Revista Tecnocientífica URU
Universidad Rafael Urdaneta
No. 28 Enero - Junio 2025
Depósito Legal: PPI 201402ZU4464
ISSN: 2343-6360

Desarrollo de un marco de trabajo basado en componentes para la construcción de clientes web: Slice.js

Development of a component based framework for the construction of web clients: Slice.js

Julio A. Graterol-Bracho
Universidad Rafael Urdaneta, Facultad de Ingeniería, Escuela de Ingeniería en Computación.
Maracaibo, Venezuela.
https://orcid.org/0009-0006-9943-7182 | Correo electrónico: juliograterolb@gmail.com
Victor J. Kneider-Al Nahi
Universidad Rafael Urdaneta, Facultad de Ingeniería, Escuela de Ingeniería en Computación.
Maracaibo, Venezuela.
https://orcid.org/0009-0004-0434-430X | Correo electrónico: victorkneider@gmail.com
Jubert J. Pérez-Zabala
Universidad Rafael Urdaneta, Facultad de Ingeniería, Escuela de Ingeniería en Computación.
Maracaibo, Venezuela.
https://orcid.org/0009-0003-8615-9927 | Correo electrónico: jubert.perez.10233@uru.edu
Recibido: 08/06/2025 Admitido: 05/07/2025 Aceptado:11/07/2025

Resumen

La presente investigación introduce el desarrollo de un marco de trabajo (framework) basado en componentes, ofreciendo una arquitectura intuitiva para la construcción efectiva y organizada de clientes web. Se empleó una metodología de investigación descriptiva con diseño de campo no experimental, analizando el proceso de desarrollo sin intervenir las condiciones naturales del fenómeno estudiado. Para el desarrollo del Framework se utilizó una metología de cascada o de fases secuenciales, siguiendo las seis fases tradicionales del desarrollo de Software: requerimientos, análisis, diseño, implementación, pruebas y mantenimiento. Destacando por su facilidad de implementación y enfoque en la retrocompatibilidad, el framework proporciona una sólida base de componentes desacoplados para los desarrolladores al igual que una sólida estructura de componentes y herramientas funcionales que facilitan el proceso de desarrollo de aplicaciones web.

Palabras clave: Framework, desarrollo, aplicaciones, componentes, desarrolladores.

Abstract

The present research introduces the development of a component-based framework, offering an intuitive architecture for the effective and organized construction of web clients. A descriptive research methodology with non-experimental field design was employed, analyzing the development process without intervening in the natural conditions of the studied phenomenon. For the framework development, a waterfall or sequential phases methodology was used, following the six traditional phases of software development: requirements, analysis, design, implementation, testing, and maintenance. Highlighting its ease of implementation and focus on backward compatibility, the framework provides a solid foundation of decoupled components for developers, along with a robust structure of components and functional tools that facilitate the web application development process.

Key words: *Framework, development, application, components, developers.*

Introducción

El desarrollo de aplicaciones web implica crear programas que pueden ser accesibles a través de navegadores web. La complejidad de una aplicación web generalmente está relacionada con la cantidad

de funcionalidades que tiene y el tiempo y esfuerzo necesarios para desarrollarla. Con el tiempo, los desarrolladores web han identificado ciertas funcionalidades comunes que se repiten en muchas aplicaciones. Para evitar tener que escribir el mismo código una y otra vez, comenzaron a crear herramientas y bibliotecas de código que pueden ser reutilizadas en diferentes proyectos. Estas herramientas y bibliotecas a menudo se agrupan en lo que se conoce como frameworks web.

Los frameworks web son conjuntos de herramientas y bibliotecas predefinidas que facilitan el desarrollo de aplicaciones web al proporcionar estructuras y patrones de diseño comunes. Esto permite a los desarrolladores construir aplicaciones de manera más eficiente y rápida, ya que no tienen que empezar desde cero cada vez. Sin embargo, existen algunos desafíos con los frameworks web existentes. Uno de los problemas principales es la inestabilidad y la dependencia de estos frameworks. Los frameworks web a menudo se actualizan con frecuencia, lo que puede llevar a que los proyectos existentes se vuelvan obsoletos si no se actualizan. Además, las actualizaciones pueden introducir cambios que requieren modificaciones en el código existente, lo que puede ser costoso en términos de tiempo y recursos. Además, algunos frameworks web pueden cambiar a modelos de pago, lo que podría limitar el acceso de los desarrolladores a herramientas y recursos esenciales.

Como antecedente, la investigación de Mármol y Pérez [1] titulado: "Desarrollo de un marco de trabajo con node. js basado en componentes para el manejo de solicitudes a objetos de negocios embebidos en el backend", desarrolló una interfaz única y sencilla para el acceso y ejecución de lógica pertenecientes a objetos de negocios, a través de un único endpoint web.

Por su parte, Björemo y Trninć [2] en su trabajo sobre: "Evaluación de bases de aplicaciones web con respecto al desarrollo rápido", realizaron una investigación acerca de los diferentes frameworks web utilizados en la época, generando una robusta documentación de las comparaciones realizadas entre los mismos, tomando en cuenta la dificultad de la curva de aprendizaje y la dificultad de la aplicación web de cada uno de ellos.

Para abordar los problemas anteriormente mencionados, se propone el desarrollo de un framework web propio basado en componentes. Este enfoque tiene como objetivo proporcionar a los desarrolladores una herramienta robusta y confiable para construir aplicaciones web, mientras se asegura la estabilidad y el control directo sobre el proceso de desarrollo. Al tener un framework propio, los desarrolladores pueden tener más control sobre las actualizaciones y no sujetos a los cambios y restricciones de los frameworks externos.

Fundamentos Teóricos

HTML: El lenguaje de Marcado de Hipertexto o Hypertext Markup Language es el bloque de construcción más básico de la Web. Define el significado y la estructura del contenido de una página web. "Hipertexto" se refiere a los enlaces que conectan las páginas web entre sí, ya sea dentro de un mismo sitio web o entre sitios web diferentes. El bloque de construcción definido por HTML son los elementos, los cuales describen o definen diferentes partes del contenido, como texto, imágenes, enlaces, listas, etc. Estos elementos están delineados por etiquetas, que indican dónde comienza y dónde termina un elemento en el código HTML.

CSS: CSS (Cascading Style Sheets) es un lenguaje de hojas de estilo utilizado para describir la presentación y el formato de un documento HTML o XML. Proporciona un conjunto de reglas y propiedades que permiten controlar la apariencia visual de los elementos en una página web, incluyendo el diseño, colores, fuentes, márgenes y otros aspectos visuales.

Javascript: JavaScript (JS) es un lenguaje de programación ligero interpretado (o compilado en tiempo real) con funciones de primera clase. JavaScript es un lenguaje basado en prototipos, multiparadigma, de un solo hilo, y dinámico, que admite estilos de programación orientados a objetos, imperativos y declarativos (por ejemplo, programación funcional) (Mozilla Foundation). Las normas por las que se rige son Especificación del Lenguaje ECMAScript (ECMA-262) y la especificación internacional API ECMAScript (ECMA-402).

Web Components API: Los web components son un conjunto de APIs (Interfaces de Programación de Aplicaciones) de la plataforma web que te permiten crear nuevas etiquetas HTML personalizadas, reutilizables

y encapsuladas para utilizar en páginas web y aplicaciones web. Los componentes y widgets personalizados, construidos según los estándares de los Componentes Web, funcionarán en navegadores modernos y pueden ser utilizados con cualquier biblioteca o marco de JavaScript que trabaje con HTML.

