# Catálogo de componentes visuales y librería de software *NPM* para el equipo de *front-end* de la empresa TSO Mobile

Santiago Orozco Holguin

Informe de practica académica para optar al título de:

**Ingeniero Electrónico**

Asesor interno:
Prof. Gustavo Adolfo Patiño Álvarez
Universidad de Antioquia

Asesor externo:
Ronal Camilo Cuellar
Engineering Lead

Universidad de Antioquia
Facultad de Ingeniería, Departamento de Ingeniería Electrónica y Telecomunicaciones
Medellín, Colombia
2023

| Cita | (Orozco Holguin, S (2023)) |
| --- | --- |
| Referencia<br><br>Estilo APA 7 (2020) | Orozco Holguin, S (2023). Catálogo de componentes visuales y librería de software *NPM* para el equipo de *front-end* de la empresa TSO Mobile [Semestre de industria]. Universidad de Antioquia, Medellín. |

Centro de Documentación Ingeniería (CENDOI)

**Repositorio Institucional:** http://bibliotecadigital.udea.edu.co

Universidad de Antioquia - www.udea.edu.co

**Rector:** John Jairo Arboleda Cespedes.

**Decano/Director:** Jesús Francisco Vargas Bonilla.

**Jefe departamento:** Augusto Enrique Salazar Jiménez.

El contenido de esta obra corresponde al derecho de expresión de los autores y no compromete el pensamiento institucional de la Universidad de Antioquia ni desata su responsabilidad frente a terceros. Los autores asumen la responsabilidad por los derechos de autor y conexos.

**TABLA DE CONTENIDO:**


**1. Resumen.... 6**
**2. Introducción.... 6**
**3. Objetivos.... 7**
3.1. Objetivo general.... 7
3.2. Objetivos específicos.... 7
**4. Marco teórico.... 8**
4.1. Contexto general del proyecto.... 9
4.1.1. TSO Mobile a GPS Trackit company.... 9
4.1.2. Planteamiento del problema y área de aplicación.... 9
4.1.3. Planteamiento de solución propuesta.... 10
4.1.4 Componentes a ser migrados a TypeScript.... 11
4.1.4.1 Button.... 11
4.1.4.2 Checkbox.... 11
4.1.4.3 Radio Button.... 11
4.1.4.4 TextArea.... 12
4.1.4.5 Input.... 12
4.1.4.6 SidePanel.... 12
4.1.4.7 Switch.... 12
4.1.4.8 Tooltip.... 12
4.1.4.9 Modal.... 13
4.1.5. Diagrama de bloques de la solución propuesta.... 14
4.2. Desarrollo web.... 16
4.2.1. Frontend.... 19
4.2.2. HTML.... 19
4.2.3. CSS.... 20
4.2.4. JavaScript.... 21
4.2.4.1. Motor de JavaScript.... 22
4.2.4.2 Lenguaje de tipado débil.... 22
4.2.5 TypeScript.... 22
4.3. Framework de desarrollo web.... 23
4.3.1. React.... 23
4.3.1.1. Componentes visuales.... 24
4.3.1.2. Tipos de componentes.... 26
4.3.1.3 Ciclos de vida de un componente tipo clase.... 26
4.3.1.4 Hooks.... 27
4.4. Storybook.... 27

4.5. Pruebas unitarias....29
4.5.1. Jest....30
4.6. Librería NPM.... 31
4.6.1. Control de versiones de una librería de NPM.... 31
5.7. Sistema de control de versiones GIT.... 32
5.7.1 Git-flow.... 33
5.7.1.1. Rama Master o Main.... 33
5.7.1.2. Rama Develop.... 33
5.7.1.1. Rama Feature....34
5.7.1.2. Rama Release o QA....34
5.7.1.3. Rama Hotfix.... 34
**5. Metodología.... 36**
5.1. Definición de componentes a migrar....36
5.2. Creación del proyecto de React e instalación de herramienta Storybook....36
5.3. Migración de componente Button....40
5.3.1 Componente Button en JavaScript.... 41
5.3.2. Nuevo componente Button migrado a TypeScript.... 43
5.4. Storybook del componente....48
5.4.1. Ejecución de Storybook.... 51
5.4.2. Controles interactivos de un componente....52
5.4.3. Documentación de un componente.... 53
5.5. Creación de pruebas unitarias.... 55
5.6. Procedimiento realizado con los otros componentes....60
5.7. Despliegue de librería NPM....63
**6. Resultados y Análisis....64**
6.1 Resultados....65
6.1.1 Repositorio en GitHub de componentes en TypeScript.... 65
6.1.2 Storybook.... 66
6.1.2.1 Componentes.... 67
6.1.2.2 Documentación....70
6.1.3 Pruebas unitarias.... 72
6.1.4 Librería NPM.... 72
6.2. Análisis.... 73
**7. Conclusiones.... 74**
**8. Referencias bibliográficas.... 76**
**9. Anexos.... 78**

**LISTA DE FIGURAS:**

Figura 1. Input/output de prácticas académicas............................................................. 11
Figura 2. Diagrama de bloques de la solución propuesta................................................ 14
Figura 3. Ejemplo de página dinámica.......................................................................... 17
Figura 4. Ejemplo de página estática............................................................................ 18
Figura 5. Ejemplo de dispositivos IOT sin frontend......................................................... 18
Figura 6. Estructura básica de un archivo HTML.............................................................20
Figura 7. Efecto de archivo CSS sobre elementos de HTML........................................... 21
Figura 8. Ejemplo de uso del estado interno de un componente de React........................ 25
Figura 9. Render basado en valor de prop en componente <Button>...............................25
Figura 10. Cambio de propiedades por medio de controles visuales en Storybook [23]....28
Figura 11. Documentación automática de props (descripción, tipo de dato y valor por defecto) con Storybook.................................................................................................. 29
Figura 12. Versionamiento de librería NPM [27].............................................................. 32
Figura 13. Representación gráfica de las ramas Main, Hotfix, Release, Develop y Feature del modelo Git-flow [29].............................................................................................. 35
Figura 14. comando para actualizar NPM........................................................................ 37
Figura 15. Snippet para creación de proyecto básico de React con TypeScript................ 37
Figura 16. Comando para inicializar la herramienta Storybook sobre un proyecto existente de React.......................................................................................................................37
Figura 17. Comando para ejecutar la herramienta Storybook...........................................37
Figura 18. Url de acceso a la herramienta Storybook producido por el comando “npm run storybook”.................................................................................................................... 38
Figura 19. Página introductoria al Storybook...................................................................38
Figura 20. Stories del componente Button...................................................................... 39
Figura 21. Documentación generada por Storybook........................................................ 40
Figura 22. Parte del código del componente Button en JavaScript donde se aprecian algunas props................................................................................................................ 42
Figura 23. Sketch del componente Button realizado por el equipo de diseño....................44
Figura 24. Interfaz que define el tipo de dato de las props del nuevo componente Button 45
Figura 25. Inicio del código del nuevo componente Button migrado a TypeScript............ 46
Figura 26. Valores por defecto y classNames del nuevo componente Button migrado a TypeScript.................................................................................................................... 47
Figura 27. Extracto de classNames del archivo Button.scss............................................ 48
Figura 28. Parte inicial del archivo “Button.stories.tsx”.....................................................49
Figura 29. Stories Primary, Secondary y Tertiary del componente Button..........................51
Figura 30. Storybook con el componente Button y los controles de sus props................. 52
Figura 31. Diferentes render del componente Button dependiendo de la story

seleccionada........................................................................................................ 53
Figura 32. Documentación del componente Button........................................................ 54
Figura 33. Generación de código necesario para replicar componente con props modificadas desde controles interactivos........................................................................ 55
Figura 34. Definición inicial de pruebas unitarias........................................................... 57
Figura 35. Pruebas unitarias para verificar empleo de className de tamaño................. 58
Figura 36. Pruebas unitarias para verificar que el componente Button renderiza como deshabilitado y verificar el comportamiento de la función onClick.................................... 59
Figura 37. Comando para ejecutar las pruebas unitarias del archivo Button.test.tsx......... 60
Figura 38. Ejecución de todas las pruebas unitarias del componente Button pasando satisfactoriamente........................................................................................................ 60
Figura 39. Definición de componente tipo función.......................................................... 61
Figura 40. Estructura de componente tipo clase y componente tipo función.................... 62
Figura 41. Parte del archivo de ejemplo llamado package.json con información de la liberia NPM.................................................................................................................... 64
Figura 42. Repositorio en GitHub con el proyecto desarrollado....................................... 65
Figura 43. Lenguajes utilizados en el repositorio del proyecto de prácticas..................... 66
Figura 44. Página de inicio del Storybook con catalogo de componentes........................ 66
Figura 45. Componente Button en Storybook con controles interactivos.......................... 68
Figura 46. Componente Input en Storybook con controles interactivos............................ 69
Figura 47. Componente modal con controles interactivos................................................ 70
Figura 48. Documentación de componente Input............................................................ 71
Figura 49. Generación de código desde documentación de Storybook............................ 72
Figura 50. Resultado de ejecución de pruebas unitarias de los componentes de sección 6.1.2............................................................................................................................... 72
Figura 51. Librería NPM de los componentes visuales alojada en npmjs.com................. 73
Figura 52. Documentación de componente Checkbox...................................................... 79
Figura 53. Documentación de componente Radio Button................................................. 80
Figura 54. Documentación de componente TextArea....................................................... 80
Figura 55. Documentación de componente SidePanel..................................................... 81
Figura 56. Documentación de componente Switch........................................................... 82

## 1. Resumen

El siguiente documento pretende dar a conocer el trabajo realizado bajo modalidad de prácticas académicas realizada entre octubre del año 2022 y marzo del año 2023 en la empresa TSO Mobile. El objetivo de este documento es mostrar el desarrollo de un catálogo interactivo de componentes visuales de React y su respectiva documentación en la herramienta llamada Storybook, el desarrollo de pruebas unitarias para cada componente en el framework Jest y por último como la creación de una librería NPM que contenga esos componentes con el propósito de ser instalada en cualquier proyecto de la empresa y pueda ser administrada de forma centralizada logrando una estandarización de estilos y funcionalidades requeridos por la empresa.

## 2. Introducción

Desde TSO Mobile, una empresa líder en soluciones de gestión de recursos móviles, productos logísticos y servicios de monitoreo satelital de vehículos a través de GPS, se plantea la necesidad de desarrollar un catálogo privado de componentes visuales propios de la empresa. Este catálogo tiene como objetivo aumentar la productividad del equipo de desarrollo front-end y mejorar la comunicación con el equipo de diseño. Además, servirá como una referencia de los componentes visuales con los que cuenta la empresa y como documentación para que los desarrolladores puedan utilizarlos de manera fácil y estandarizada.

El presente proyecto busca el desarrollo de un catálogo interactivo de componentes visuales y su respectiva documentación a través de la herramienta Storybook. El equipo de diseño podrá variar parámetros de los diferentes componentes y entregar esta información al equipo de desarrollo front-end para que los tengan en cuenta al implementarlos en las plataformas web de la empresa. De esta manera, se asegura que los colores, diseños y funcionalidades no se vean afectados por la interpretación del desarrollador.

Para administrar los componentes visuales, se plantea la creación de una librería de NPM privada que estandarice los componentes utilizados en las plataformas web. Por medio de actualizaciones, se podrán corregir bugs, cambiar colores corporativos, tipos de letras o diseños de manera centralizada y rápida.

Dado que los componentes utilizados por el equipo de desarrollo están desarrollados en JavaScript, se sugiere migrarlos al lenguaje TypeScript al crear la librería de NPM. El tipado de variables en TypeScript permite encontrar errores en tiempo de compilación en lugar de tiempo de ejecución, lo que ayuda al desarrollador a encontrar errores de manera más rápida. También se crearán pruebas unitarias en el framework Jest para validar los estilos y funcionalidades de los componentes y detectar posibles errores en su comportamiento o visualización después de modificaciones en el código.

## 3. Objetivos

### 3.1. Objetivo general

Migrar un conjunto de componentes de software y controles visuales utilizados en el diseño de plataformas web por parte de la empresa TSO Mobile, hacia una librería privada de software del tipo ***NPM***, requerida por dicha empresa para crear un único repositorio de referencia de los componentes que son utilizados por los equipos de diseño de *front-end*, haciendo uso del framework llamado *storybook*, y mediante el lenguaje Typescript.

### 3.2. Objetivos específicos

- Identificar y caracterizar el conjunto de componentes de software y controles visuales web necesarios para el diseño de *front-ends* que requieren ser migrados hacia una nueva librería de software en la empresa TSO Mobile, identificando también su arquitectura de software requerida en dicha librería.
- Redefinir los componentes de software (previamente identificado y caracterizados) del tipo *class component*, en componentes del tipo *functional component,*

reestructurando el código existente y dejando listas las bases para lograr un correcto funcionamiento del framework *storybook*.

- Migrar los componentes escritos en JavaScript mediante la redefinición y actualización de su código al lenguaje TypeScript, que al ser un lenguaje fuertemente tipado, permite encontrar errores en el momento de compilación y no en momento de ejecución, logrando agilizar el proceso de desarrollo.
- Desarrollar una documentación apropiada de cada componente de software mediante el framework *storybook*, que permita a las áreas de desarrollo web de la compañía mayor usabilidad de los componentes de UI, además de una corrección ágil de *bugs* y actualización de dichos componentes desde un único repositorio central.
- Validar cada componente visual por medio de pruebas unitarias con la librería de JavaScript/TypeScript llamada Jest, y adicionalmente obtener la aprobación del *storybook* por parte del equipo de diseño.
- Generar un paquete ***NPM*** privado donde se centralizarán los componentes visuales utilizados por el área de desarrollo web en sus diversos productos, siguiendo la guía de documentación oficial de ***NPM***.

## 4. Marco teórico

Para un mejor entendimiento de este documento, a continuación se presentan los conceptos teóricos utilizados en la construcción y desarrollo de este proyecto. De esta forma, en esta sección se encontrará información y contexto sobre la empresa donde se desarrolló la práctica académica, los lenguajes de programación empleados y algunos términos claves de desarrollo web.

## 4.1. Contexto general del proyecto

### 4.1.1. TSO Mobile a GPS Trackit company

TSO Mobile es una empresa líder en soluciones de gestión de recursos móviles, productos logísticos y servicios de monitoreo satelital de vehículos por medio de GPS. Las soluciones de software que proporciona están basadas en la web, tanto para mercados comerciales como para mercados de consumo en países como Estados Unidos, México, Chile, Ecuador, Colombia, Perú, Panamá, Australia, Costa Rica y Canadá. En total se monitorean más de 160.000 vehículos del sector agrícola, construcción, minero, energía, logística, transporte, gobierno, entre otros.

La empresa TSO Mobile hace parte de GPS Trackit, una compañía líder en Estados Unidos en Rastreo Satelital y soluciones telemáticas [1].

### 4.1.2. Planteamiento del problema y área de aplicación

Para TSO Mobile la satisfacción de sus clientes es una de sus más altas prioridades, por lo que el buen desempeño de sus plataformas digitales con las que interactúan sus clientes es fundamental para proveer un excelente servicio.

Para aumentar la productividad del equipo de desarrollo *front-end* y mejorar la comunicación con el equipo de diseño, la empresa ha pensado en desarrollar un catálogo privado de los componentes visuales que requieren para el desarrollo de interfaces de usuario de sus plataformas. Este catálogo servirá como referencia para saber rápidamente los componentes visuales con los que cuenta la empresa y así mismo servirá como documentación para que los desarrolladores puedan desplegar dichos componentes de una forma más fácil y de forma estandarizada.

Actualmente los *sketches* propuestos por el equipo de diseño solo pueden ser validados después de que el equipo de desarrollo ha desplegado los componentes esperados en producción o en un entorno de pruebas, este flujo tiende a ser demorado ya que el

feedback del equipo de diseño solo se da algunos días o semanas después, cuando el proceso de desarrollo y verificación de funcionamiento se ha realizado.

### 4.1.3. Planteamiento de solución propuesta.

Como solución a la problemática planteada, la empresa propone usar la herramienta *Storybook* para generar un catálogo interactivo donde el equipo de diseño pueda variar parámetros de los diferentes componentes y entregar esta información al equipo de desarrollo *front-end* para que sean tenidos en cuenta a la hora de ser desplegados en las plataformas web, asegurando que los colores, diseños, funcionalidades, etc, no se vean afectados por la interpretación del desarrollador. A su vez *Storybook* también ofrece la posibilidad de generar documentación de cada componente de forma sencilla y amigable para los desarrolladores o usuario final.

