessionID+'\n');
    require(__dirname+'/'+fileStorage.js).empty(path+sessionID+'/input');
    require(__dirname+'/'+fileStorage.js).empty(path+sessionID+'/output');
    var name = filename.substring(0, filename.length - 4);
    //decompilamos con el terminal
    t.write('alternative '+ filename + ' input/-name+ -s\n');
    //salimos del terminal
    t.write('exit\n');
}
```

Fig. 6.5: Código para decompilar un archivo.

Como podemos ver, antes de decompilar el archivo, vaciaremos los directorios de entrada y salida, es decir, los códigos que actualmente el usuario tenía en el `workspace`, pues nos interesa tener únicamente el nuevo programa.