Metodología

Se realizó una investigación de tipo descriptiva, donde se detalló el proceso de desarrollo de un framework web, desde la determinación de las características, hasta la construcción del mismo. Este enfoque implicó un análisis detallado, el cual no buscó exclusivamente identificar los problemas, sino también busca comprenderlos para la solución propuesta. El diseño de la investigación se basó en un enfoque de campo o no experimental, donde no se intervienen ni alteran las condiciones en las que existe el fenómeno estudiado [3], en su lugar, se centra en la recolección de datos de la realidad del desarrollo de aplicaciones web y la experiencia de los desarrolladores. La presente utiliza al propio framework en desarrollo como unidad de investigación, para una comprensión detallada de cada etapa del proceso de desarrollo, evaluando así, en tiempo real, la eficiencia y el rendimiento del Framework mediante pruebas específicas.

En relación con las fases de desarrollo, se adoptó un enfoque metodológico conocido como "metodología de la cascada", el cual se caracterizó por su estructura lineal y secuencial, donde las distintas fases del proceso se suceden de manera ordenada y cada una de ellas debe ser completada antes de avanzar a la siguiente. Esto proporcionó una planificación exhaustiva desde las etapas iniciales del proyecto, estableciendo objetivos claros y recopilando detalladamente los requisitos. La metodología de la cascada ofreció una previsibilidad en el proceso, facilitando la gestión y la evaluación progresiva del avance del proyecto.

Fases de desarrollo

Durante esta fase, se investigó activamente acerca de las funcionalidades que el framework debe ofrecer, así como los requisitos técnicos y operativos que deben cumplirse para garantizar su eficacia y funcionalidad. Esto implicó explorar las necesidades y expectativas de los usuarios, así como las demandas del entorno tecnológico en constante evolución.

Resultados

Fase 1. Requerimientos:

Tabla 1. Requerimientos funcionales

Proporcionar un conjunto completo de componentes web reutilizables.
Permitir la fácil personalización y extensión de los componentes.
Ser compatible con los estándares web modernos (HTML5, CSS3, JavaScript ES6, etc.).
Ofrecer una experiencia de usuario consistente y atractiva en diferentes navegadores y dispositivos.
Admitir la creación de interfaces de usuario responsivas y adaptables.
Proporcionar un mecanismo a los usuarios de creación de componentes propios para ser utilizados con el Framework.

Tabla 2. Requerimientos no funcionales

Manejar un rendimiento óptimo, minimizando los tiempos de carga y respuesta.
Seguir principios de código modular para facilitar la mantenibilidad y la escalabilidad del framework.
Ser fácil de aprender y utilizar para desarrolladores web de diferentes niveles de experiencia.
Mantener la compatibilidad hacia atrás con versiones anteriores del framework para garantizar una transición suave para los usuarios que actualizan desde versiones anteriores.
Mantener una documentación completa y actualizada que refleje con precisión las características, API y mejores prácticas del framework, para facilitar su adopción y uso por parte de los desarrolladores.

Fase 2. Análisis:

Durante esta etapa, se llevó a cabo un examen minucioso de los requerimientos identificados en la fase previa, con el objetivo de comprender a fondo las expectativas de los usuarios y las limitaciones del sistema. Se analizaron las funcionalidades esenciales que el framework debe ofrecer, así como los requisitos técnicos, operativos y de rendimiento que deben cumplirse para garantizar su eficacia y funcionalidad.

El framework desarrollado se basó en una estructura modular fundamentada en componentes, lo que ofreció una base sólida para la construcción de clientes web. Esta arquitectura modular permitió a los desarrolladores dividir la funcionalidad del framework en componentes independientes y reutilizables, facilitando su mantenimiento y extensión.

De manera similar, los clientes web se construyeron siguiendo el mismo enfoque basado en componentes, lo que implicó la creación de interfaces de usuario y funcionalidades de aplicación mediante la combinación y configuración de componentes predefinidos. Este proceso agilizó el desarrollo y promovió la reutilización de código.

Para lograr la separación de componentes en el código JavaScript, se utilizaron módulos de JavaScript, los cuales permitieron dividir el código en archivos separados con funcionalidades específicas, mejorando la organización y la modularidad del código. Además, se integró una clase principal al framework para centralizar la lógica y el funcionamiento de todos los componentes. Esta clase actuó como un punto central, permitiendo que los componentes consumieran funcionalidades entre sí mediante su instancia. Se le asignó el nombre "Slice" para distinguirla y facilitar la conexión entre los diferentes componentes.

En cuanto a la clasificación de los componentes, se dividieron en tres categorías principales: visuales, estructurales y servicios. Los componentes visuales fueron fundamentales para la construcción de interfaces de clientes web; mientras que, los estructurales proporcionaron la infraestructura necesaria para gestionar y supervisar todos los aspectos del framework y la aplicación.

Por otro lado, los servicios encapsularon la lógica de negocio de la aplicación y ofrecieron la capacidad de gestionar tecnologías asociadas con el cliente web, como solicitudes HTTP y almacenamiento local. Además, se incluyeron componentes visuales y servicios de usuario, que fueron creados por los desarrolladores para proyectos específicos utilizando el framework.

Fase 3. Diseño:

Para iniciar la fase de diseño, siguiendo con la clasificación realizada en la Fase de Análisis de los componentes según su categoría y funcionalidad, se encontró con:

Visuales (Visual): Estos componentes desempeñan un papel fundamental en la construcción de las interfaces de los clientes web. Se basan en etiquetas personalizadas de HTML y hacen uso de la Web Components API. Al combinar estas etiquetas con clases de JavaScript, se logra encapsular la lógica necesaria para manipular estos componentes de manera eficiente, permitiendo al desarrollador abarcar una amplia gama de funcionalidades según sus necesidades, desde simples cambios de estilo, como el color de la fuente, hasta la creación dinámica de nuevos componentes.

Además, estos componentes pueden interactuar con el backend a través de solicitudes HTTP, lo que les permite realizar operaciones como recuperar datos, enviar formularios o actualizar la interfaz de usuario en respuesta a eventos específicos.

Estructurales o De estructura (Structural): Estos componentes son esenciales para el funcionamiento integral del framework, ya que proporcionan la infraestructura necesaria para gestionar y supervisar todos los aspectos del mismo y de la aplicación a desarrollar. Dentro de esta categoría, se encuentran los componentes encargados del control de las instancias de los elementos, permitiendo un manejo eficiente y dinámico de los mismos. Además, ofrecen herramientas para el registro y seguimiento de actividades en tiempo real, lo

que facilita la depuración de errores y el análisis de rendimiento durante el desarrollo y la ejecución de la aplicación.

Por otro lado, los componentes estructurales también incluyen funcionalidades para la visualización y manipulación de propiedades de los elementos de la aplicación. Esto permite a los desarrolladores inspeccionar y ajustar el estado de los componentes según sea necesario, lo que resulta fundamental para la personalización y optimización de la aplicación. En conjunto, estos componentes proporcionan una base sólida y versátil para la construcción de aplicaciones robustas y eficientes, garantizando un control completo sobre la infraestructura del framework y la aplicación resultante.

Cabe resaltar que, aunque su categoría no es visual, algunos de estos componentes también utilizan la tecnología de la Web Components API para brindar a los desarrolladores interfaces gráficas que facilitan el proceso de desarrollo y de pruebas de la aplicación. Los componentes de categoría estructural no pueden ser instanciados a través del método build de la instancia principal.

Las tecnologías de componentes web o Web Components API permiten crear sus propios elementos HTML. Son un conjunto de tecnologías complementarias para encapsular HTML, javascript y estilos en paquetes reutilizables, con soporte nativo disponible en el navegador [4].

Servicios (Service): Los servicios constituyen un tipo fundamental de componente que desempeña un papel clave en el desarrollo de aplicaciones web. Su principal función radica en encapsular la lógica de negocio de la aplicación, lo que facilita la reutilización del código y mejora la legibilidad y efectividad del proceso de desarrollo.