Por último se plantea crear una librería de *NPM* privada desde donde se administrarán los componentes visuales por medio de versiones. Esta librería estandariza los componentes que se usan en las plataformas web, debido a que por medio de actualizaciones se podrán corregir bugs, cambiar colores corporativos, tipos de letras o diseños, de una forma centralizada y rápida.

En resumen en la figura 1 se puede observar que el practicante para poder realizar este proyecto tendrá a su disposición el repositorio de la empresa donde se encuentran los componentes visuales y apoyándose en los diseños o Mockups suministrados por el equipo de diseño deberá entregar como resultado del proyecto un repositorio en GitHub en lenguaje TypeScript con los componentes, un catálogo visual de componentes con su documentación utilizando la herramienta Storybook y por último una librería NPM que podrá ser instalada en cualquier proyecto de la empresa.

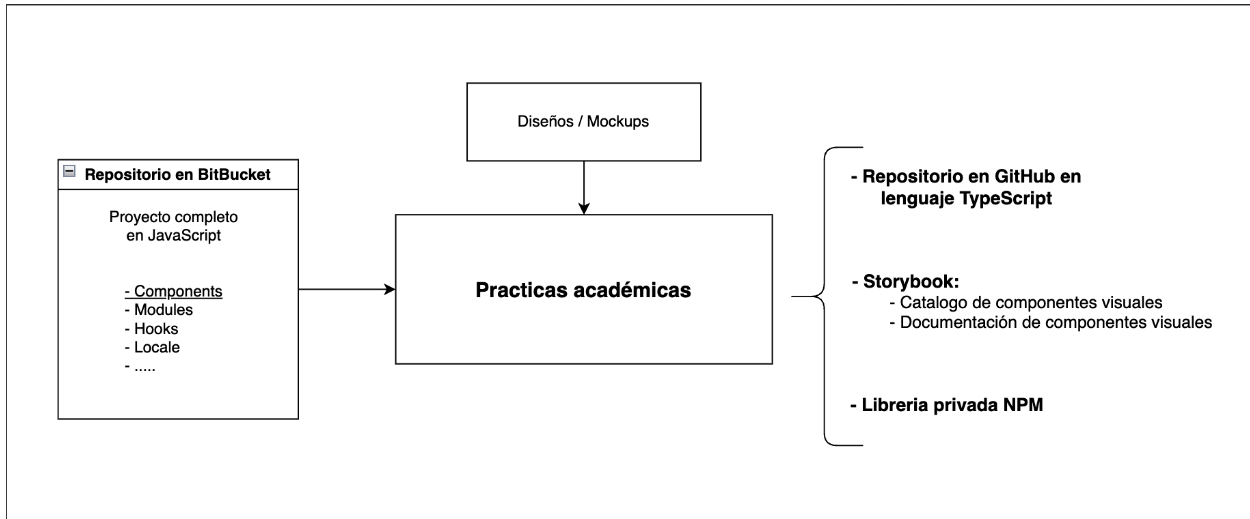



**Figura 1. Input/output de prácticas académicas**

### 4.1.4 Componentes a ser migrados a *TypeScript*

#### 4.1.4.1 Button

Un botón es un componente de la interfaz de usuario que se utiliza para crear un elemento al cual los usuarios pueden hacer clic para realizar una acción, como abrir un menú, enviar un formulario, redireccionar a otra página interna o externa.

#### 4.1.4.2 *Checkbox*

El componente *checkbox* en español se refiere a una casilla de verificación o una caja de selección. Es un elemento de la interfaz de usuario que permite al usuario seleccionar una o varias opciones de una lista de opciones disponibles. Cuando el usuario marca una casilla de selección, se considera que ha seleccionado esa opción, mientras que si la desmarca, se considera que no ha seleccionado esa opción.

#### 4.1.4.3 *Radio Button*

*Radio button* en español se refiere a un botón de opción o un botón de radio. Es un elemento de la interfaz de usuario que permite al usuario seleccionar una única opción

de una lista de opciones. Cuando el usuario selecciona una opción, automáticamente se deselecciona cualquier otra opción previamente seleccionada.

#### 4.1.4.4 *TextArea*

*Textarea* se refiere a un área de texto o un cuadro de texto grande. Es un elemento de la interfaz de usuario que permite al usuario introducir y editar grandes cantidades de texto en una página web o en una aplicación.

#### 4.1.4.5 *Input*

Input en español se refiere a una entrada o un campo de entrada. Es un elemento de la interfaz de usuario que permite al usuario introducir y editar información en una página web o en una aplicación. Los campos de entrada pueden ser de varios tipos, como texto, numérico, fechas, correos electrónicos, contraseñas, archivos, entre otros.

#### 4.1.4.6 *SidePanel*

Un *side panel* es un componente de la interfaz de usuario que se utiliza para mostrar información adicional en un panel lateral de la pantalla. Este componente se utiliza a menudo en aplicaciones web para permitir al usuario interactuar con la aplicación de manera más eficiente.

#### 4.1.4.7 *Switch*

Un *switch* es un componente de interfaz de usuario que se utiliza para alternar entre dos estados o valores. Es similar a una casilla de verificación, pero en lugar de tener una sola opción de activado/desactivado, tiene dos opciones diferentes, como "sí/no", "encendido/apagado" o "verdadero/falso".

#### 4.1.4.8 *Tooltip*

Un tooltip es un componente de interfaz de usuario que se utiliza para proporcionar información adicional sobre un elemento en una página web o aplicación. Un tooltip generalmente aparece cuando el usuario coloca el cursor sobre un elemento, y se presenta en forma de una pequeña ventana emergente que muestra un mensaje breve

y descriptivo sobre elementos como botones, enlaces, imágenes, iconos, campos de entrada y otros elementos interactivos. También se pueden utilizar para proporcionar información sobre errores o advertencias.

#### 4.1.4.9 *Modal*

Un modal es un componente de interfaz de usuario que se utiliza para mostrar contenido adicional o interactuar con el usuario mientras se mantiene el contexto principal de la página o aplicación. Un modal se muestra como una ventana emergente que se superpone sobre el contenido principal de la página y se enfoca en un mensaje específico o en una tarea que el usuario necesita realizar.

Los modales pueden ser utilizados para mostrar información importante o solicitar acciones del usuario, cómo confirmar una acción, ingresar información o proporcionar opciones adicionales. También pueden ser utilizados para proporcionar retroalimentación visual sobre la finalización exitosa de una tarea o para alertar sobre errores o advertencias.

### 4.1.5. Diagrama de bloques de la solución propuesta

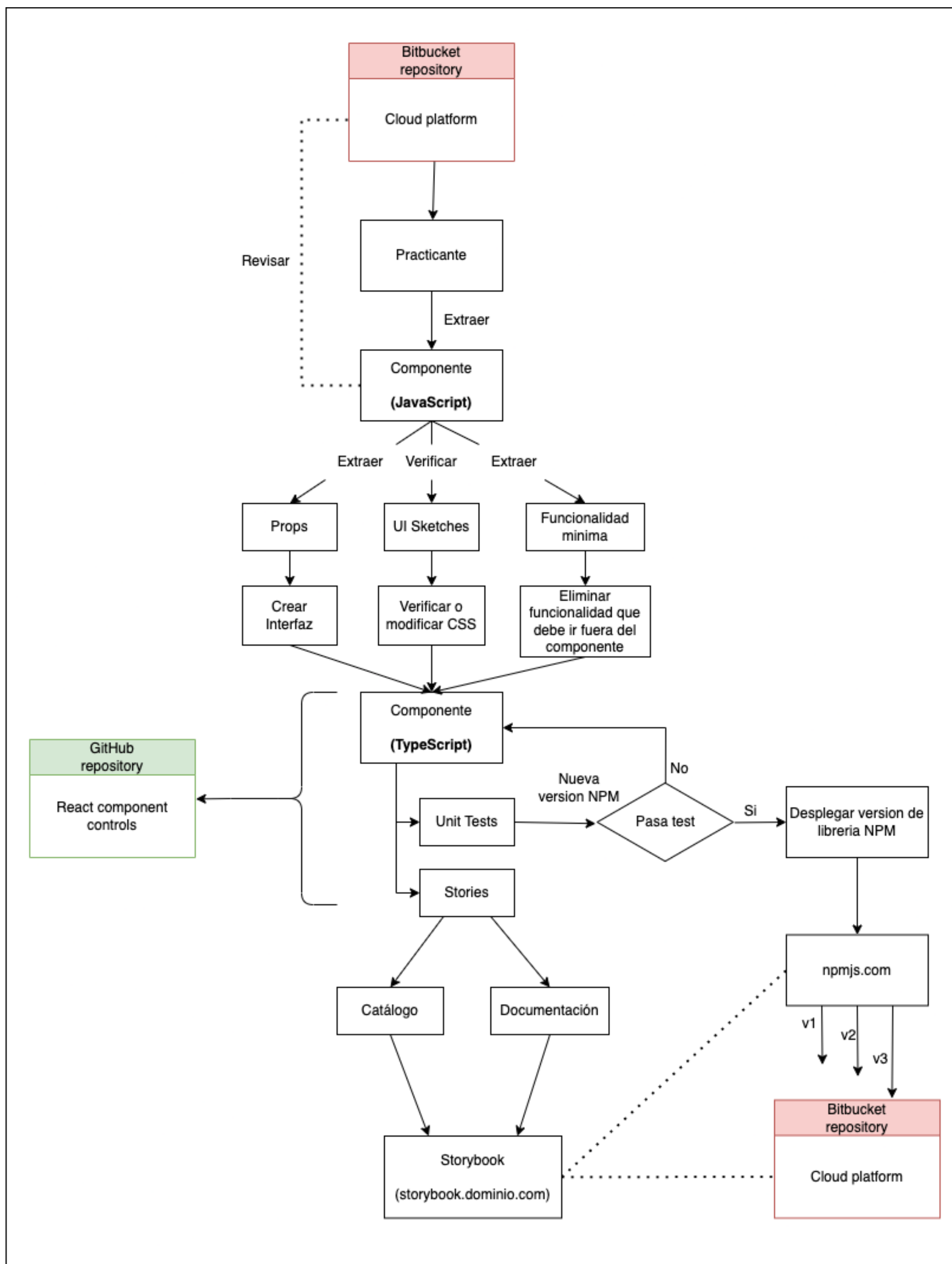



**Figura 2. Diagrama de bloques de la solución propuesta**

En la figura 2 se puede observar la solución empleada para la solución al proyecto de la práctica académica:
Se inicia con el acceso al repositorio principal desde donde se desarrolla la plataforma web con la que interactúan los clientes, este repositorio se encuentra alojado en BitBucket, el cual es un servicio basado en la nube que ayuda a los desarrolladores a almacenar y manejar su código, así como a hacerle seguimiento y controlar los cambios del mismo [2], el practicante tiene la labor de revisar el código allí descrito y extraer los códigos en JavaScript referentes a los 9 componentes antes mencionados, a cada uno de estos componentes debe extraer las propiedades o *props* que le son asignadas al momento de instanciar dicho componente, debe revisar que los sketch concuerden con las hojas de estilo (CSS) y por último debe extraer la funcionalidad mínima del componente, eliminando las funcionalidades que puedan ir por fuera de él.

Posteriormente debe crear el archivo en TypeScript y usando las *props* extraídas anteriormente debe crear una interfaz que permita definir el tipo de las variables con anterioridad. Como el código de los componentes ya existe y la tarea consiste en migrarlos a otro lenguaje, se procede a la verificación del funcionamiento por medio de la herramienta *Storybook* y una vez el componente funcione correctamente, se crean las pruebas unitarias que prueban el comportamiento, funcionalidad y/o estilos por medio de código y en caso de que las pruebas no pasen, se vuelve al componente a revisar porque no está pasando la prueba o se reestructura la prueba en caso de que hubieran errores en la elaboración de esta.

Cuando todos los componentes del proyecto han sido migrados a TypeScript y cuentan con sus *stories* y pruebas unitarias, o cuando se requiera, se despliega una nueva versión de librería *NPM* privada, la cual es almacenada en *npm*js.com, siempre y cuando las pruebas unitarias pasen satisfactoriamente. Una vez que el repositorio principal (BitBucket) requiera emplear estos componentes deberá descargar e instalar la versión que necesite de la librería *NPM* privada.

Como objetivo final, fuera del alcance de este proyecto, se espera que la empresa pueda borrar los códigos de los componentes de su repositorio en BitBucket y utilizar 100% los componentes almacenados en la librería *NPM* privada de la empresa.

### 4.2. Desarrollo web

La revolución del internet trajo consigo un conjunto de tecnologías y protocolos que permitieron la creación de diversos usos para este medio de comunicación que a su vez conlleva a la creación de diversas áreas de enfoque en la programación relacionada con el internet.

Según el modelo cliente-servidor el mundo del desarrollo web está dividido en *groso modo* en dos áreas [3]:

- Frontend (programación del lado del cliente)
- Backend (programación del lado del servidor)

El enfoque de este proyecto se centra en componentes de interfaz de usuario los cuales se encuentran del lado del área de frontend, por lo que el concepto de backend se describe brevemente, pero creo importante explicar la relación que existe entre el cliente y el servidor por medio de los siguientes ejemplos.

Cuando un usuario (cliente) ingresa a una página web ya sea desde su celular o desde un computador, generalmente ingresa por medio de un dominio tipo ejemplo.com o ejemplo.org, esto genera una serie de sucesos que termina en que el usuario reciba un conjunto de archivos (.html, .css y .js) que son ejecutados en el dispositivo que hace las peticiones al dominio (eg. ejemplo.com). Estos archivos contienen la información suficiente para renderizar o representar gráficamente un conjunto de datos e información de una forma predeterminada por la empresa o creador del sitio web al que se trata de ingresar, a esta parte de la web se le conoce como frontend (parte delantera de su traducción del inglés) ya que es la parte visible de una página web.

Cuando se ingresa a una página web que contiene contenido dinámico, este puede ser diferente para cada usuario en función de su idioma, ubicación o preferencias de la cuenta, y se carga a partir de información almacenada en bases de datos. En este caso, se utiliza un backend (parte trasera según la traducción del inglés) para proporcionar toda esta información en el momento en que sea solicitada. Un ejemplo de este escenario se presenta en la figura 3 donde el cliente (PC) interactúa con un servidor frontend que le proporciona los archivos HTML, CSS y JS que le permiten renderizar la página deseada, pero cuando el usuario requiere contenido almacenado en su cuenta como texto o imágenes o requiere manipular bases de datos estos son manejados por el servidor backend.

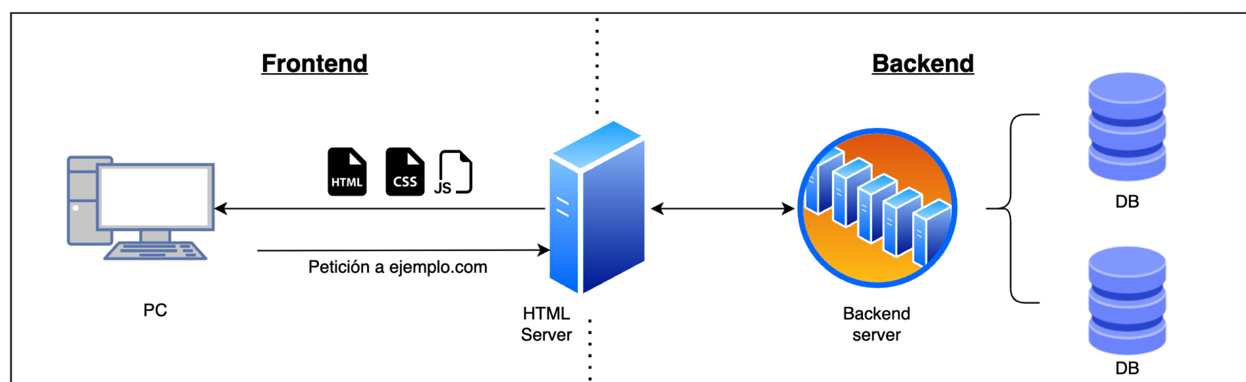



**Figura 3. Ejemplo de página dinámica**

Cuando la página web a la que se ingresa solo despliega contenido estático, es decir, no cambia con el tiempo de interacción con la página y el contenido es igual para todas las personas que ingresen a esta página, en este caso, representado por la figura 4, no existe un backend como el descrito anteriormente, más bien se comporta como un computador/servidor que cumple la función de mandar los 3 archivos del frontend (.html, .css y .js) para poder renderizar la página en el lado del usuario (cliente) .