Además de abordar la lógica específica de la aplicación, los servicios ofrecen la capacidad de encapsular y gestionar diversas tecnologías asociadas con el cliente web, como el manejo de solicitudes HTTP, el almacenamiento local (localStorage), y el acceso y modificación de bases de datos indexadas (indexedDB), entre otros.

A pesar de no hacer uso directo de la Web Components API, los servicios juegan un papel crucial al permitir la creación dinámica de componentes visuales que sí la utilizan, así como la modificación dinámica del documento en función de los métodos de estos servicios. Esto proporciona una flexibilidad adicional en el desarrollo de la interfaz de usuario y la manipulación dinámica del DOM, lo que contribuye significativamente a la versatilidad y capacidad de respuesta de la aplicación web resultante.

Visuales de Usuario (UserVisual): Cuentan con las mismas características que los componentes de categoría “Visual”, pero difieren con estos en que estos son creados por los desarrolladores para sus proyectos específicos mientras que, los de categoría “Visual” son creados por los desarrolladores del Framework.

Servicios de Usuario (UserService): Al igual que los componentes visuales de usuario, poseen la característica de que son creados por los desarrolladores para sus proyectos específicos utilizando el Framework y cuentan con las mismas características que los componentes de categoría “Service” o servicios.

Seguidamente, se realizó un diagrama de los componentes estructurales que constituyen la base sobre la que trabaja el framework (Figura 1), junto con un archivo de configuración que controla el comportamiento de estos componentes. La instancia principal del framework, llamada "Slice", estuvo compuesta por instancias de los diferentes componentes estructurales, permitiendo el acceso a través de esta instancia.

Sin embargo, se consideró necesario que el servicio “Translator” fuera añadido en los diagramas que componen la estructura del framework, dado que la habilitación del mismo se puede realizar desde la configuración del framework para brindar soporte a páginas multilenguajes y así evitar crear múltiples instancias del mismo.

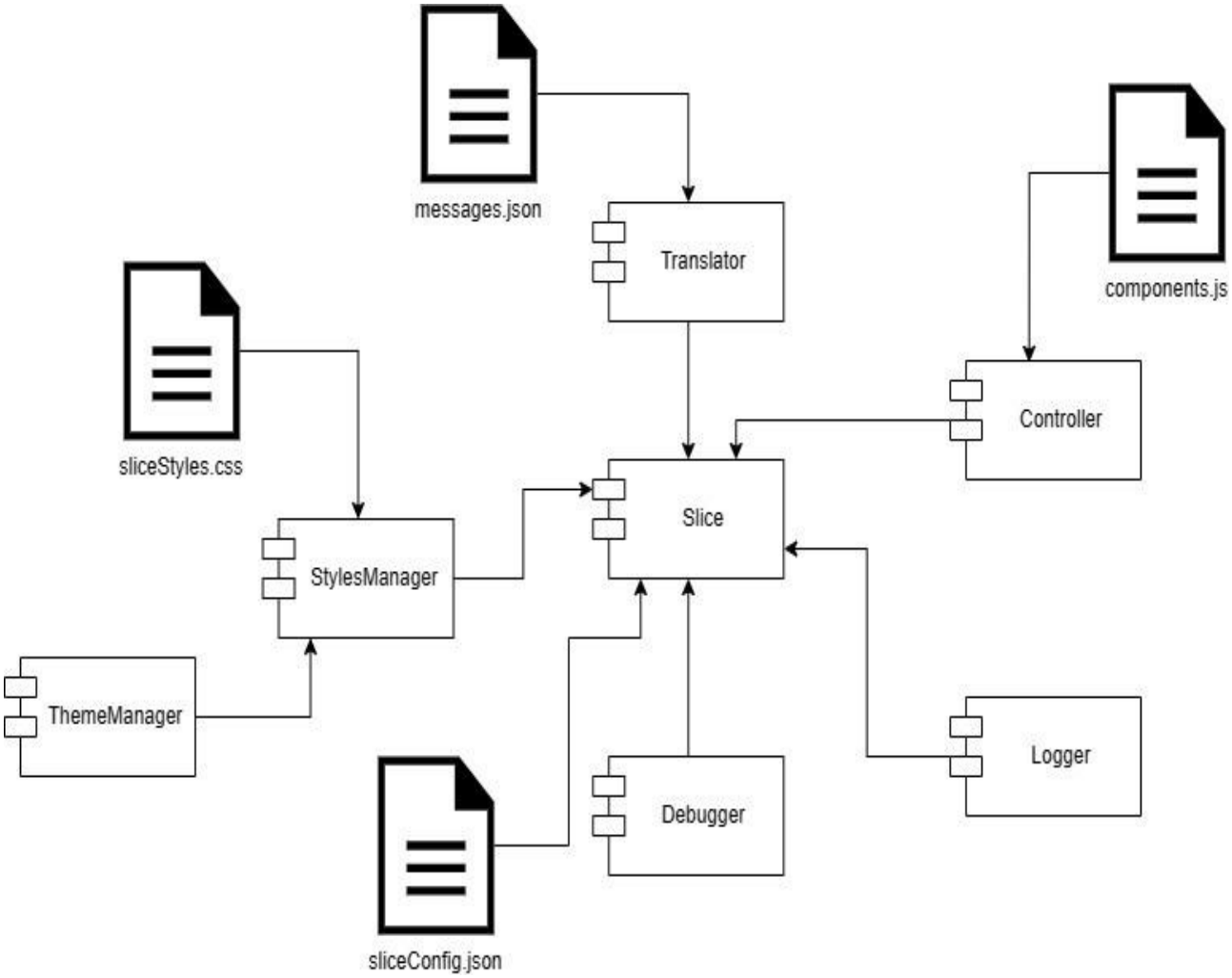


Figura 1. Diagrama de componentes del modelado

Se definió un archivo de configuración llamado “sliceConfig”, que permitió a los desarrolladores manejar el comportamiento del framework habilitando o deshabilitando funcionalidades de los diferentes componentes. Por ejemplo, los componentes de depuración como “Logger” o “Debugger” pueden ser necesarios durante la construcción de una aplicación, pero no durante el lanzamiento.

En este archivo se permitió a los mismos, utilizar la ruta de su preferencia, aunque la iniciación del proyecto del framework siguió una organización de ruta funcional sugerida por los autores desarrolladores del framework (Figura 2).