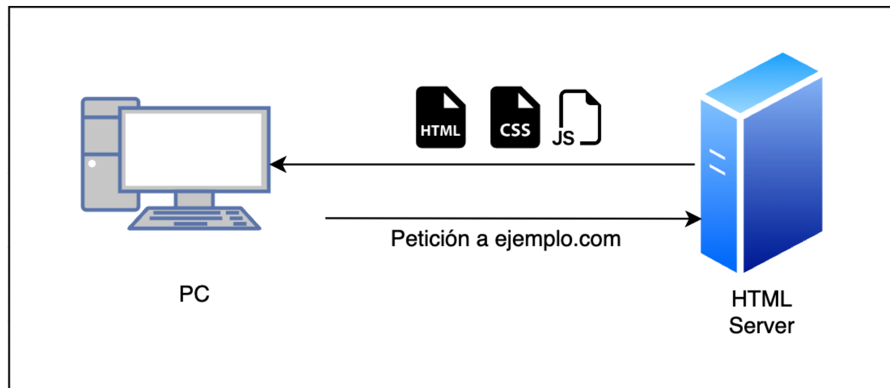



**Figura 4. Ejemplo de página estática**

También existen ocasiones representadas por la figura 5 donde existe un backend sin la necesidad de un frontend como es el caso de internet de las cosas (IoT en inglés) [4], donde los dispositivos de hardware se conectan con un servidor/backend para enviar o pedir información de forma directa, sin pasar por una interfaz de usuario y de esta forma conectarse entre sí o simplemente almacenar información requerida en una base de datos.

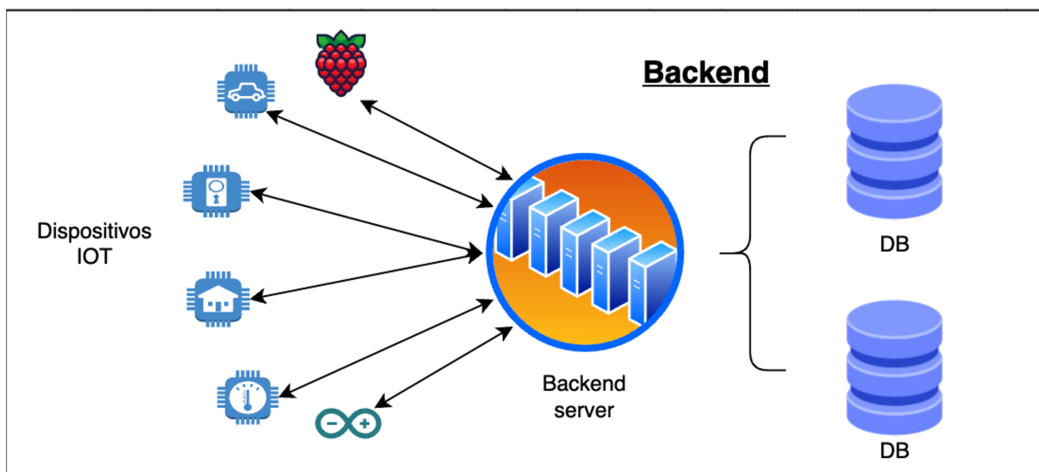



**Figura 5. Ejemplo de dispositivos IOT[1] sin frontend**

[1] **IoT:** (Internet of Things, por sus siglas en inglés) se refiere a la interconexión de dispositivos físicos, objetos y sistemas a través de internet para recopilar y compartir datos. lo que permite mejorar la eficiencia, la seguridad y la calidad de vida de las personas.

El concepto de backend hace referencia a ese servidor o computador que:

- Envía información a petición del cliente.
- Recibe información y de acuerdo con esta devuelve cierta información alojada en una base de datos.
- Recibe datos, los procesa y devuelve el resultado.

Se puede decir que el backend es la parte que se encarga del acceso y/o procesamiento de los datos.

Anteriormente estos servidores estaban ubicados en las oficinas de las empresas y se debían mantener y asegurar un óptimo funcionamiento y disponibilidad, pero en la actualidad las empresas están optando por utilizar servicios de Amazon como AWS [5], o de Microsoft como Azure [6], para delegar el mantenimiento y estabilidad del backend a estas compañías tecnológicas con mayor conocimiento técnico, y así centrarse en el desarrollo de la lógica de negocio.
Es a estos servidores en estas empresas como Amazon o Microsoft a lo que se le conoce como la "nube", esta abstracción que no vemos pero que responde a petición bajo demanda [7].

#### 4.2.1. Frontend

También conocido como la parte del lado del cliente ya que es el código que se ejecuta en el computador o celular del usuario (cliente) que trata de ingresar a una página web, y está constituido por 3 archivos principales: HTML, CSS y JS.

#### 4.2.2. HTML

Lenguaje de Marcas de Hipertexto (del inglés *HyperText Markup Language*) es el componente más básico de una página web, ya que define el significado y la estructura (figura 6) del contenido de esta y se caracteriza por emplear una serie de tags, marcas o etiquetas que definen los elementos de los que está constituida dicha página web. Entre los elementos que define HTML se encuentran:

- <head>, <title>, <body>, <footer>, <img>

Y muchos otros elementos que permiten una organización y ubicación del contenido dentro de una página web [8].

```
1   <!DOCTYPE html>
2   <html lang="en">
3       <head>
4           <title>Título</title>
5       </head>
6       <body>
7           .
8           ..
9           ...
10              } Contenido
11          ...
12          ..
13          .
14      </body>
15  </html>
```

**Figura 6. Estructura básica de un archivo HTML.**

### 4.2.3. CSS

Hojas de Estilo en Cascada (del inglés *Cascading Style Sheets*) o CSS es el lenguaje de estilos utilizado para describir la presentación de documentos HTML. CSS describe cómo debe ser renderizado el elemento estructurado en la pantalla del cliente [9].

Propiedades como el color, tipo de letra, tamaño de letra, color del borde, ancho del borde son algunas de las características de los elementos de HTML que pueden ser modificados por medio de CSS.

En la figura 7 se puede apreciar un ejemplo de los efectos que produce el archivo CSS sobre los elementos de HTML, donde manteniendo las mismas variables: color, tipo de letra y tamaño de letra del primer escenario del tag <p> (que hace referencia a párrafo) se logra ver en el segundo escenario el efecto del cambio del tamaño de letra y en el tercer escenario el cambio del color de letra.

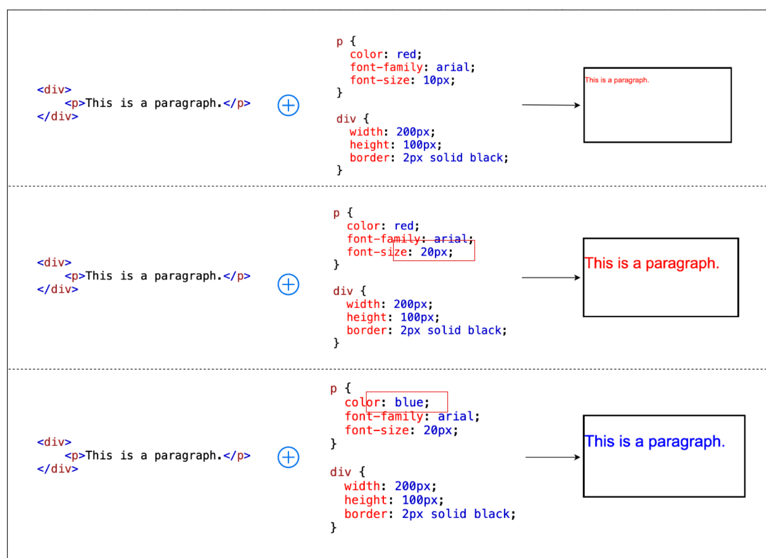



**Figura 7. Efecto de archivo CSS sobre elementos de HTML**

### 4.2.4. JavaScript

Anteriormente, las páginas web eran estáticas, similares a las páginas de un libro. Una página estática mostraba principalmente información en un diseño fijo y no todo aquello que se espera de un sitio web moderno. JavaScript surgió como una tecnología del lado del navegador para hacer que las aplicaciones web fueran más dinámicas. Por medio de JavaScript, los navegadores son capaces de responder a la interacción de los usuarios y cambiar la distribución del contenido en la página web [10].

A medida que el lenguaje evolucionó, los desarrolladores de JavaScript establecieron librerías, framework y prácticas de programación, y comenzaron a utilizarlo fuera de los navegadores web. En la actualidad, se puede utilizar JavaScript para el desarrollo tanto del lado del cliente como del lado del servidor.

JavaScript se clasifica principalmente como un lenguaje interpretado. El código es directamente traducido a código de lenguaje de máquina mediante un motor de JavaScript.

#### 4.2.4.1. Motor de JavaScript

Un motor JavaScript es un programa de computación que ejecuta código JavaScript. Los primeros motores de JavaScript eran verdaderos intérpretes, pero todos los motores modernos utilizan el método justo a tiempo o la compilación en tiempo de ejecución para mejorar el rendimiento [10].

#### 4.2.4.2 Lenguaje de tipado débil

JavaScript es un lenguaje de tipado débil, lo que significa que no se requiere que el programador defina explícitamente el tipo de una variable. Durante la ejecución, una variable puede contener cualquier tipo de datos, y las operaciones que se realizan con ella se ajustan automáticamente al tipo actual de la variable. El resultado de una operación también puede ser de un tipo de dato diferente al de los operandos originales; por ejemplo, una operación podría devolver el resultado como el string "5" en lugar del número 5. Este comportamiento puede causar errores en el código lo que puede llevar a resultados inesperados [10].

### 4.2.5 TypeScript

TypeScript es un lenguaje de programación desarrollado por Microsoft, el cual consiste en un superconjunto de JavaScript. Se caracteriza por ser un lenguaje de tipado estático, lo que significa que el tipo de datos de cada variable tiene que ser declarado explícitamente en el código. Esto ayuda a prevenir errores comunes en el código, como operaciones con tipos de datos incorrectos. TypeScript también agrega nuevas características al lenguaje JavaScript, como la capacidad de definir interfaces lo que permite definir el tipo de datos de cada variable [11].

A diferencia de JavaScript, con TypeScript, al ser un lenguaje tipado, mientras se escribe el código presenta la ventaja de recibir un feedback de errores por el tipo de dato que se le pasa a una variable, lo que evita errores a la hora de su ejecución. Esto facilita la lectura del código escrito por otras personas ya que si en javascript alguien

modifica un código común, sola la persona que agrega nuevas variables sabrá rápidamente a qué tipo de dato hace referencia. Es normal en JavaScript que una variable empiece con un valor de 0 y que pueda terminar siendo usada para almacenar un string como “cero” o “0”, o incluso ser utilizada posteriormente para almacenar un *array* o una función.

El uso de typescript en el presente proyecto de Semestre de Industria es fundamental, ya que ayudará en un futuro a la legibilidad de código y a una correcta documentación del mismo.

## 4.3. Framework de desarrollo web

Un framework de desarrollo web es un conjunto de herramientas, librerías y patrones de diseño de software que facilitan la creación de aplicaciones web. Estos frameworks facilitan una estructura y un conjunto de estándares para que los desarrolladores puedan crear aplicaciones web de manera más rápida, eficiente y escalable [12].

### 4.3.1. *React*

React.js o *React* [13] es una librería de JavaScript para la construcción de interfaces de usuario (UI) para aplicaciones web. Fue desarrollada por la empresa Facebook ahora conocida como META [14], posteriormente siendo liberada como *open source* [15] y se utiliza para crear componentes reutilizables y escalables en aplicaciones web de una sola página.

React se basa en el concepto de componentes [16], que son bloques de código que representan una parte de la interfaz de usuario. Cada componente puede tener su propio estado y propiedades, lo que permite la creación de interfaces de usuario dinámicas. Una de las características clave de *React* es su enfoque en el modelo de programación declarativo. En lugar de manipular directamente el DOM (Document Object Model) de la página, *React* utiliza una abstracción llamada Virtual DOM o DOM virtual [17] [18].

El DOM Virtual es una copia en memoria del DOM real que se mantiene por debajo de la capa de abstracción que proporciona *React*. Cada vez que se realiza un cambio en la interfaz de usuario (UI), *React* compara la versión anterior del DOM Virtual con la nueva versión y determina cuáles son los cambios que deben realizarse en el DOM real. Al utilizar el DOM Virtual, *React* puede reducir la cantidad de operaciones necesarias para actualizar la interfaz de usuario, lo que resulta en una mayor eficiencia y rendimiento en las aplicaciones web.

#### 4.3.1.1. Componentes visuales

Un componente de *React* es una función de JavaScript a la que se le puede *agregar etiquetas de HTML* y a su vez describe el comportamiento, el estilo y la estructura de un componente de la interfaz de usuario (UI) [16].

Cada componente en *React* puede tener su propio estado (state) y propiedades (props). El estado representa los datos internos del componente, los cuales pueden cambiar con el tiempo. La figura 8 describe un ejemplo de uso del estado de un componente de *React*, donde la variable de estado *count* mantiene el valor de un contador que cambia solo con el accionar de uno de los dos botones disponibles y cambia su estado sumando 1 o sumando 2 según el botón que sea oprimido y este valor de estado del contador es mostrado en pantalla y actualizado cada que ocurre un cambio de estado.

Las propiedades (figura 9) son los datos que se pasan al componente desde un componente *padre y que cambian la forma en cómo se comporta o se ve en pantalla*. De cierta forma se puede interpretar a estos componentes como *legos*, que a medida que se agregan unos con otros se pueden crear estructuras más complejas como módulos o páginas enteras.