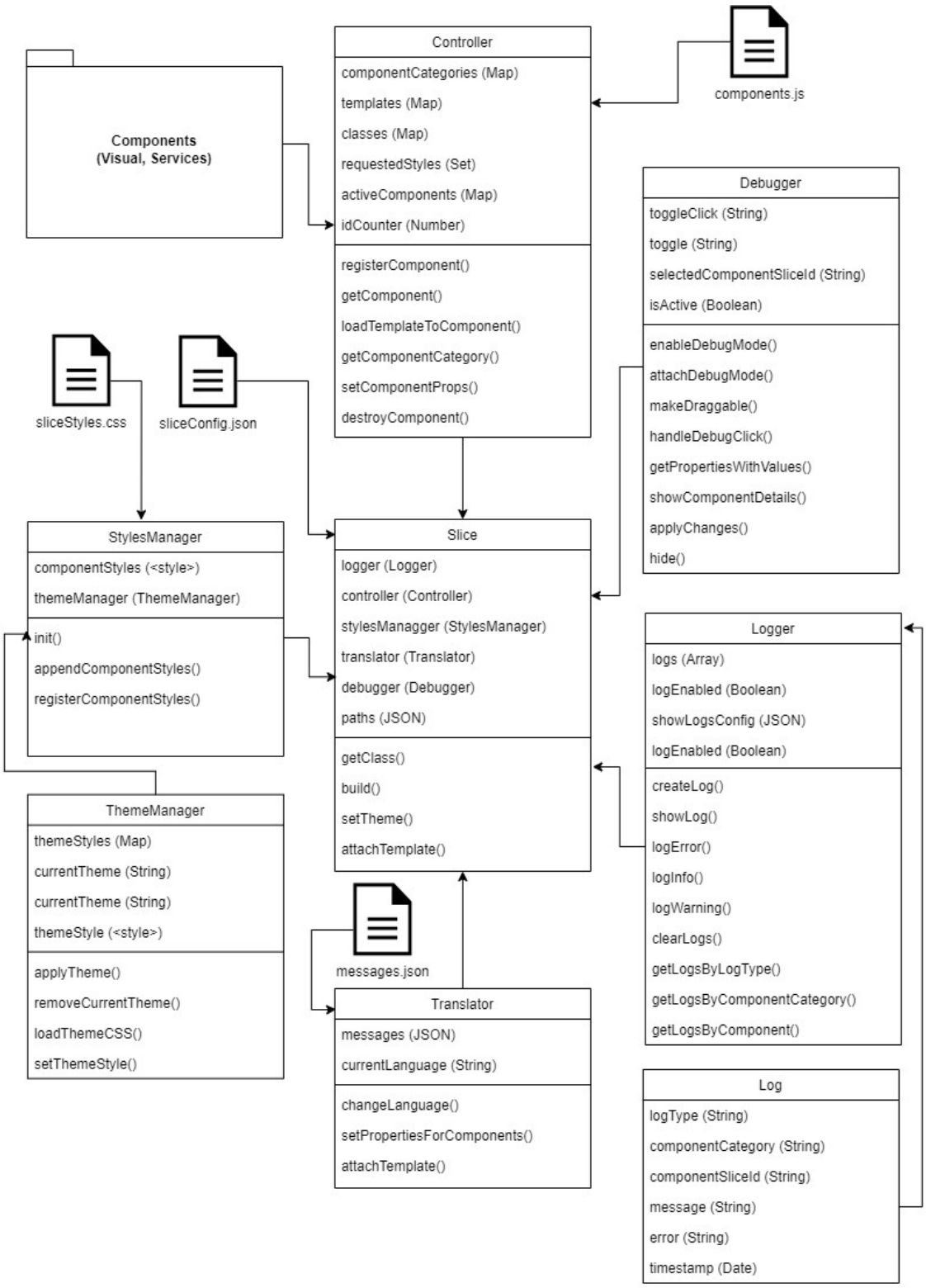


Figura 2. Diagrama de clases

Fase 4. Implementación:

Para comenzar la fase de construcción del framework basado en componentes, es esencial abordar la definición de la estructura y organización de archivos. Esta etapa fue fundamental para establecer los cimientos de un proyecto coherente y sostenible, garantizando que el código fuera legible y comprensible para

todos los desarrolladores involucrados. Una estructura de archivos bien definida es crucial para la eficacia y la longevidad a lo largo del tiempo de un proyecto del framework (Figura 3).

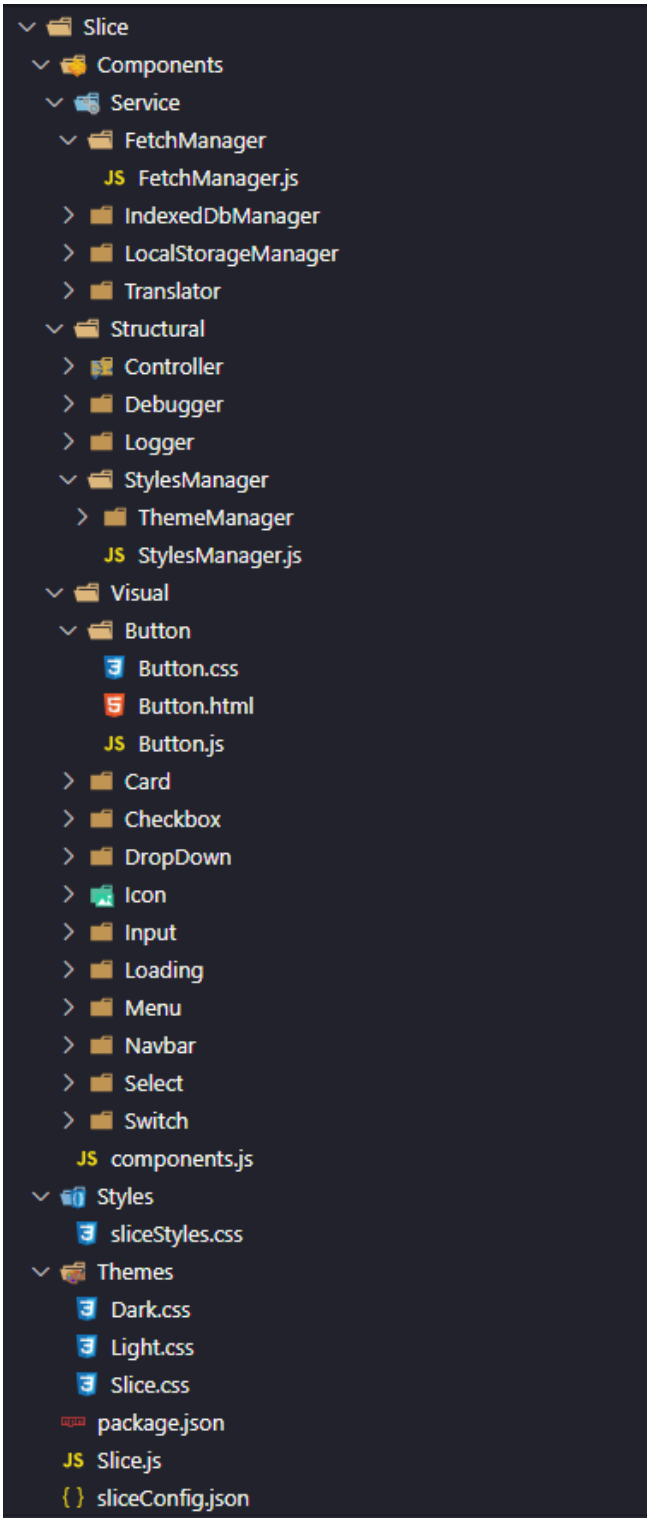


Figura 3. Arquitectura de carpetas del Framework.

Como se muestra en la figura anterior, la arquitectura de carpetas diseñada para el framework basado en componentes, denominado Slice, se organizó de manera jerárquica ascendente, donde la carpeta principal fue denominada “Slice”, la cual contiene todos los archivos correspondientes a la construcción del framework, esta carpeta en sí misma, está compuesta por el archivo de configuración del framework “sliceConfig.json”, el

archivo “Slice.js” de la instancia principal del framework, y una subcarpeta denominada “Components”. En dicha carpeta, la carpeta donde residen todos los archivos relacionados con los componentes del framework, organizó subcarpetas para cada categoría de componentes registrada en el framework, como Visual, Structural y Service. Además, contuvo el archivo "components.js" que listó todos los componentes disponibles en el framework.

En la carpeta “Visual” se encontró la carpeta de cada componente de esta categoría. Cada una de las carpetas de los componentes contuvo tres archivos esenciales para la creación del componente: un archivo “.js” para el código JavaScript, un archivo “.html” para el template HTML y un archivo “.css” para los estilos asociados.

En la carpeta “Service” cada carpeta de componente tuvo un archivo “.js” que definió la lógica del servicio correspondiente.