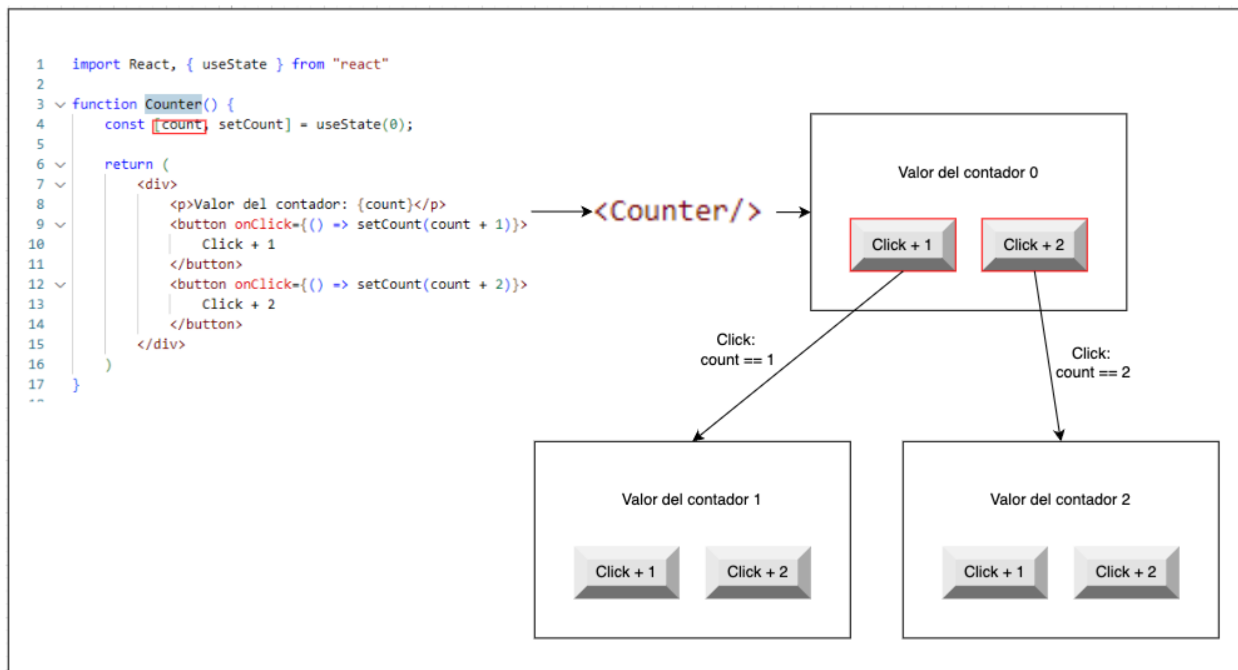



**Figura 8. Ejemplo de uso del estado interno de un componente de *React***

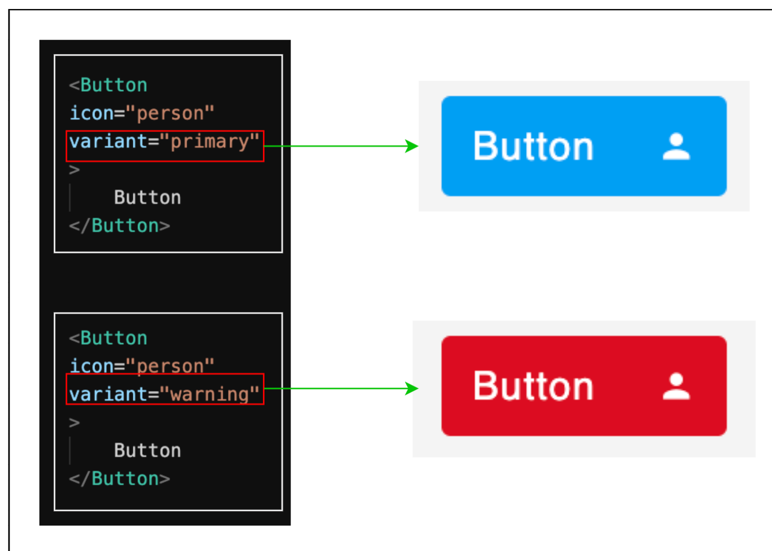



**Figura 9. Render basado en valor de *prop* en componente <Button>**

#### 4.3.1.2. Tipos de componentes

En React, hay varios tipos de componentes que se utilizan para construir una aplicación web [19]:

- Clase:
    - Extiende la clase *Component* de *React*, y requieren la definición de un método *render()* que devuelve el contenido del componente.
    - Poseen ciclos de vida como render() *componentDidMount()* o *componentWillUnmount()*
- Función:
    - La función devuelve el contenido del componente.
    - No requiere ciclos de vida y de necesitarlos se pueden usar otras funciones de *React*.
    - Son más legibles que los de tipo *clase,* por su sintaxis más sencilla, y debido a que se ahorra implementar ciclos de vida.

#### 4.3.1.3 Ciclos de vida de un componente tipo clase

En *React*, los componentes de clase tienen un ciclo de vida que se divide en tres fases: montaje, actualización y desmontaje. Durante cada fase, los componentes pueden ejecutar ciertos métodos que permiten controlar su comportamiento y actualizar la interfaz de usuario [20].

Entre los métodos más utilizados en cada fase se encuentran:

**1- Montaje:**

- *constructor()*: Este método se llama cuando se crea una instancia del componente y permite inicializar su estado y enlazar los métodos de clase.
- *render()*: Este método es obligatorio y devuelve el contenido que se va a renderizar en el navegador.
- *componentDidMount()*: Este método se llama una vez que el componente ha sido renderizado en el navegador. Aquí se puede realizar acciones como cargar datos externos y actualizar el estado del componente.

**2- Actualización:**

- *componentDidUpdate()*: Este método se llama después de que el componente ha sido actualizado en el navegador. Aquí se pueden realizar acciones adicionales como cargar datos externos y actualizar el estado del componente.

**3- Desmontaje:**

- *componentWillUnmount()*: Este método se llama justo antes de que el componente sea eliminado del DOM. Aquí se puede limpiar cualquier recurso que se haya creado durante la vida útil del componente.

#### 4.3.1.4 Hooks

Los hooks en *React* son una característica que permiten utilizar el estado y otras características de *React* en componentes de tipo función, sin necesidad de crear componentes de tipo clase [21].

Los hooks son funciones especiales que permiten conectarse a ciertos comportamientos o estados de un componente, y así poder utilizarlos dentro de la función del componente.

Los hooks en *React* incluyen una serie de hooks integrados proporcionados por *React*, como useState(), useEffect(), useContext() y useRef(), así como la posibilidad de crear hooks personalizados para reutilizar lógica entre diferentes componentes.

## 4.4. *Storybook*

Storybook es una herramienta frontend de desarrollo agnóstica al lenguaje pero disponible para *React* que permite a los desarrolladores crear y visualizar componentes reutilizables en un entorno aislado y a su vez generar documentación de dichos componentes [22].

Posee la ventaja de crear un catálogo visual de componentes y de forma interactiva cambiar las propiedades de estos y ver su efecto en el renderizado en pantalla de forma inmediata [23].

En la figura 10 se puede observar el resultado que genera la herramienta *Storybook*, con una lista de componentes en la parte izquierda pero en este caso seleccionando y mostrando un componente botón con 4 controles que al interactuar con ellos se puede cambiar su apariencia modificando el color de fondo, si es un botón primario o no, el tamaño del botón y con el control label se logra cambiar el contenido de texto del botón.

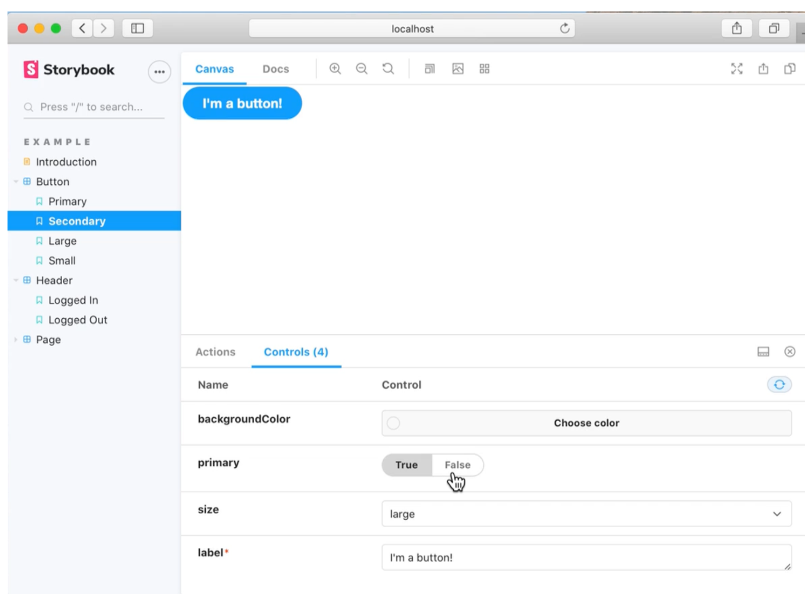



**Figura 10. Cambio de propiedades por medio de controles visuales en *Storybook* [23]**

En cuanto a la documentación automática generada por *Storybook*, esta se realiza por medio de comentarios sobre las propiedades (props) definidas en las interfaces de los componentes. En la sección 5.4.3. correspondiente en la metodología se expande esta explicación y generación.

En la figura 11 se muestra un ejemplo de la documentación generada por la herramienta *Storybook*, donde se puede apreciar por columnas el nombre de la variable o prop, la descripción de dicha prop, el valor por defecto que se le ha asignado y en la columna de control, se presenta la posibilidad de cambiar el valor de la *prop* que se requiera para cambiar el render que se muestra en la parte de arriba de las columnas. También cuenta con la opción de mostrar el código necesario para generar el componente tal como se modifica en esta documentación interactiva. En la sección 5.4.3. correspondiente se expande más el uso de esta documentación.

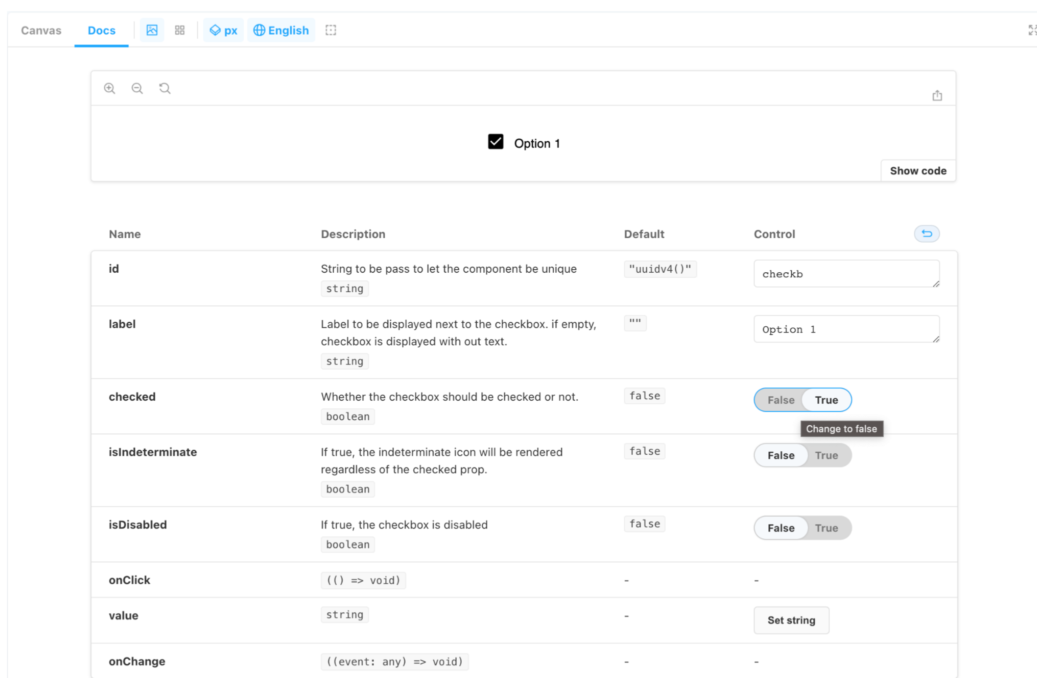



**Figura 11. Documentación automática de *props* (descripción, tipo de dato y valor por defecto) con *Storybook***

## 4.5. Pruebas unitarias

El término unit test proviene del inglés y se refiere al método de comprobación de las “unidades” (en inglés unit) más pequeñas del software. El objetivo de las pruebas unitarias es verificar que una parte específica del código funciona correctamente y que se comporta de la manera esperada ante diferentes entradas y condiciones [24].

Las pruebas unitarias se escriben en el mismo lenguaje de programación que el código que se está probando y generalmente se ejecutan automáticamente como parte del proceso de compilación o construcción del software. Las pruebas unitarias se realizan en un ambiente controlado y aislado para garantizar que los resultados sean predecibles y repetibles.

Las pruebas unitarias para componentes de *React* se centran en probar la lógica del componente, sus propiedades, estados y eventos.

Algunas de las ventajas que trae trabajar con pruebas unitarias son:

- Mejoran la calidad del software y reducen el número de errores.
- Ayudar a detectar errores en una etapa temprana del ciclo de desarrollo de software.
- Permitir realizar cambios en el código sin temor a introducir nuevos errores

### 4.5.1. Jest

Jest es un framework de pruebas de JavaScript desarrollado por Facebook, diseñado para hacer pruebas de manera fácil, rápida y con una sintaxis sencilla y legible. Es ampliamente utilizado por la comunidad de *React* ya que se integra muy bien con este [25].

Jest proporciona un catálogo completo de herramientas de pruebas, incluyendo una librería de aserciones (expect), simuladores de eventos y componentes, y herramientas para la creación de *stubs*, *mocks* y *spies.*

Una de las características más importantes de Jest es que tiene un tiempo de ejecución rápido, ya que utiliza técnicas de optimización para realizar pruebas de manera más rápida, lo que reduce el tiempo que los desarrolladores deben esperar para obtener resultados y también viene con una configuración predeterminada que se puede

modificar, pero que es adecuada para la mayoría de los proyectos sin necesidad de configuración adicional [25] [26].

## 4.6. Librería *NPM*

*NPM* (Node Package Manager) es un administrador de paquetes de software para JavaScript, que permite a los desarrolladores compartir, reutilizar y distribuir código de manera sencilla. Una librería *NPM* es un paquete de software que contiene código JavaScript listo para ser utilizado en otro proyecto. Estas librerías contienen varias funciones que se pueden importar en un proyecto de JavaScript para mejorar su funcionalidad y agilizar el desarrollo [27].

Al utilizar una librería *NPM*, los desarrolladores se pueden beneficiar ya que pueden ahorrar tiempo y esfuerzo al no tener que escribir su propio código para cada función o característica de su proyecto. Las librerías *NPM* también suelen ser actualizadas regularmente por sus creadores para mantenerse al día con las últimas tecnologías y corregir errores o vulnerabilidades de seguridad.

Por el contrario, también puede generar dependencia a librerías de terceros que en el momento de dejar de funcionar o no presentar más soporte puede poner en riesgo el proyecto que las use, así que se debe ser consciente de las dependencias que se están agregando a los proyectos

### 4.6.1. Control de versiones de una librería de *NPM*

De acuerdo con la imagen mostrada en la figura 12 sobre las versiones de la librería *NPM*:

- El número MAJOR se incrementa cuando se realizan cambios que no son compatibles con versiones anteriores de la librería.

- El número MINOR se incrementa cuando se agregan nuevas funcionalidades a la librería, pero estas no afectan la compatibilidad con versiones anteriores.
- El número PATCH se incrementa cuando se realizan correcciones de errores o se realizan mejoras menores que no afectan la compatibilidad con versiones anteriores.

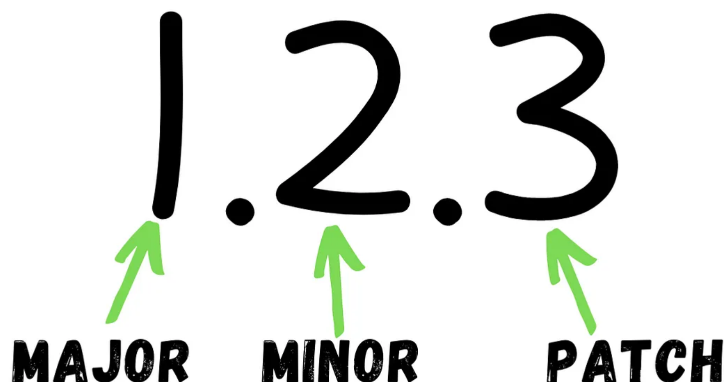



**Figura 12. Versionamiento de librería *NPM [27]***

Las librerías *NPM* pueden ser tanto públicas como privadas, en el primer caso se fomenta el desarrollo *open source* y distribución masiva de dichas librerías, y en el caso de las librerías privadas, estas están bajo protección del creador de la librería, requieren usuario y contraseña para ser utilizadas y se presenta mucho en las empresas que requieren que sus códigos sigan siendo confidenciales pero accesibles para el desarrollo de diversos productos, reutilizando código propio.

### 5.7. Sistema de control de versiones GIT

En la empresa TSO Mobile los códigos fuente de los proyectos que se desarrollan son almacenados y administrados en plataformas como BitBucket o GitHub que son dos de los servicios de alojamiento de repositorios GIT más populares utilizados por desarrolladores para gestionar el control de versiones de sus proyectos de software.

Git es un sistema de control de versiones distribuido, que se utiliza para rastrear los cambios realizados en los archivos de un proyecto de software. Con Git, los desarrolladores pueden trabajar en paralelo en diferentes ramas del código fuente, lo que permite la colaboración en equipo y la gestión de cambios. Cada desarrollador tiene una copia completa del repositorio de Git en su máquina local, lo que permite trabajar sin conexión a internet y realizar cambios de manera independiente [28].

Git permite la creación de ramas para el desarrollo de nuevas funcionalidades o para la corrección de errores, y luego la integración de estas ramas a la rama principal del proyecto mediante el proceso de "merge". También permite la creación de etiquetas para marcar versiones específicas del proyecto [28].

### 5.7.1 Git-flow

Git-flow es un modelo de flujo de trabajo para el control de versiones con Git que proporciona una estructura y un proceso definido para la colaboración y el desarrollo de software en equipo. Este modelo se basa en dos ramas principales: la rama "master" o “main” y la rama "develop".

#### 5.7.1.1. Rama Master o Main

Es la rama principal del proyecto y representa la versión estable del software. Esta rama no se utiliza para el desarrollo diario, sino que se reserva para la versión final del software.

#### 5.7.1.2. Rama Develop

Es la rama de integración de código donde se realizan los cambios y mejoras de forma regular, y que se utiliza para construir y probar nuevas funcionalidades.

Además de estas dos ramas principales, el modelo Git-flow utiliza una serie de ramas secundarias para el desarrollo de nuevas funcionalidades, la solución de errores y la preparación de despliegues a producción. Entre estas ramas se encuentran:

#### 5.7.1.1. Rama Feature

Se utiliza para el desarrollo de nuevas funcionalidades, y se crean a partir de la rama "develop". Una vez que se completa el trabajo en una "feature", se fusiona de vuelta a "develop".

#### 5.7.1.2. Rama Release o QA

Se utiliza para preparar una nueva versión del software para su lanzamiento. Se crea a partir de "develop", y se usa para corregir errores y realizar pruebas finales. Es en esta rama donde el grupo de analista de calidad de una empresa se encarga de verificar el correcto funcionamiento de la aplicación o de partes específicas que se ha desarrollado desarrollando y una vez que está lista para lanzar, se fusiona en ambas ramas, "develop" y "master".

#### 5.7.1.3. Rama Hotfix

Se utiliza para solucionar problemas críticos en la rama "master" y se crea a partir de ella. Una vez que se soluciona el problema, se fusiona de vuelta a "master" y "develop".

En la figura 13 se puede observar la diferentes relaciones entre las ramas mencionadas anteriormente, donde el eje x representa el transcurso del tiempo de desarrollo y los puntos de colores representan las diferentes ramas y su relación de origen y destino con las demás ramas. Con relación al proyecto a realizar, se debe migrar los componentes que se encuentran en la rama "main" del repositorio de BitBucket al repositorio en GitHub. La rama "main" será la versión del código que ha sido probada y se tiene certeza de su correcto funcionamiento para ser desplegada en producción. Una nueva versión de la librería NPM se libera cuando la rama "main" se actualiza. Por otro lado, la rama "develop" es la rama más actualizada donde todavía se están haciendo cambios a los componentes, y se encuentran en revisión de estilo y

funcionalidad. La rama "feature" corresponde a la migración de cada componente, y se crea una rama por cada componente. Una vez que se termina el desarrollo de cada componente, dicha rama se mezcla con la rama "develop". Finalmente, cuando se crea conveniente, se crea una rama "release" a partir de la rama "develop" y se entrega al equipo de calidad de la empresa para que realicen las pruebas pertinentes de funcionalidad. Si dicho equipo aprueba la rama "release", esta se mezcla con la rama "main", obteniendo una versión con nuevas funcionalidades y/o componentes en producción.

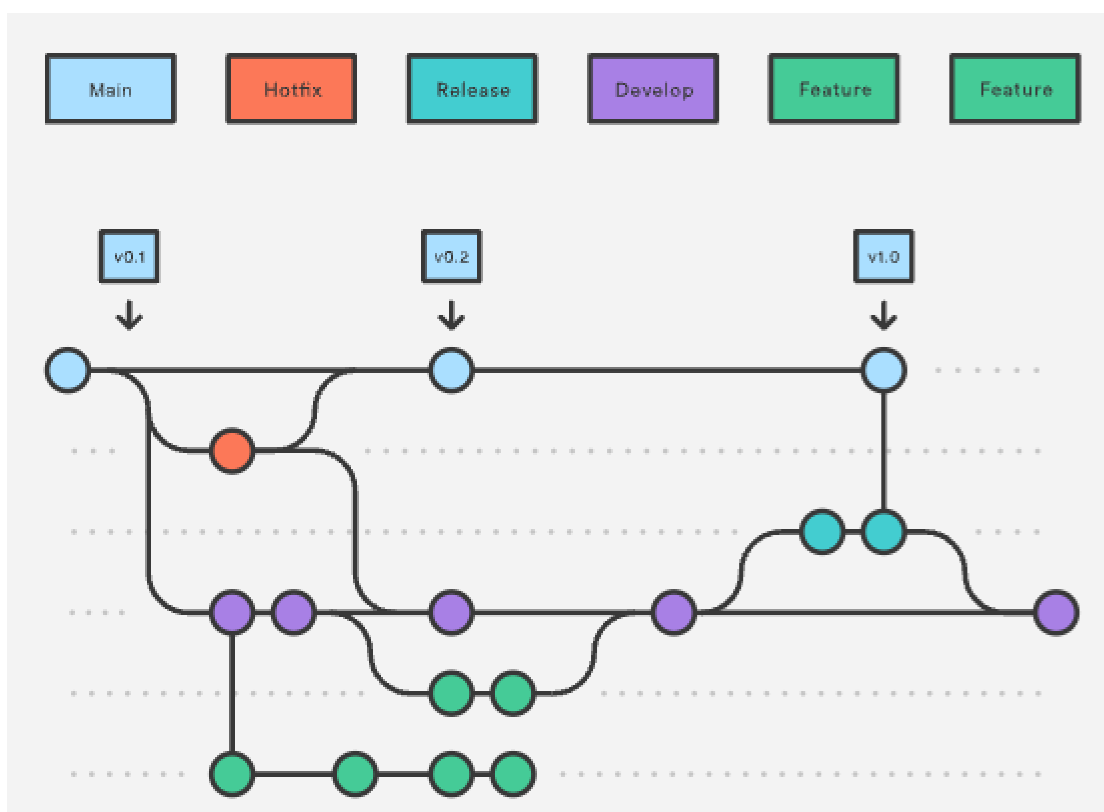



**Figura 13. Representación gráfica de las ramas *Main*, *Hotfix*, *Release*, *Develop* y *Feature* del modelo *Git-flow* [29]**

## 5. Metodología

En esta sección se describe de forma detallada el procedimiento que se empleó para darle solución a la propuesta planteada en el marco teórico.

### 5.1. Definición de componentes a migrar

Esta etapa estuvo guiada por el líder del proyecto y asesor externo de este trabajo. Durante las primeras dos semanas se realizaron reuniones para introducir el repositorio, me asignaron credenciales que me permitían ingresar al repositorio y descargarlo en el computador que me fue asignado por la empresa, se me explicó la forma como está estructurado el proyecto de la plataforma web, y finalmente se escogieron los 9 componentes iniciales en JavaScript para ser migrados y posteriormente comenzar con la creación del nuevo repositorio en TypeScript, donde se iba a generar el catálogo de componentes visuales y su documentación con la ayuda de la herramienta *Storybook* y a su vez generar las pruebas unitarias de cada componente.

### 5.2. Creación del proyecto de *React* e instalación de herramienta *Storybook*

En esta etapa se parte creando un proyecto desde cero de *React*, para esto se debe asegurar que se tenga instalado NodeJs en el computador donde se realiza el desarrollo [30]. Una vez se esté seguro que NodeJs está instalado correctamente, se procede con la actualización del manager de paquetes de Node llamado *NPM*, este se actualiza desde la terminal con el comando de la figura 14.

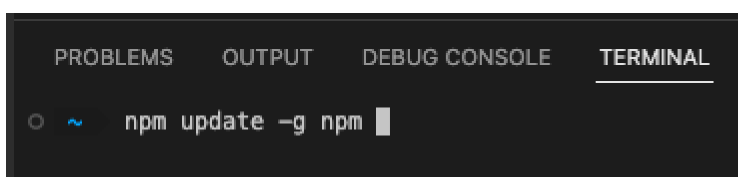


**Figura 14. comando para actualizar *NPM***

Se continua con la creación del proyecto de *React* utilizando el snippet de node que garantiza contener la versión más actualizada de *React* y sus dependencias. Para efectos prácticos en la figura 15 se muestra cómo se crea un proyecto en TypeScript de *React* llamado test desde el escritorio (Desktop).

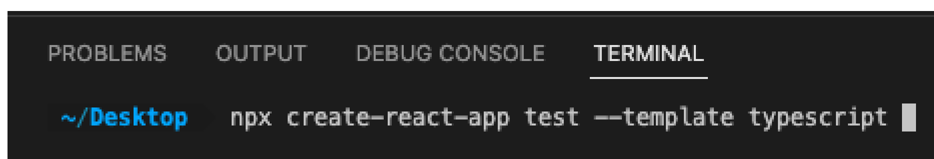



**Figura 15. Snippet para creación de proyecto básico de *React* con TypeScript**

Una vez se ha creado el proyecto de *React* se procede a instalar e inicializar el proyecto con la herramienta *Storybook* con el comando mostrado en la figura 16

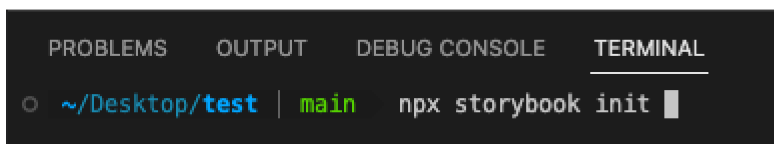



**Figura 16. Comando para inicializar la herramienta *Storybook* sobre un proyecto existente de *React***

Una vez inicializado el proyecto de *React* con la herramienta de *Storybook*, se corre el comando de la figura 17 para ejecutar el *Storybook* que viene por defecto con la inicialización de la herramienta.

**Figura 17. Comando para ejecutar la herramienta *Storybook***

Al finalizar de ejecutar el comando anterior, aparece en la terminal el mensaje de la figura 18 que indica que la herramienta ya puede ser accedida por medio de la url http://localhost:6006/

```
Storybook 6.5.16 for React started
7.87 s for preview

 Local:            http://localhost:6006/
 On your network:  http://192.168.1.2:6006/
```

**Figura 18. Url de acceso a la herramienta *Storybook* producido por el comando "*npm* run *storybook*"**

Cuando ingresamos a esta url nos recibe una página con una introducción al *Storybook* (figura 19) y debajo de esta podemos acceder a un catálogo de componentes interactivo bastante corto que vienen por defecto al instalar la herramienta *Storybook*, en este caso cuenta con los componentes *Button*, Header y Page en la parte izquierda.

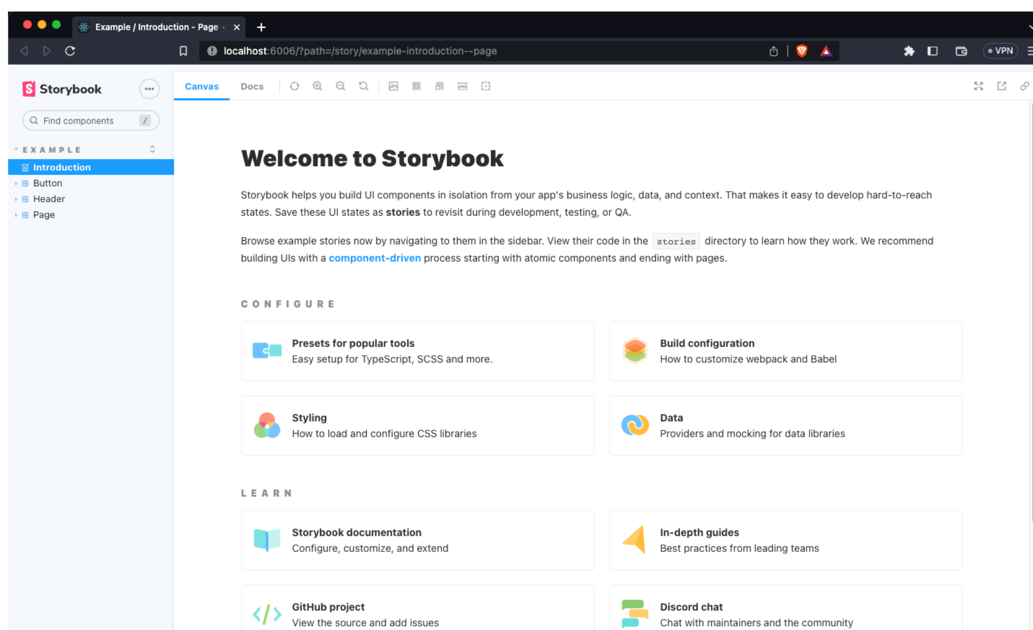



**Figura 19. Página introductoria al *Storybook***

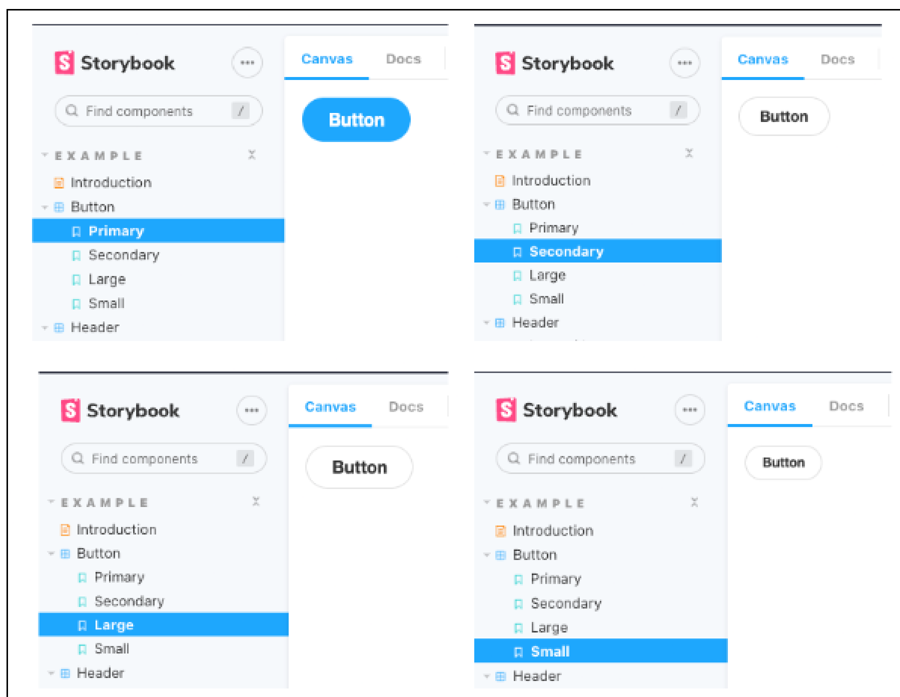


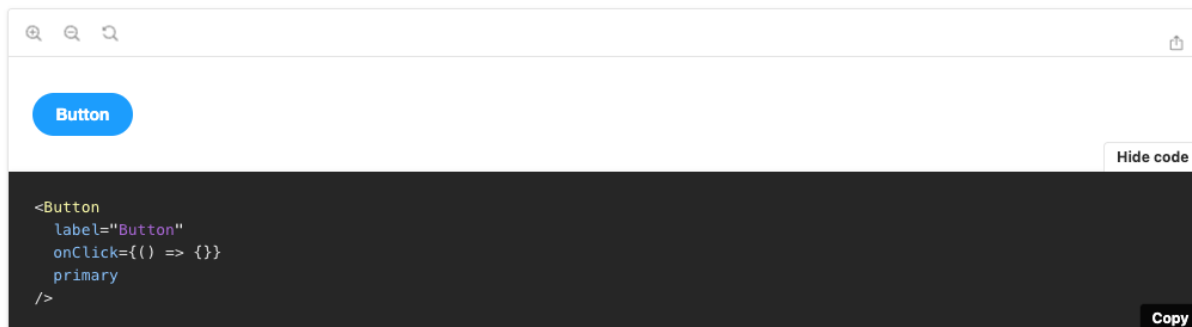

**Figura 20. *Stories* del componente *Button***

Tomando como ejemplo el componente *Button*, en la figura 20 se puede apreciar que este tiene 4 “elementos”: Primary, Secondary, Large y Small. A estos elementos se les conoce como *stories* o Historias y se encargan de describir de forma rápida el resultado de manipular 1 ó varios controles (propiedades o props) como el tamaño o el tipo de botón, según se requiera.

En la pestaña llamada Docs y como se muestra en la figura 21 se puede encontrar una documentación rápida de las *props* del componente, así como su valor por defecto, se puede manipular la *prop* e incluso generar el código necesario para asegurarse que lo que se está viendo en el *Storybook* sea lo mismo que se va a instanciar en el proyecto donde se use ese componente.