En la carpeta “Structural”, la última categoría de componentes, se encontraron las carpetas para cada componente de la categoría Structural, que contuvieron los archivos vinculados a esos componentes. Los componentes de tipo Structural tuvieron al menos un archivo ".js", pero también pueden tener archivos ".html" y ".css" en caso de tener una interfaz gráfica. Los componentes de esta categoría que no pueden ser instanciados mediante el método "build" se almacenaron juntos para una mejor organización.

En la carpeta “Styles” se almacenaron los estilos relacionados con el framework. Incluyó el archivo esencial “sliceStyles.css” que configuró aspectos fundamentales de los estilos de los componentes, como tipografías y grosores de bordes.

Según Meyer [5], CSS permite separar la estructura del contenido de un documento web de su presentación visual, lo que facilita la creación y el mantenimiento de sitios web. Esta tecnología es la que permitió manejar los estilos mencionados previamente.

Por último, en la carpeta “Themes” se guardaron los archivos correspondientes a los temas visuales utilizados en el desarrollo del framework. Cada tema se representó mediante un archivo ".css". Además de los temas predefinidos como "Light" (Claro), "Dark" (Oscuro) y "Slice" (estilos por defecto del framework), los desarrolladores pueden agregar temas personalizados según las necesidades del proyecto.

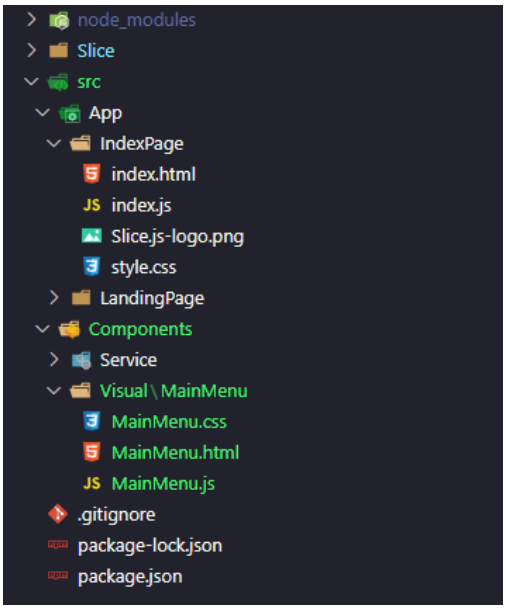


Figura 4. Arquitectura de la carpeta “src”.

La carpeta “src” se encontró en la raíz del proyecto, junto con la carpeta principal “Slice”. Su propósito fue contener los archivos necesarios para la construcción de las aplicaciones web. Dentro de “src”, se encontraron las carpetas “App” y “Components”.

La carpeta “App” contuvo todos los archivos relacionados con las diferentes vistas de una aplicación web. Para cada vista, hubo una carpeta con su nombre correspondiente, que contuvo los archivos necesarios para construir la estructura del documento y los diferentes componentes utilizados con el framework. Estos archivos incluyeron “index.html” para la estructura HTML de la vista, “index.js” como módulo de JavaScript para la importación e implementación del framework, “style.css” para los estilos específicos de esa vista, así como imágenes u otros recursos necesarios.

Por último, en la carpeta “Components” se encontraron los componentes que fueron creados exclusivamente por los desarrolladores. Se organizó en una carpeta "Visual" para los componentes de categoría "UserVisual" y una carpeta "Service" para los componentes de categoría "UserService". Esto permitió una separación clara entre los componentes proporcionados por el framework y los desarrollados por los usuarios para sus proyectos específicos.

A continuación, se presenta el código de la instancia principal del Framework “Slice.js” (Figura 5):

```
import Logger from "./Components/Structural/Logger/Logger.js";
import Controller from "./Components/Structural/Controller/Controller.js";
import StylesManager from "./Components/Structural/StylesManager/StylesManager.js";
import sliceConfig from "./sliceConfig.json" assert { type: "json" };

export default class Slice {
  constructor() {
    this.logger = new Logger();
    this.controller = new Controller();
    this.stylesManager = new StylesManager();
    this.paths = sliceConfig.paths;
  }
```

Figura 5. Método constructor de la clase Slice

Como se mencionó anteriormente, el framework fue diseñado siguiendo un paradigma basado en componentes. En esta línea, los primeros elementos que se encontraron fueron las importaciones necesarias para vincular los componentes estructurales que forman parte esencial de la instancia principal del framework. Estos componentes incluyeron el Logger, Controller, StylesManager, y el archivo de configuración “sliceConfig. json” A su vez, dentro del constructor de la clase se crearon las instancias de los mismos y se asignaron a las propiedades de la instancia principal para poder ser accedidas por todos los componentes del Framework, junto con la configuración de rutas definida en el archivo de configuración.

```
async getClass(module) {
    try {
      const { default: myClass } = await import(module);
      return await myClass;
    } catch (error) {
      this.logger.logError("Slice", `Error loading class ${module}`, error);
    }
  }
```

Figura 6. Método getClass de la clase Slice

El método getClass de la instancia principal es fundamental en el proceso de solicitud o importación dinámica de los módulos JavaScript de los componentes (Figura 6). Utilizó la reflexión para importar módulos según la demanda del desarrollador en tiempo de ejecución, lo que permitió una flexibilidad excepcional adaptada a las necesidades específicas del framework. Esta característica fue esencial para cargar componentes

de forma dinámica y eficiente, garantizando que solo se importaran los módulos necesarios en el momento preciso, sin cargar innecesariamente recursos durante la compilación.

```
async build(componentName, props = {}) {
    if (!componentName) {
      this.logger.logError(
        "Slice",
        null,
        `Component name is required to build a component`
      );
      return null;
    }

    if (typeof componentName !== 'string') {
      this.logger.logError(
        "Slice",
        null,
        `Component name must be a string to build a component`
      );
      return null;
    }

    if (!this.controller.componentCategories.has(componentName)) {
      this.logger.logError(
        "Slice",
        null,
        `Component ${componentName} not found in components.js file`
      );
      return null;
    }

    let compontentCategory = this.controller.componentCategories.get(component
Name);

    if (compontentCategory === "Structural") {
      this.logger.logError(
        "Slice",
        null,
        `Component ${componentName} is a Structural component and cannot be bu
ilt`
      );
      return null;
    }
```