**Button**

Primary UI component for user interaction




| Name | Description | Default | Control |
|---|---|---|---|
| primary | Is this the principal call to action on the page? `bool` | `false` | False True |
| label* | Button contents `string` | - | Button |
| backgroundColor | What background color to use `string` | - | Choose color... |
| size | How large should the button be? `'small'` `'medium'` `'large'` | `'medium'` | small / medium / large |
| onClick | Optional click handler `func` | - | - |

**Figura 21. Documentación generada por *Storybook***

### 5.3. Migración de componente *Button*

Para esta etapa donde se migran los componentes existentes de *JavaScript* alojados en el repositorio de *BitBucket* al repositorio de *GitHub* en lenguaje *TypeScript* se comienza con la identificación de las *props* que hacen parte de cada componente.

Debido a que los componentes ya han sido desarrollados en *JavaScript* en años anteriores por desarrolladores de la empresa, el trabajo de migración de estos componentes a TypeScript consiste en identificar las *props* y el tipo de dato que se espera de cada uno de estos componentes, actualizar la sintaxis de *TypeScript* con el

uso de interfaces (*interface*) y añadir la definición de cada prop como comentario a la interfaz para ser usada como documentación.

Para efectos prácticos se toma como ejemplo el componente *Button* para explicar el procedimiento que se realizó con cada uno de los nueve componentes identificados en la sección 4.1.4.

### 5.3.1 Componente *Button en JavaScript*

Como primer acercamiento al componente *Button* ya existente en el repositorio de la empresa, se realiza un listado de las *props* que son usadas dentro del código del botón, estas *props* son fácilmente identificables porque tienen el prefijo "*props." antes de la variable, un ejemplo de esto se puede notar en la figura 22 en la cual se pueden identificar algunas props como:*

- *props.type*
- *props.tabIndex*
- *props.style*
- *props.disabled*
- *props.className*
- *props.onClick*
- *props.children*
- *props.Tooltip*
- *props.placement*

```
10  const renderButton = (props) => {
11      switch (props.type) {
12          default:
13          case 'button':
14              return (
15                  <button
16                      tabIndex={props.tabIndex}
17                      style={{ ...props.style }}
18                      className={cx('btn', props.disabled && 'btn-disabled', props.className)}
19                      disabled={props.disabled}
20                      onClick={(e) => {
21                          if (props.onClick) props.onClick(e);
22                      }}
23                  >
24                      {props.children}
25                  </button>
26              );
27          case 'link':
28              return (
29                  <NavLink
```

↓

```
72  const Button = (props) => {
73      return (
74          <Fragment>
75              {props.tooltip ? (
76                  <Tooltip title={props.tooltip} placement={props.placement || 'bottom'}>
77                      <span>{renderButton(props)}</span>
78                  </Tooltip>
79              ) : (
80                  renderButton(props)
81              )}
82          </Fragment>
83      );
84  };
85
86  export default Button;
```

**Figura 22. Parte del código del componente *Button en JavaScript* donde se aprecian algunas props**

Además, dentro del análisis del código del componente se pudo notar que este componente *Button* es utilizado para renderizar otros tipos de componente como “link”, “icon” o “span” por medio de:

- prop.type=”button”,
- props.type=”link”,
- prop.type=”icon”
- props.type=”span”

Esto terminó siendo una mala práctica de parte de los desarrolladores de la empresa que han venido actualizando este componente año tras año para satisfacer necesidades del momento y que posiblemente solo fue usado pocas veces. Como este componente hace referencia a un botón, solo la funcionalidad del botón es lo que se debería tener en dicho componente por lo que los componentes “link”, “icon” y “span” terminan siendo componentes diferentes dentro de la librería *NPM* que se está creando.

Una vez identificados los *props* que son usados por el componente *Button*, se descartan las *props* que no tienen relación directa con el componente como “props.Tooltip” o “props.placement” ya que estos hacen parte de un componente externo al botón llamado Tooltip.

### 5.3.2. Nuevo componente Button migrado a TypeScript

Teniendo en cuenta los sketchs[2] que crea el equipo de diseño mostrados en la figura 23, Se crean nuevas *props*:

- **variant**: Para escoger entre las variantes posibles como *“primary”*, *“secondary”*, *“tertiary”* o *“warning”*
- **size**: Para estandarizar tamaños por defecto de los botones como *“small”*, *“medium”* o *“large”*

[2] Sketch: se refiere a un dibujo o boceto utilizado para explorar ideas y conceptos de diseño de manera rápida y eficiente y se emplean en la empresa para definir gráficamente como debería verse un componente

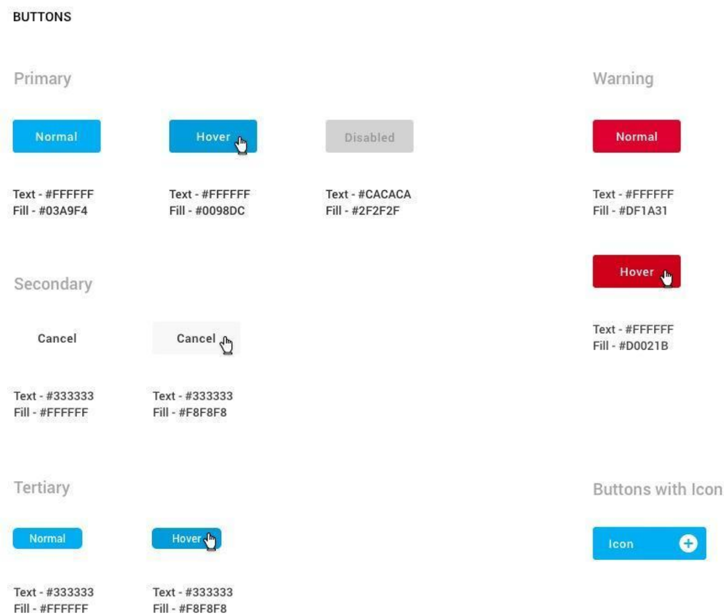



**Figura 23. Sketch del componente *Button* realizado por el equipo de diseño**

Con esta información se crea una interfaz, que en TypeScript es una estructura que define la forma o estructura que deben tener los objetos en un programa y estas ayudan en la detección de errores en tiempo de compilación, ya que permiten verificar que los objetos tienen las propiedades y métodos necesarios antes de ejecutar el código y también ayudan a documentarlo, ya que las interfaces describen la estructura de los objetos que se utilizan en este.

Es importante notar que en la interfaz por medio del uso del signo “?” se puede indicar al componente si una *prop* es opcional, en caso de que el símbolo “?” no se encuentre antes de la definición del tipo de dato, esta *prop* es tomada como obligatoria. De la figura 24 se puede notar que las *props* obligatorias de este componente son: children y onClick y las *props* opcionales son: variant, size, allCaps, isDisabled, icon, style y className.

La interfaz creada para el componente *Button* con nombre *Button*Props se puede observar en la figura 24, en la cual se identifican las únicas *props* que puede recibir el

componente, así como el tipo de dato y los comentarios (documentación) que describen la funcionalidad de estas. También se puede notar que *props* como "*className*" son del tipo *string*, pero *props* como "*variant*" o "*size*" que a pesar que también son tipo *string* se puede reducir las opciones que pueden tomar estas *props* a valores como *"primary"* o *"secondary"* para *"variant"* y *"small"* o *"medium"* para *"size"* logrando reducir las posibilidades de valores que puede tomar una *prop* de tipo *string.*

```
3   export interface ButtonProps {
4     /**
5      * Specify the content of the Button
6      */
7     children: string;
8     /**
9      * Specify the kind of Button you want to create
10     */
11    variant?: "primary" | "secondary" | "tertiary" | "warning";
12    /**
13     * Specify the size of the Button you want to create
14     */
15    size?: "xsmall" | "small" | "medium" | "large";
16    /**
17     * If true, the text inside the button is rendered as uppercase
18     */
19    allCaps?: boolean;
20    /**
21     * Specify whether the Button should be disabled, or not
22     */
23    isDisabled?: boolean;
24    /**
25     * Add icon by passing a, icon name from material UI
26     */
27    icon?: iconNames;
28    /**
29     * Modify default css by passing new style object
30     */
31    style?: React.CSSProperties;
32    /**
33     * Specify an optional className to be added to your Button
34     */
35    className?: string;
36    /**
37     * Action taken when the button is clicked
38     */
39    onClick: () => void;
40  }
```

**Figura 24. Interfaz que define el tipo de dato de las *props* del nuevo componente *Button***

Esta interfaz es guardada en un archivo aparte con el nombre de *Button*.types.ts que posteriormente es importada en el archivo principal del componente *Button* migrado a TypeScript llamado *Button*.tsx que es mostrado en la figura 25 en la línea 3.

```
1   import React, {FC} from "react";
2   import "./Button.scss";
3   import { ButtonProps } from "./Button.types";
4   import { Icons } from "../Icons";
5
6   export const Button: FC<ButtonProps> = ({
7     children = '',
8     size = "medium",
9     variant = "primary",
10    allCaps = false,
```

**Figura 25. Inicio del código del nuevo componente *Button migrado a TypeScript***

En la figura 25 también se puede observar que en la línea 6 se hace una asignación de las *props "ButtonProps"* al componente *Button* con por medio de la función de *React* FC importada en la línea 1, donde FC hace referencia a *"Functional Component"* (Componente Funcional) y es una forma de definir componentes utilizando una función en lugar de una clase, y de esta forma se establece las únicas *props* que puede recibir este componente. En las líneas inferiores, que se puede ver con más claridad en la figura 26, se realiza una importación de las *props* que son usadas dentro del componente y a su vez a todas estas *props* con excepción de onClick se les asigna un valor por defecto en caso de que estas no se pasen al componente por ser opcionales.

```
6  export const Button: FC<ButtonProps> = ({
7    children = '',
8    size = "medium",
9    variant = "primary",
10   allCaps = false,
11   isDisabled = false,
12   icon = "",
13   className = "",
14   style = {},
15   onClick,
16 }) => {
17
18   return (
19     <button
20       type="button"
21       style={style}
22       className={[
23         "button-ejemplo-practicas",
24         `button-ejemplo-practica-${variant}-${theme}`,
25         `button-ejemplo-practica-${size}`,
26         `${icon && "button-ejemplo-practica-icon"}`,
27         `${isDisabled && "button-ejemplo-practica-disabled"}`,
28         `${className}`
29       ].join(" ")}
30       disabled={isDisabled}
31       onClick={onClick}
32     >
33       <div className="button-ejemplo-practica-content">
34         <span className={`${!icon ? "button-ejemplo-practica-text" : "button-ejemplo-practica-text-icon"}`}>
35           {allCaps ? children.toUpperCase() : children}
36         </span>
37         {icon && <Icons name={icon} color={variant != "secondary" ? "white":"black"} size="inherit" />}
38       </div>
39     </button>
40   );
41 };
```

**Figura 26. Valores por defecto y *classNames* del nuevo componente *Button* migrado a *TypeScript***

En la figura 26 se puede observar el código del nuevo componente *Button* migrado a *TypeScript* el cual retorna un elemento de HTML <button> pero cambiando sus *props* de acuerdo a las *props* definidas en la interfaz del componente mostrada en la figura 24. Si se compara este nuevo componente en *TypeScript* con el componente antiguo en *JavaScript* mostrado en la figura 22 se puede notar un código mucho más limpio y con menos líneas de código.

A continuación se muestra una sección del archivo de estilos donde se han cambiado los nombres de los classNames utilizados y se usa el prefijo de “button-ejemplo-practicas-” para ejemplificar. Los classNames hacen referencia a un objeto de css que contiene estilos que permite cambiar la apariencia del botón al

momento de renderizar en pantalla, un elemento de HTML puede recibir varios classNames y de acuerdo con la unión de estos cambios de estilos, la visualización del componente es diferente. De la misma figura 26 se aprecia que siempre se recibe un className llamado “button-ejemplo-practicas” pero dependiendo del valor de la *prop* variant, se puede recibir el className “button-ejemplo-practicas-primary”, “button-ejemplo-practicas-secondary”, “button-ejemplo-practicas-warning”, etc, según sea el caso, igualmente pasa con la *prop* size, se puede recibir “button-ejemplo-practicas-small”, “button-ejemplo-practicas-medium”, etc. Estos estilos se encuentran definidos en el archivo de nombre *Button*.scss y alguna de estas definiciones se pueden observar en la figura 27.

```
.button-ejemplo-practicas{
    font-family: "Arial";
    border: 0;
    font-size: 13px;
    overflow: hidden;
    cursor: pointer;
    font-weight: 500;
    font-style: normal;
    letter-spacing: 0.54px;
    text-align: center;
    border-radius: 3px;
}
```

```
.button-ejemplo-practicas-primary{
    color: white;
    background-color: blue;

    &:hover{
        background-color: darkblue;
        transition: 0.5s;
    }

    &:active{
        background-color: darkblue;
        transition: 0.5s;
    }
}
```

```
.button-ejemplo-practicas-xsmall{
    font-size: 10px;
    padding: 9px 12px;
}

.button-ejemplo-practicas-small{
    font-size: 12px;
    padding: 10px 16px;
}
```

**Figura 27. Extracto de classNames del archivo *Button*.scss**

### 5.4. *Storybook* del componente

Al finalizar esta parte en el proceso de migración a *TypeScript*, se cuenta con todo lo necesario para renderizar el componente, pero como este proyecto de *React* solo contiene componentes visuales se hace uso de la herramienta *Storybook* que permite la renderización de forma aislada del componente, la manipulación de las *props* descritas en la interfaz y acceder a una documentación breve definida en los comentarios de cada *prop* de la interfaz.

Para utilizar la herramienta *Storybook* se crea un archivo con el nombre *Button*.stories.tsx ya que esta se encarga de buscar los archivos con extensión ".stories.tsx" para ser renderizados de forma aislada. Cada archivo ".stories.tsx" puede contener varias *stories* donde cada *story* define una vista diferente del componente.

```
TS Button.stories.tsx ×
src > stories > TS Button.stories.tsx > ...
1   import React from "react";
2   import { ComponentStory, ComponentMeta } from "@storybook/react";
3   import { action } from "@storybook/addon-actions";
4   import { Button } from "../components/Button";
5
6   export default {
7     title: "Components/Button",
8     component: Button,
9     argTypes: {
10      variant: {
11        control: "select",
12        options: ["primary", "secondary", "tertiary", "warning"],
13      },
14      icon: {
15        control: "select"
16      },
17      children: {
18        control: 'text'
19      }
20    },
21  } as ComponentMeta<typeof Button>;
22
23
24
25  const Template: ComponentStory<typeof Button> = (args) => {
26    return (
27      <Button {...args} />
28    )
29  }
```

**Figura 28. Parte inicial del archivo "Button.stories.tsx"**

A continuación se describe el contenido general de un archivo con extensión ".stories.tsx" usando como ejemplo el componente *Button*.

Usando la figura 28 como base:

En la línea 2 y 3 se realizan importaciones de funciones propias de *Storybook* para un correcto funcionamiento.
En la línea 4 se importa el componente al cual se le va a realizar las *stories*, en este caso se importa el componente *Button*.
De la línea 6 a la línea 21 se realiza una descripción de la *story* (historia)

- Línea 7: define el título y ubicación dentro del catálogo de componentes.
- Línea 8: define el componente a renderizar (importado en línea 4).
- línea 9-19: define el tipo de control que se quiere para ciertas props
    - variant: un selector con las opciones allí descritas
    - icon: un selector.
    - children: tipo texto
- línea 21: exporta por defecto la *story* tipo componente, en este caso, tipo *Button*

De la línea 25-29 se crea un *template* que define las *stories* y le pasa las *props* por medio de la variable *args* al componente.

Para crear las *stories* individuales se exporta una constante con el nombre que sea descriptivo y a la cual se vincula con el *template* creado anteriormente usando la función .bind() como se muestra en la línea 31 de la figura 29 para la *story* llamada *Primary* haciendo referencia a un componente *Button* con *prop variant="primary".* Para asignar valores a las *props* de una *story* se hace por medio de la *prop "args"* que se le asigna a la constante previamente creada, en esta ocasión (figura 31, línea 32-39) se asigna a *Primary.args* un objeto que contiene las *props* descritas en la interfaz del componente *Button*, en caso de no asignar un valor a las *props* definidas como opcionales, su valor será tomado por el que ha sido designado por defecto y en caso de no asignar un valor a las *props* obligatorias se generará un error que indicará que *prop* queda faltando por inicialización. En la figura 29 también se enseñan como se crearon las *stories* para *"Secondary"* y *"Tertiary"*.

Este proceso se repite la cantidad de veces que se requiera con la intención de recrear las posibles variaciones de las *props* que contenga el componente y que se requiera

mostrar, además se cuenta con la opción de crear una sola *story* y dejar que el usuario interactúe con los controles de las *props* para generar las variaciones que desee.

```
31    export const Primary = Template.bind({});
32  ∨ Primary.args = {
33      variant: "primary",
34      children: 'primary',
35      allCaps: false,
36      isDisabled: false,
37      icon: "",
38      onClick: action("Primary Link pressed"),
39    };
40
41    export const Secondary = Template.bind({});
42  ∨ Secondary.args = {
43      variant: "secondary",
44      children: 'Secundary',
45      allCaps: false,
46      isDisabled: false,
47      icon: "",
48      onClick: action("Secondary Link pressed"),
49    };
50
51    export const Tertiary = Template.bind({});
52  ∨ Tertiary.args = {
53      variant: "tertiary",
54      children: 'Tertiary',
55      allCaps: false,
56      isDisabled: false,
57      icon: "",
58      onClick: action("Tertiary Link pressed"),
59    };
```

**Figura 29. *Stories Primary, Secondary* y *Tertiary* del componente *Button***

### 5.4.1. Ejecución de *Storybook*

Cuando los archivos con extensión ".stories.tsx" se han creado se procede a la ejecución del *Storybook* con la intención de visualizar de forma aislada los componentes. Para esto desde la terminal se ejecuta el comando mostrado en la figura 17 de la sección 5.2, se espera un momento y cuando este termina la ejecución, en la terminal aparece un mensaje como el mostrado por la figura 18 de la sección 5.2 que indica que la herramienta está disponible para ser usada en la url "http://localhost:6006"

### 5.4.2. Controles interactivos de un componente

Al ingresar a la url disponible (“http://localhost:6006”) nos encontramos con un catálogo como el mostrado en la figura 30 donde en la parte izquierda y resaltado con un color azul se encuentran los componentes que tienen asociado un archivo con extensión “.stories.tsx” en este caso solo está disponible el componente *Button* y sus *stories “Primary”, “Secondary”, “Tertiary”, “Warning”* y *“With Icon”,* en el centro resaltado con color verde se encuentra el componente renderizado de forma aislada y que cambia de acuerdo a la *story* seleccionada o por el cambio de alguna *prop* que se encuentra en la parte derecha resaltada con color rojo.

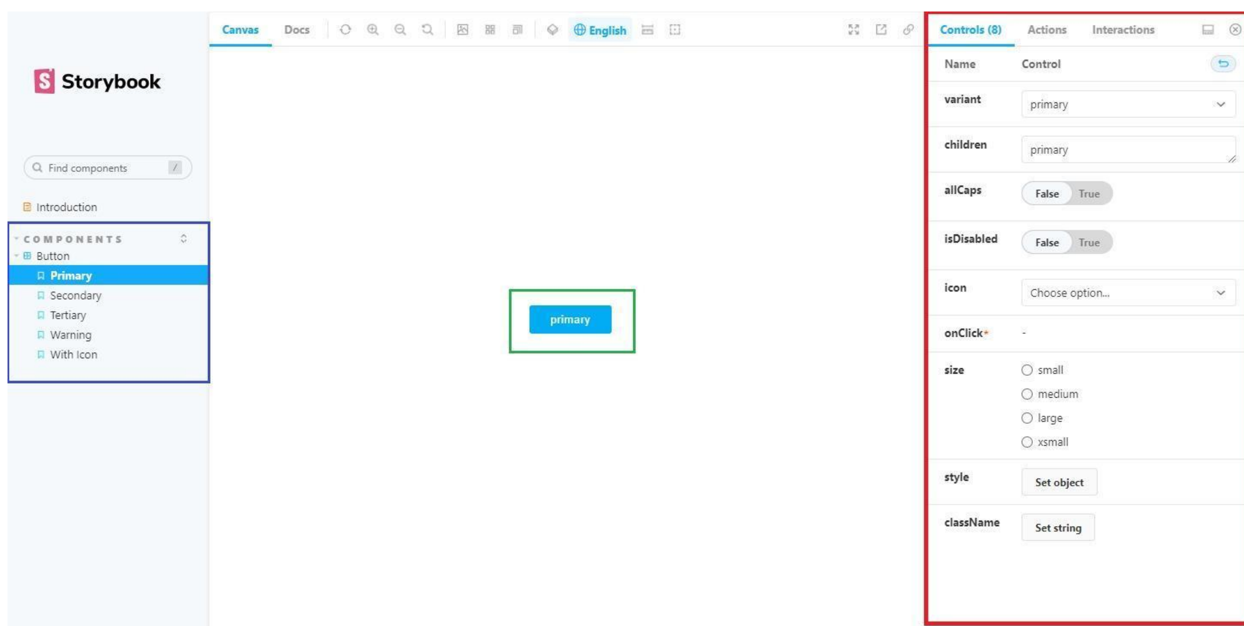



**Figura 30. *Storybook* con el componente *Button* y los controles de sus *props***

En la figura 31 se puede apreciar el comportamiento del componente renderizado en la mitad de la pantalla cuando se selecciona una *story* diferente, en este caso las *stories* de *“Secondary”*, *“Warning”* y *“With Icon”*

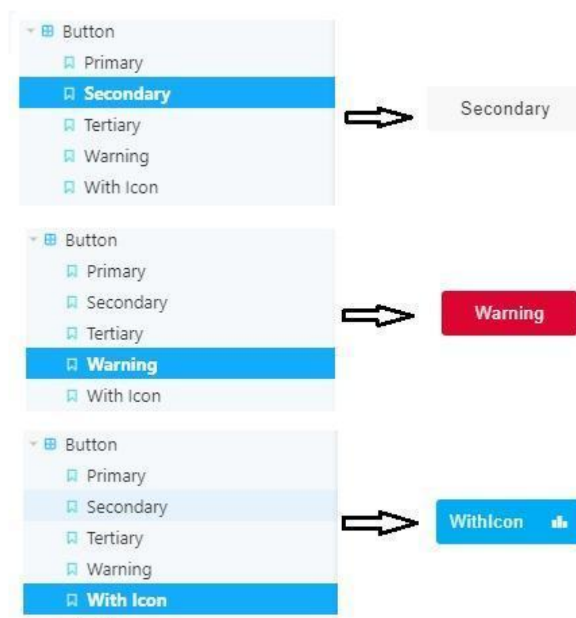



**Figura 31. Diferentes render del componente *Button* dependiendo de la *story* seleccionada**

### 5.4.3. Documentación de un componente

Cuando se selecciona la pestaña Docs como se muestra en la figura 32, se accede a la sección de documentación del componente, donde se puede encontrar una tabla con cuatro columnas resaltadas en color azul. Estas columnas corresponden al nombre de cada prop, la descripción de dicha prop, como recordatorio esta descripción proviene de la descripción hecha anteriormente en la interfaz del componente, la tercera columna indica el valor por defecto asignado y la cuarta columna son los controles interactivos a los que se puede acceder desde la documentación.

**Figura 32. Documentación del componente *Button***

Otra opción y ventaja con la que cuenta esta documentación es la posibilidad de interactuar con los controles y cuando se esté satisfecho con el resultado, esta documentación puede generar el código necesario para recrear ese estado visual que se obtuvo en otro proyecto de *React* siempre y cuando estos componentes existan. En la sección 5.6. se explica la forma en la que se puede crear una librería *NPM* de *React* con estos componentes y que posteriormente puede ser instalada en cualquier proyecto que utilice *React* y hacer uso de este catálogo y documentación, incluyendo la generación del código necesario para instanciar los componentes tal y como se obtienen al interactuar con los controles. Un ejemplo de cómo se genera este código se muestra en la figura 33 donde los controles resaltados en verde son modificados hasta obtener el renderizado del componente *Button* resaltado en azul, lo que resulta en el código resaltado en rojo que puede ser usado en otro proyecto.

Button

EJEMPLO PRACTICAS

Hide code

```
<Button
  allCaps
  icon="dashboard"
  onClick={function noRefCheck(){}}
  variant="warning"
>
  Ejemplo practicas
</Button>
```

Copy

| Name | Description | Default | Control |
| --- | --- | --- | --- |
| variant | Which type of button you need ? "primary" "secondary" "tertiary" "warning" | "primary" | warning |
| children | what should the button say inside ? string | "" | Ejemplo practicas |
| allCaps | Should the string given be in Uppercase ? boolean | false | False True |
| isDisabled | Should the button be disabled ? boolean | false | False True |
| icon | Add icon "" "network_check" "info" "person" "videocam" "leaderboard" "insights" "dashboard" Show 20 more... | "" | dashboard |

**Figura 33. Generación de código necesario para replicar componente con *props* modificadas desde controles interactivos**

### 5.5. Creación de pruebas unitarias

En esta sección, se aborda la creación de pruebas unitarias, una práctica fundamental en el desarrollo de software que permite asegurar la calidad del código que se desarrolla. Las pruebas unitarias permiten evaluar de manera sistemática y automatizada el comportamiento de pequeñas partes de nuestro código, conocidas como unidades, y verificar que éstas funcionan correctamente en distintos escenarios. Además, las pruebas unitarias ayudan a detectar errores y *bugs* de manera temprana, lo que reduce el costo y el tiempo necesario para arreglarlos.

Para ejemplificar la creación de pruebas unitarias se continúa con el componente *Button* para explicar todo el proceso realizado.

Las pruebas unitarias en este proyecto se llevan a cabo con la ayuda del framework Jest (Sección 4.5.1.). Para escribir pruebas en Jest se debe crear un archivo con extensión ".test.tsx" y estos archivos deben contener al menos una función que defina la prueba utilizando la función test() o it().

Se parte creando el archivo "Button.test.tsx" el cual contiene todas las pruebas realizadas sobre el componente. Tomando como referencia la figura 34, al inicio de archivo (línea 2) se importa la librería de *testing* (testing-library/jest-dom).
En la línea 3 se importan las funciones:

- **render**: Permite renderizar el componente de forma aislada (no visual)
- **cleanup**: Permite hacer una limpieza del render.
- **fireEvent**: Permite disparar eventos como clicks, ingreso de texto, etc

En la línea 5 se importa el componente a ser testeado, en este caso el componente *Button.*
A partir de la línea 7 se comienza a crear las pruebas para el componente y en esta misma línea se describe cuál es el componente que se va a testear en este archivo.
Al inicio de las prueba se instala una función afterEach() que se encarga de ejecutar la función *cleanup* después de ejecutada cada prueba en el caso de que exista más de una.

En la definición de cada prueba, se puede elegir entre la función 'test()' o 'it()'. Ambas realizan la misma función, pero se ha decidido usar la función 'it()' porque en inglés permite describir de manera más clara qué parte del código se va a probar. Por ejemplo, en la línea 9 se describe la primera prueba con la función 'it()' que dice *"it('Should render the Button')"*, que traducido al español significa *"Debería renderizar el botón"*. Esto aumenta la trazabilidad de la prueba, ya que es más clara en lo que debe hacer.

```
1   import React from "react";
2   import "@testing-library/jest-dom";
3   import { render, cleanup, fireEvent } from "@testing-library/react";
4
5   import {Button} from ".";
6
7   describe("Running Test for Button", () => {
8     afterEach(cleanup);
9     it("Should render the button", () => {
10      const {getByRole} = render(<Button variant="primary" onClick={() => {}} >test button</Button>);
11      const button = getByRole("button", { name: "test button" })
12      expect(button).toBeInTheDocument();
13    });
14
15    it("Should render the button as variant primary", () => {
16      const {getByRole} = render(<Button variant="primary" onClick={() => {}} >test button</Button>);
17      const button = getByRole("button", { name: "test button" })
18      expect(button).toHaveClass("button-ejemplo-practicas-primary");
19    });
20
21    it("Should render the button as variant secondary", () => {
22      const {getByRole} = render(<Button variant="secondary" onClick={() => {}} >test button</Button>);
23      const button = getByRole("button", { name: "test button" })
24      expect(button).toHaveClass("button-ejemplo-practicas-secondary");
25    });
```

**Figura 34. Definición inicial de pruebas unitarias**

En la línea 10 se obtiene la función “getByRole” al ejecutar la función “render()” del componente *Button* que tiene el texto *“test Button”*, con *props variant=”primary”*, y como *onClick* es una *prop* obligatoria pero no se va a probar todavía, se pasa una función vacía.

“getByRole()” es una función proporcionada por la biblioteca ”testing-library/react”, la cual es utilizada en Jest para obtener un elemento de la interfaz de usuario en función de su "rol" o "tipo", como "botón", "enlace", "entrada de texto", entre otros. Como el componente que se está probando es un tipo *Button* y existe un “rol” definido se usa esta función. Como se observa en la línea 11 “getByRole()” recibe dos parámetros, el primero es el rol que se busca (“button”) y segundo parámetro es el nombre del botón, que en esta ocasión seria *“test Button”*, el resultado de la ejecución de esta función retorna un componente al cual se ha llamado *“button”*.

Para finalizar la primera prueba en la línea 12 se ejecuta la la función "expect(button).toBeInTheDocument()" que comprueba si el elemento *"button"* de la interfaz de usuario está presente en la página, y si no lo está, la prueba fallará.

La segunda y tercera pruebas tienen una definición similar a la primera, pero al ejecutar la última función *expect("...").toHaveClass("..")*, se espera que el elemento *"button"* tenga la clase *"button-ejemplo-practicas-primary"* y *"button-ejemplo-practicas-secondary"*, respectivamente. Esto se debe a que al renderizar el componente para estas pruebas (líneas 16 y 22), se utiliza la propiedad *variant="primary"* para la segunda prueba y *variant="secondary"* para la tercera prueba. En caso de que el elemento no contenga las clases mencionadas, las pruebas fallarán. En cambio, si el elemento contiene las clases correspondientes, las pruebas pasarán.

Las pruebas realizadas para verificar el uso correcto de las clases de tamaño se presentan en la figura 35, donde se prueba para tamaño *"small"* y *"medium"*.

```
45    it("Should render the button with size small", () => {
46      const {getByRole} = render(<Button variant="primary" size="small" onClick={() => {}} >test button</Button>);
47      const button = getByRole("button", { name: "test button" })
48      expect(button).toHaveClass("button-ejemplo-practicas-small");
49    });
50
51    it("Should render the button with size medium", () => {
52      const {getByRole} = render(<Button variant="primary" size="medium" onClick={() => {}} >test button</Button>);
53      const button = getByRole("button", { name: "test button" })
54      expect(button).toHaveClass("button-ejemplo-practicas-medium");
55    });
```

**Figura 35. Pruebas unitarias para verificar empleo de *className* de tamaño.**

En la figura 36 se muestran otras pruebas que verifican la funcionalidad de distintas propiedades del componente *Button*. La primera prueba de la línea 63 verifica si el componente se renderiza como deshabilitado. Para ello, se le pasa al componente la propiedad *isDisabled* y se espera que la función *"expect(button).toBeDisabled()"* pase la prueba en caso de que el componente esté realmente deshabilitado, y falle en caso contrario.

Para realizar pruebas sobre funciones, Jest permite crear funciones de prueba asignadas como jest.fn(). En las líneas 70 y 79 se asigna esta función a la variable *handleClick*, y es esta variable la que se pasa como *prop onClick* para validar su ejecución. Utilizando la propiedad de expect() llamada *".toHaveBeenCalledTimes()"*, se espera que la función *handleClick* se haya ejecutado el número de veces especificado como argumento.