Figura 7. Método build de la clase Slice

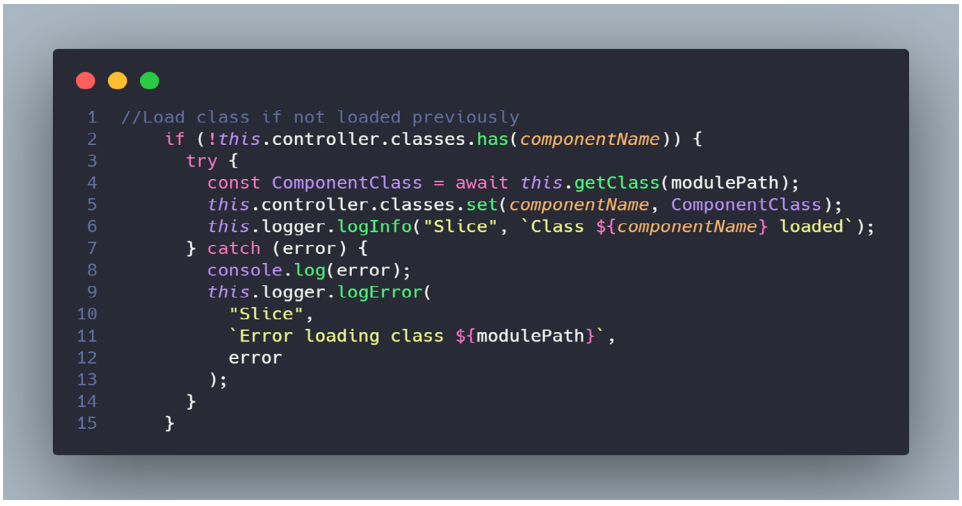
Como se mencionó anteriormente, el método build de la instancia principal es fundamental para ensamblar los componentes, marcando el inicio de su ciclo de vida (Figura 7). Inicialmente, se llevó a cabo una verificación del parámetro "componentName", que contuvo el nombre del componente a instanciar. Se verificaron diferentes condiciones, entre éstas la existencia de este parámetro, su tipo de dato (debe ser un string), y se confirmó que estuviera listado en el archivo de componentes "components.js". Además, se aseguró de que el componente no fuese de naturaleza estructural, ya que estos no pueden ser instanciados desde el método build. En caso de no cumplir con estas condiciones de validación, el método retornó null para el componente que se estuvo tratando de construir (Figura 8).

```
let componentBasePath;

    if(componentCategory.includes("User")) {
      componentCategory = componentCategory.replace("User", "")
      componentBasePath = this.paths.userComponents
    } else {
      componentBasePath = this.paths.components
    }

    const isVisual = componentCategory === "Visual";
    let modulePath = `${componentBasePath}/${componentCategory}/${componentName}/${componentName}.js`;

    // Load template if not loaded previously and component category is Visual
    if (!this.controller.templates.has(componentName) && isVisual) {
      try {
        const html = await this.controller.fetchText(componentName, "html", componentBasePath, componentCategory);
        const template = document.createElement("template");
        template.innerHTML = html;
        this.controller.templates.set(componentName, template);
        this.logger.logInfo("Slice", `Template ${componentName} loaded`);
      } catch (error) {
        console.log(error);
        this.logger.logError(
          "Slice",
          `Error loading template ${componentName}`,
          error
        );
      }
    }
```

Figura 8. Mapeo de Ruta y solicitud de template HTML de componente en el método “build”

Seguidamente, se creó la variable componentBasePath la cual permitió obtener la ruta base de la carpeta, es decir, ayudó a identificar si el componente a instanciar fue creado por los desarrolladores del framework o si fue un componente de usuario. En caso de ser un componente de usuario, asignó el valor de la variable a la ruta especificada para los componentes de usuario en el archivo de configuración “sliceConfig. json” y eliminó el prefijo “User” para que el ciclo de construcción de un componente continuara su flujo con normalidad.

Posteriormente, se creó la variable “modulePath” la cual almacenó la ruta javascript del módulo a instanciar, dicha ruta es la que es enviada al método getClass. En caso que el mapa de templates HTML que forma parte del componente controller no posea la template HTML para dicho componente, y que la categoría del componente a instanciar sea Visual, se procedió a solicitar dicha template al servidor a través del método fetchText del componente Controller. En la misma línea, se creó un elemento html “template” y se almacenó la template del componente dentro del mismo, para ser almacenado dentro del mapa templates previamente mencionado.



```
//Load class if not loaded previously
    if (!this.controller.classes.has(componentName)) {
      try {
        const ComponentClass = await this.getClass(modulePath);
        this.controller.classes.set(componentName, ComponentClass);
        this.logger.logInfo("Slice", `Class ${componentName} loaded`);
      } catch (error) {
        console.log(error);
        this.logger.logError(
          "Slice",
          `Error loading class ${modulePath}`,
          error
        );
      }
    }
```

Figura 9. Solicitud de módulo de componente en el método “build”

.

A continuación, se verificó si el mapa de clases del componente controller contuvo la clase del componente que se desea instanciar. En caso de que la clase no esté almacenada, se importó dinámicamente utilizando el método mencionado anteriormente, "getClass" (Figura 9).

```
//Load css if not loaded previously and component category is Visual
    if (!this.controller.requestedStyles.has(componentName) && isVisual) {
      try {
        const css = await this.controller.fetchText(componentName, "css");
        this.stylesManager.registerComponentStyles(componentName, css);
        this.logger.logInfo("Slice", `CSS ${componentName} loaded`);
      } catch (error) {
        console.log(error);
        this.logger.logError(
          "Slice",
          `Error loading css ${componentName}`,
          error
        );
      }
    }
```

Figura 10. Solicitud de estilos de componente en el método “build”

Seguidamente, se verificó si la categoría del componente a instanciar era visual y si el Set de estilos del componente “requestedStyles” controller no contenía almacenados los estilos para el componente a instanciar. En caso que se cumplieran dichas condiciones, se solicitaron los estilos al servidor utilizando el método fetchText del componente controller y se procedió al registro de los mismos utilizando el método “registerComponentStyles” del componente StylesManager (Figura 10).

```
//Create instance
    try {
      let componentIds = {};
      if (props.id) componentIds.id = props.id;
      if (props.sliceId) componentIds.sliceId = props.sliceId;

      delete props.id;
      delete props.sliceId;

      const ComponentClass = this.controller.classes.get(componentName);
      const componentInstance = new ComponentClass(props);

      if (componentIds.id && isVisual) componentInstance.id = componentIds.id;
      if (componentIds.sliceId)
        componentInstance.sliceId = componentIds.sliceId;


      if (!this.controller.registerComponent(componentInstance)) {
        this.logger.logError(
          "Slice",
          `Error registering instance ${componentName} ${componentInstance.sliceId}`
        );
        return null;
      }

      if (sliceConfig.debugger.enabled && compontentCategory === "Visual") {
        this.debugger.attachDebugMode(componentInstance);
      }

      //if the component has a method called init, call it
      if (componentInstance.init) await componentInstance.init();

      this.logger.logInfo(
        "Slice",
        `Instance ${componentInstance.sliceId} created`
      );
      return componentInstance;
    } catch (error) {
      console.log(error);
      this.logger.logError(
        "Slice",
        `Error creating instance ${componentName}`,
        error
      );
    }
```

Figura 11. Creación de la instancia

Después de solicitar todos los recursos necesarios para construir el componente, como se mencionó anteriormente, se procedió a eliminar los identificadores para luego asignarlos nuevamente. A continuación, se obtuvo la clase del componente y se creó la instancia utilizando reflection de manera satisfactoria. Luego, se realizaron validaciones para asignar los identificadores y registrar la instancia en el contexto de componentes del framework, específicamente en el mapa "activeComponents" del componente Controller (Figura 11).

Se procedió con una verificación para determinar si el componente Debugger estaba activo. En caso afirmativo, y si la categoría del componente instanciado fue "Visual", se ejecutó el método "attachDebugMode" del componente Debugger en dicha instancia. Para completar la construcción del componente, se ejecutó el método "init" de la instancia, si estaba disponible, y se devolvió al desarrollador de manera satisfactoria.

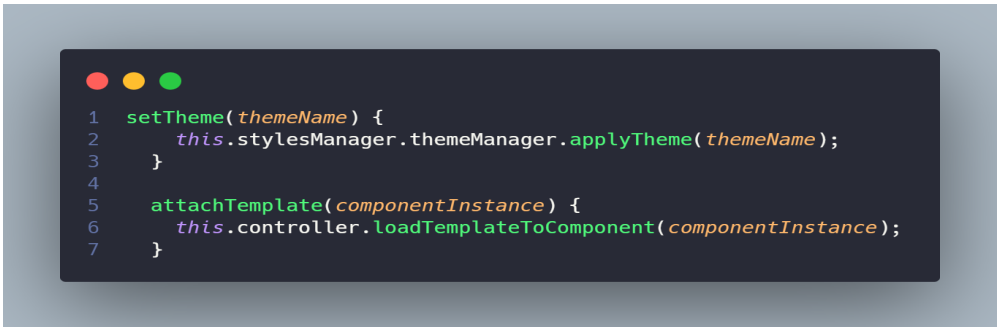


Figura 12. Métodos “setTheme” y “AttachTemplate”.

Los métodos setTheme y AttachTemplate funcionan como atajos para la ejecución de métodos de otros componentes facilitando la escritura de código (Figura 12).

Con la finalidad de brindar una interfaz más amigable para la inicialización de proyectos, creación, modificación y eliminación de componentes, se creó la herramienta “slicejs-cli”. Esta consistió en un paquete de npm que al instalarlo, permitió a través de comandos, facilitar las labores de los desarrolladores en su proceso de construcción de clientes web y a su vez, también facilitaron la creación de diferentes archivos propios del framework al igual que de brindar una estructura de proyecto estandarizada para todos los desarrolladores.

Para instalar dicha herramienta fue necesario tener instalado el manejador de paquetes de node.js “npm” y ejecutar dentro de una instancia de terminal el siguiente comando “npm install slicejs-cli”. Posterior a la instalación del paquete, se agregaron al archivo “package.json” del proyecto, los atajos de los comandos que utiliza el paquete “slicejs-cli”. En caso de que no existiera el archivo “package.json” del proyecto donde se trató de inicializar el proyecto de Slice, la herramienta se encargó de crear uno que también tuviera los comandos que brinda la herramienta de consola. A continuación, se presenta la Tabla 3 que contiene la descripción de cada comando junto a su utilización:

Tabla 3. Comandos de “slice-cli”

Comando	Descripción	Utilización
init	Inicializa el proyecto, crea la estructura de archivos para un proyecto estándar de Slice	npm run slice:init
create	Crea los archivos pertinentes para un componente según su categoría (plantilla de componente incluída)	npm run slice:create ComponentName – -category <category> - properties <properties>
update	Modifica el código de un componente añadiendo o eliminando propiedades de su contenido.	npm run slice:modify ComponentName – -add <properties> -remove properties>
delete	Elimina los archivos respectivos de un componente.	npm run slice:delete ComponentName – -category <category> - properties <properties>
list	Lista los componentes creados en el archivo components.js	npm run slice:list

Los componentes creados por el Cli se encontraron en la carpeta src/App/Components y contaron con las categorías de UserVisual y UserService.

Fase 5. Pruebas:

Como prueba de estrés, se realizó la solicitud de componentes al framework en grandes cantidades, para ser más específicos la construcción de 10 mil componentes Checkbox. Esta prueba fue cumplida satisfactoriamente por el framework, como prueba se tuvieron las siguientes imágenes (Figuras 13, 14 y 15) donde se observaron los registros de creación del componente Logger del framework y el document (DOM).

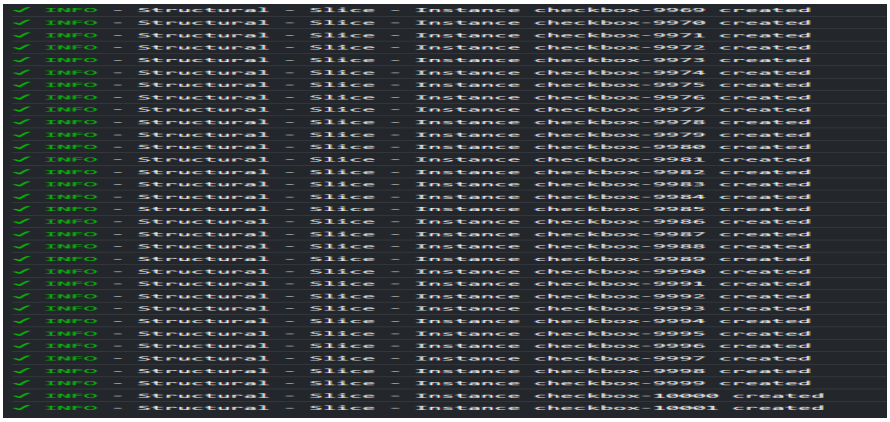


Figura 13. Registros del componente "Logger" tras la prueba de estrés.

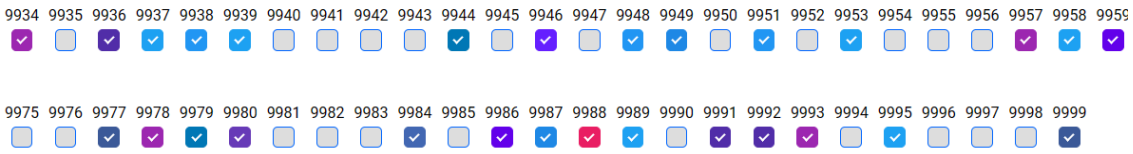


Figura 14. Creación masiva de componentes para la prueba de estrés.

```
for (let i = 0; i < 10000; i++) {
  const checkboxTest = await slice.build("Checkbox", {
    label: i,
    position: "top",
    checked: Math.random() > 0.5,
    customColor: colors[Math.floor(Math.random() * colors.length)],
  });
  document.body.appendChild(checkboxTest);
}
```

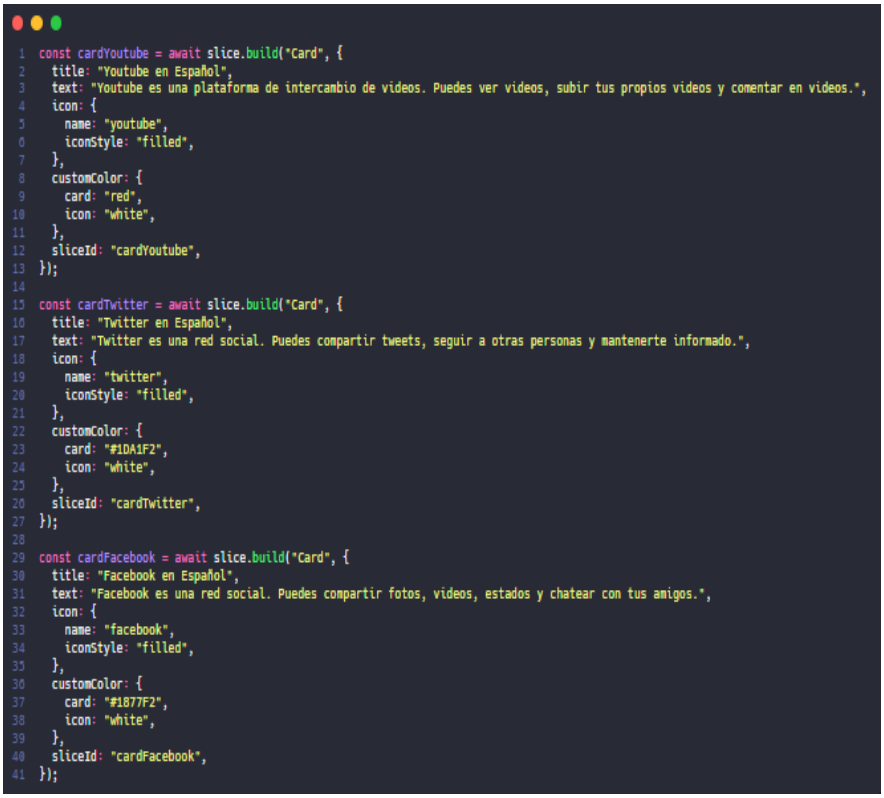
Figura 15. Código utilizado para la creación masiva de componentes para la prueba de estrés.

Seguidamente, se realizó la prueba de integración al framework la cual se llevó a cabo entre el componente visual denominado "Card" y el componente de servicio "Translator". Para este propósito, se construyó un componente "Button", al cual se le asignó la funcionalidad de cambiar entre los distintos lenguajes registrados en el archivo de mensajes "messages.json" durante el evento "onClick" (Figura 16). Esta funcionalidad se implementó mediante la validación del lenguaje actual definido por el componente "Translator". Es importante mencionar que el componente "Translator" fue previamente habilitado desde el archivo de configuración del framework "sliceConfig.json".

```
const button2 = await slice.build("Button", {
  value: "Cambiar Idioma",
  customColor: {
    button: "red",
  },
  onClickCallback: () => {
    if (slice.translator.currentLanguage === "es") {
      slice.translator.changeLanguage("en");
    } else {
      slice.translator.changeLanguage("es");
    }
  },
  sliceId: "buttonLanguage",
});

button2.classList.add("center-screen");

form.appendChild(button2);

```

Figura 16. Código de la creación de un componente “Button” para el uso del componente “Translator”.

Posteriormente, se realizó la creación de tres componentes “Card” con diferentes propiedades y temáticas como se observa en la figura 17:



```
const cardYoutube = await slice.build("Card", {
  title: "Youtube en Español",
  text: "Youtube es una plataforma de intercambio de videos. Puedes ver videos, subir tus propios videos y comentar en videos.",
  icon: {
    name: "youtube",
    iconStyle: "filled",
  },
  customColor: {
    card: "red",
    icon: "white",
  },
  sliceId: "cardYoutube",
});

const cardTwitter = await slice.build("Card", {
  title: "Twitter en Español",
  text: "Twitter es una red social. Puedes compartir tweets, seguir a otras personas y mantenerte informado.",
  icon: {
    name: "twitter",
    iconStyle: "filled",
  },
  customColor: {
    card: "#1DA1F2",
    icon: "white",
  },
  sliceId: "cardTwitter",
});

const cardFacebook = await slice.build("Card", {
  title: "Facebook en Español",
  text: "Facebook es una red social. Puedes compartir fotos, videos, estados y chatear con tus amigos.",
  icon: {
    name: "facebook",
    iconStyle: "filled",
  },
  customColor: {
    card: "#1877F2",
    icon: "white",
  },
  sliceId: "cardFacebook",
});
```

Figura 17. Código de la creación de tres componentes “Card” para la prueba de integración.

La figura anterior mostró los componentes creados junto con sus propiedades en el idioma "Español". Se eligió deliberadamente utilizar el idioma en el título para una mayor claridad. Luego, al hacer clic en el componente Button previamente creado, se activó la función asociada al evento onClickCallback del componente. Dado que el idioma actualmente asignado era "es" (abreviatura de español), se procedió a ejecutar el método ChangeLanguage del componente Translator con el idioma "en" (abreviatura de inglés), lo que provocó un cambio automático en los mensajes mostrados en las Cards con tan solo un clic en el botón.

De esta manera, se confirmó que los componentes del framework permitieron una integración fluida de sus funcionalidades, lo que indicó que las pruebas de integración fueron exitosas (Figuras 18 y 19).

Figura 18. Componentes "Card" y "Button" creados para la prueba de integración.

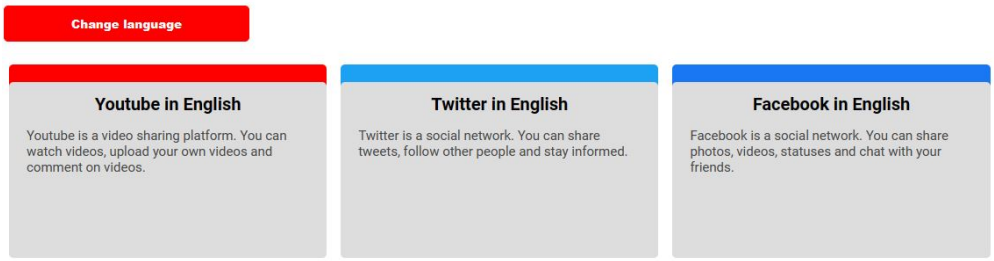


Figura 19. Componentes "Card" y "Button" tras el uso del componente "Translator".

Fase 6. Mantenimiento y Actualización

Una de las características del framework "Slice.js" es que mantuvo retrocompatibilidad entre versiones de manera que, en cada actualización se mantuviera la funcionalidad principal del framework constante y no tuvieran que realizarse actualizaciones de código forzosas y a último momento por parte de los desarrolladores de aplicaciones, durante cada actualización del framework.

Se creó una página web dedicada específicamente a mostrar la documentación del framework desarrollado, denominado "Slice" a través de un repositorio en GitHub que contuvo todo el código fuente y los archivos necesarios para la página web de la documentación del framework "Slice.js". Esta página web sirvió como un punto centralizado donde los usuarios pueden acceder a toda la información relevante sobre el framework, incluyendo detalles técnicos, ejemplos de uso, guías de integración y cualquier otra información importante.

https://slicejs-docs.vercel.app/

Página Web de la Documentación de "Slice.js".

Conclusiones

Se logró el desarrollo exitoso de un framework basado en componentes, el cual ofrece una arquitectura y funcionalidades intuitivas para la creación eficiente de aplicaciones web de diferentes tipos. Una de las características sobresalientes de este framework es su facilidad de implementación, diseñada para ofrecer una curva de aprendizaje rápida a los desarrolladores. Esto se logra mediante el uso de tecnologías estándar de la web, lo que facilita su comprensión y uso en comparación con otros frameworks más complejos.

Un aspecto crucial del framework es su enfoque en la retrocompatibilidad entre versiones. Esta característica garantiza que las aplicaciones web desarrolladas con el mismo seguirán siendo funcionales y compatibles con futuras actualizaciones, evitando así la obsolescencia y permitiendo a los desarrolladores mantener sus proyectos de manera sostenible a lo largo del tiempo.

Se ha creado una amplia gama de componentes desacoplados que se ofrecen a los desarrolladores como recursos y ejemplos para la creación de sus propios componentes. Esta biblioteca de componentes

proporciona una base sólida y coherente para el desarrollo de aplicaciones web, permitiendo una mayor eficiencia y consistencia en el proceso de desarrollo.

Para mejorar aún más la accesibilidad y la productividad de los desarrolladores, se ha desarrollado una herramienta complementaria denominada "slicejs-cli". Esta herramienta simplifica y optimiza las tareas relacionadas con el uso del framework, ofreciendo funciones como la generación automática de código, la gestión de dependencias y la automatización de tareas repetitivas.

Todos los componentes del framework están exhaustivamente documentados y disponibles en la página de documentación del proyecto. Esta documentación proporciona a los desarrolladores una referencia completa y detallada sobre cada componente, incluyendo ejemplos de uso, descripciones de propiedades y métodos, así como guías paso a paso para su implementación.

Referencias bibliográficas

[1] A. Mármol., J. Pérez, "Desarrollo de un marco de trabajo con node.js basado en componentes para el manejo de solicitudes a objetos de negocio embebidos en el backend". Revista Tecnocientífica URU, no. 19, Julio-Diciembre, 2020. [En línea]. Disponible en: http://uruojs.insiemp.com/ojs/index.php/tc/article/view/549

[2] M. Björemo., P. Trninić, "Evaluation of web application frameworks". Thesis in Software Engineering and Technology, Univ. of Gothenburg, Göteborg, Sweden, 2010. [En línea]. Disponible en: https://odr.chalmers.se/items/d880ebb9-e1b3-4300-95f0-5c25d8441870

[3] F. G. Arias, El proyecto de investigación, Sexta edición. Editorial Episteme. Caracas, Venezuela, 2012. [En línea]. Disponible en: https://www.researchgate.net/publication/301894369_EL_PROYECTO_DE_INVESTIGACION_6a_EDICION

[4] E. A. Meyer., E. Weyl, "CSS: The definitive guide: visual presentation for the web". Fourth edition, O´Reilly Media, Inc, United States of America, 2018. [En línea]. Disponible en: https://dokumen.pub/css-the-definitive-guide-visual-presentation-for-the-web-fourth-edition-9781449393199-1449393195.html