En las dos últimas pruebas se simula un evento de tipo *click* sobre el elemento *"button"* mediante la función *fireEvent.click(button).* Antes del disparo del evento, se espera que la función *handleClick* no haya sido llamada ninguna vez. Una vez se dispare el evento, se espera que la función haya sido llamada una única vez. En la última prueba se está haciendo uso de la propiedad *isDisabled*, por lo que se espera que después del disparo del evento de tipo *click*, la función *handleClick* no se haya llamado ninguna vez ya que el componente debería renderizar como deshabilitado.

```
63    it("Should render the button as disabled", () => {
64      const {getByRole} = render(<Button variant="primary" size="xsmall" onClick={() => {}} isDisabled >test button</Button>);
65      const button = getByRole("button", { name: "test button" })
66      expect(button).toBeDisabled();
67    });
68
69    it("Should call the onClick function when clicked ", () => {
70      const handleClick = jest.fn()
71      const {getByRole} = render(<Button variant="primary" size="xsmall" onClick={handleClick} >test button</Button>);
72      const button = getByRole("button", { name: "test button" })
73      expect(handleClick).toHaveBeenCalledTimes(0)
74      fireEvent.click(button)
75      expect(handleClick).toHaveBeenCalledTimes(1)
76    });
77
78    it("Should NOT call the onClick function when clicked but isDisabled={true}", () => {
79      const handleClick = jest.fn()
80      const {getByRole} = render(<Button variant="primary" size="xsmall" onClick={handleClick} isDisabled >test button</Button>);
81      const button = getByRole("button", { name: "test button" })
82      expect(handleClick).toHaveBeenCalledTimes(0)
83      fireEvent.click(button)
84      expect(handleClick).toHaveBeenCalledTimes(0)
85    });
86  });
```

**Figura 36. Pruebas unitarias para verificar que el componente *Button* renderiza como deshabilitado y verificar el comportamiento de la función onClick**

Por último para ejecutar las pruebas unitarias y verificar si las pruebas pasan o fallan se debe correr el comando de la figura 37:

```
npm run test Button
```

**Figura 37. Comando para ejecutar las pruebas unitarias del archivo *Button.test.tsx***

Al terminar la ejecución del comando anterior, y en caso de que todas las pruebas sean aprobadas, se observa una respuesta similar a la de la figura 38:

```
> jest "Button"

PASS  src/components/Button/Button.test.tsx
  Running Test for Button
    √ Should render the button (50 ms)
    √ Should render the button as variant primary (22 ms)
    √ Should render the button as variant secondary (25 ms)
    √ Should render the button as variant tertiary (20 ms)
    √ Should render the button as variant warning (16 ms)
    √ Should render the button with size xsmall (15 ms)
    √ Should render the button with size small (15 ms)
    √ Should render the button with size medium (11 ms)
    √ Should render the button with size large (12 ms)
    √ Should render the button as disabled (9 ms)
    √ Should call the onClick function when clicked  (19 ms)
    √ Should NOT call the onClick function when clicked but isDisabled={true} (12 ms)

Test Suites: 1 passed, 1 total
Tests:       12 passed, 12 total
Snapshots:   0 total
Time:        5.813 s, estimated 6 s
```

**Figura 38. Ejecución de todas las pruebas unitarias del componente *Button* pasando satisfactoriamente**

**5.6. Procedimiento realizado con los otros componentes.**

Como se menciona en al inicio de la sección 5.3 los componentes a migrar son componentes que han sido desarrollados previamente por los desarrolladores de la empresa, por lo que la lógica y estilos que poseen son válidos y no había necesidad de

implementar o crear nueva lógica, el trabajo de migración de *JavaScript* a *TypeScript* consistia en la identificación de los tipos de datos que recibe cada *prop* de cada componente y con esta información crear una interfaz que es la que define los tipos de datos que puede recibir en nuevo componente migrado a *TypeScript*. Conociendo el tipo de dato y la función de cada *prop* se crea una descripción a modo de comentario en la interfaz la cual es usada por la herramienta *Storybook* para generar la documentación de dichas props en la ventana *Docs* de cada componente en el *Storybook*.

Componentes como *Input* y *Modal* fueron los dos únicos componentes que estaban definidos como tipo clase por lo que a estos componentes se aplicó una serie de pasos para convertirlos a tipo función.

Para estos componentes se realizó el siguiente procedimiento:

1. **Convertir la clase en una función:** Se debe eliminar la definición de la clase y el método *render()*, y reemplazarla por una función que acepte las *props* como argumento como se muestra en la figura 39.

```
function NombreComponente(props) {
  // código del componente
}
```

**Figura 39. Definición de componente tipo función**

2. **Eliminar el constructor y el estado:** Los componentes de tipo función no tienen estado local y no necesitan un constructor por lo que se deben eliminar.
3. **Convertir los métodos de ciclo de vida en hooks:** Si el componente de tipo clase utiliza métodos de ciclo de vida [Seccion 4.3.1.3], como *componentDidMount()* o *componentDidUpdate()*, se deben convertir en hooks [Sección 4.3.1.4]. Para ello, se deben utilizar los hooks proporcionados por

*React*, como *useEffect()* o *useState()*. En la figura 40 se observa un ejemplo de cambio de componente tipo clase a componente tipo función, donde el método de ciclo de vida llamado *componentDidMount()* del componente tipo clase es cambiado por el hook *useEffect()* con el segundo parámetro vacío ([ ]), lo que permite realizar la misma función que en este caso sería el de realizar una acción cuando el componente se encuentre montado/renderizado satisfactoriamente.

```
class MiComponente extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      datos: []
    };
  }

  componentDidMount() {
    // código a ejecutar después del montaje del componente
  }

  // código del componente

  render() { return (..) } // retorno del componente
}
```

```
import React, { useEffect } from 'react';

function MiComponente() {
  useEffect(() => {
    // código a ejecutar después del montaje del componente
  }, []);
  // código del componente

  return (...)  // retorno del componente
}
```

**Figura 40. Estructura de componente tipo clase y componente tipo función**

4. **Retornar el "HTML" del componente:** En la función del componente de tipo función, se debe devolver el HTML que se renderiza en la interfaz de usuario.

Para crear los *stories* que representan el componente en el catálogo de *Storybook* se debe instanciar el componente en un archivo aparte (*.stories.tsx*) con valores por defecto en las props previamente identificadas por lo que este proceso es similar al ejemplificado con el componente *Button* en la sección 5.4.

Para la creación de las pruebas unitarias de los otros componentes es similar al mostrado en la sección 5.5 en el sentido que se debe comprobar que el componente o un texto en específico se renderice correctamente con la función

*expect("...").toBeInTheDocument(),* que los estilos se carguen correctamente con la función *expect("...").toHaveClass("..")*, o que las funcionalidades debido a un click del usuario por medio de la función *fireEvent.click("componente")* ejecuten las acciones adecuadas con la función *expect("...").toHaveBeenCalled() o que haya sido llamada el número de veces adecuado con la función expect("...").toHaveBeenCalledTimes(#).*

Debido a lo mencionado en esta sección es que se toma la decisión de explicar detalladamente el procedimiento realizado con el componente *Button* desde la sección 5.3 a la sección 5.5 para definir el procedimiento que se realiza en con los otros componentes definidos en la sección 4.1.4.

### 5.7. Despliegue de librería *NPM*

Al finalizar la migración de los nueve componentes requeridos en este trabajo, se realizó la publicación de la librería *NPM*. Para este paso se contó con la ayuda de una persona que tenía credenciales válidas de la empresa en *npm*js.com ya que la librería sería publicada como una librería privada.
La publicación se realizó siguiendo los siguientes pasos:

1. Tener configurado correctamente el archivo package.json. asegurarse que la propiedad *"name"* contenga un nombre único para la librería, la propiedad *"version"* contenga la versión actual y la propiedad *"private"* tenga el valor *true*. También se debe asegurar de tener una propiedad *"main"* que apunte al archivo principal de tu librería. Un ejemplo de la configuración de este archivo se puede notar en la figura 41.

```json
{
  "name": "nombre-de-la-libreria",
  "version": "0.0.4",
  "description": "Descripción de la librería",
  "main": "dist/index.js",
  "types": "dist/index.d.ts",
  "repository": "https://github.com/tu-usuario/nombre-de-la-libreria",
  "private": true,
  "dependencies": {
    "react": "^17.0.2"
  },
  "devDependencies": {
    "@types/react": "^17.0.30",
    "typescript": "^4.3.5"
  },
```

**Figura 41. Parte del archivo de ejemplo llamado package.json con información de la liberia *NPM***

2. En la terminal, iniciar sesión con la cuenta que posee credenciales válidas de npmjs.com usando el comando “npm login”.
3. Se ejecuta el comando “tsc && npm run build” que se encarga de compilar el código de TypeScript a código JavaScript válido y se genera la versión optimizada de la aplicación.
4. Luego, se ejecuta el comando “npm publish”. Esto crea una nueva versión de la librería y la publica en *NPM*.

## 6. Resultados y Análisis.

Este trabajo se compone de cuatro entregables, como se puede apreciar en los objetivos específicos. Por esta razón, esta sección se divide en subsecciones para explicar y presentar cada uno de los resultados obtenidos durante la práctica académica.

## 6.1 Resultados

### 6.1.1 Repositorio en GitHub de componentes en TypeScript

El repositorio donde se encuentra alojado el código correspondiente a el trabajo de este proyecto es privado y para uso interno de la empresa por lo que el acceso está restringido por usuario y contraseña, pero en la figura 42 se puede notar que el repositorio de GitHub perteneciente a la empresa posee 100 commits o confirmaciones de cambios de código y estos hacen referencia a los cambios que fueron desplegados a la rama production (main) para hacer el primera liberación de la librería NPM.

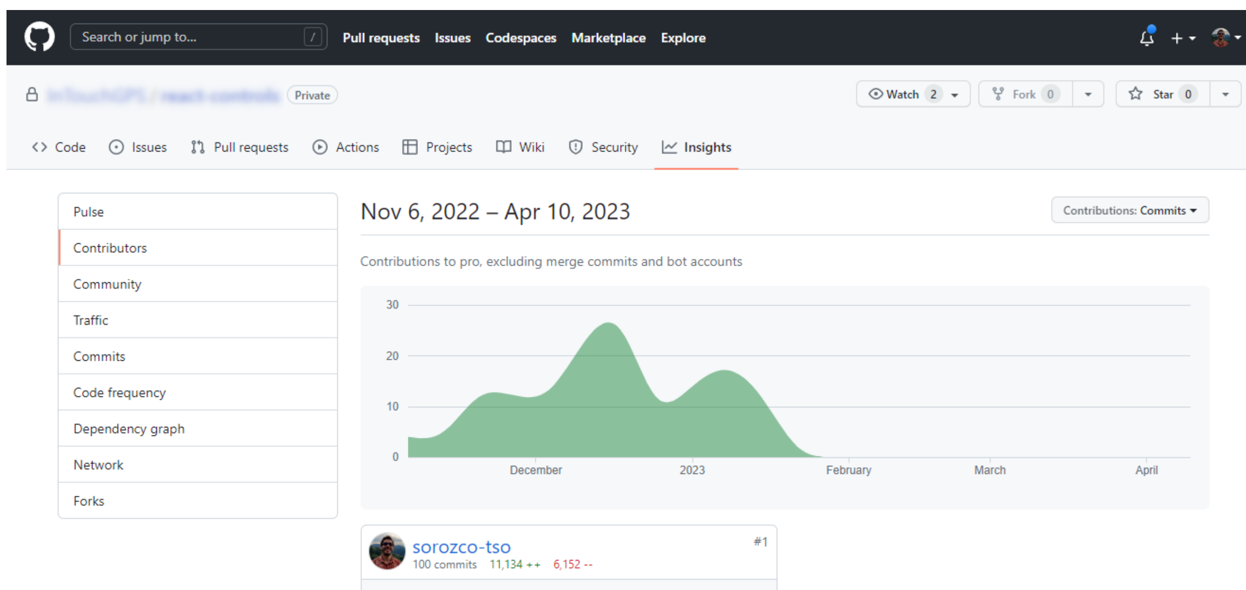



**Figura 42. Repositorio en GitHub con el proyecto desarrollado**

En la Figura 43 se muestra que el 75% del repositorio corresponde a archivos de TypeScript los cuales incluyen los códigos de los componentes, las interfaces y los tests correspondientes. El 18% de los archivos son de SCSS o CSS, que contienen los estilos de los componentes, como colores, tamaños y tipografía, mientras que el 6,6% restante se compone de archivos de JavaScript específicos del proyecto de *React*.

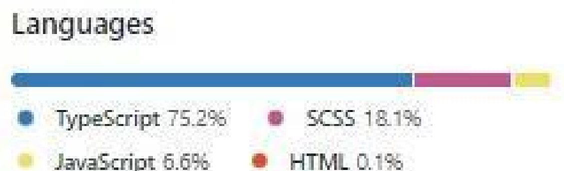



**Figura 43. Lenguajes utilizados en el repositorio del proyecto de prácticas**

### 6.1.2 *Storybook*

Al ingresar la URL donde se encuentra alojado el *Storybook* con el catálogo de los componentes visuales, el usuario se encuentra con una introducción (figura 44) que, en el momento de la entrega del proyecto, cuenta con un texto de relleno *Lorem Ipsum*. Sin embargo, en los días posteriores, el contenido de esta página debería contener información detallada del funcionamiento del *Storybook* y de su documentación, una guía paso a paso para la correcta instalación de la librería *NPM* y demás información que la empresa considere necesaria.

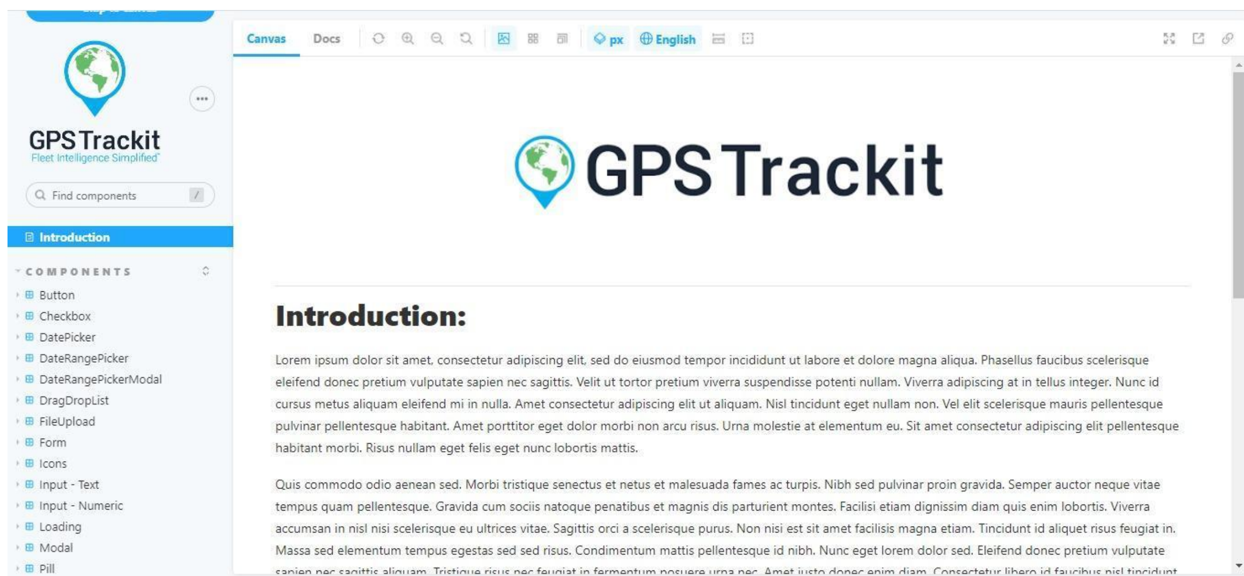



**Figura 44. Página de inicio del *Storybook* con catalogo de componentes**

En la la columna de la izquierda bajo el título de *COMPONENTS* se encuentra la lista de componentes migrados a TypeScript y que se encuentran disponibles en la librería *NPM*. Para la demostración de resultados, se explicarán 3 componentes y la documentación de uno de ellos pero en la sección de anexos se puede encontrar la documentación generada para los demás componentes que fueron migrados en este proyecto.

#### 6.1.2.1 Componentes

**Button:**
El componente *Button* en *React* es un componente que representa un botón en una interfaz de usuario. Este componente se utiliza para activar acciones, enviar formularios, navegar entre páginas, entre otras funciones.

En la pestaña *canvas* del *storybook* se encuentra la vista aislada del componente. Como se puede ver en la figura 45, el componente *Button* se renderiza en la mitad de la pantalla con sus valores por defecto. En la columna de la derecha en la pestaña de *Controls* se pueden observar todas las *props* con las que cuenta el componente.

Por medio de estos controles el usuario puede interactuar en tiempo real con el componente y presenciar el resultado de cambiar uno o varios controles para así conocer el resultado que puede llegar a tener al momento de renderizar el componente en algún proyecto por medio de la librería *NPM* que los contiene.

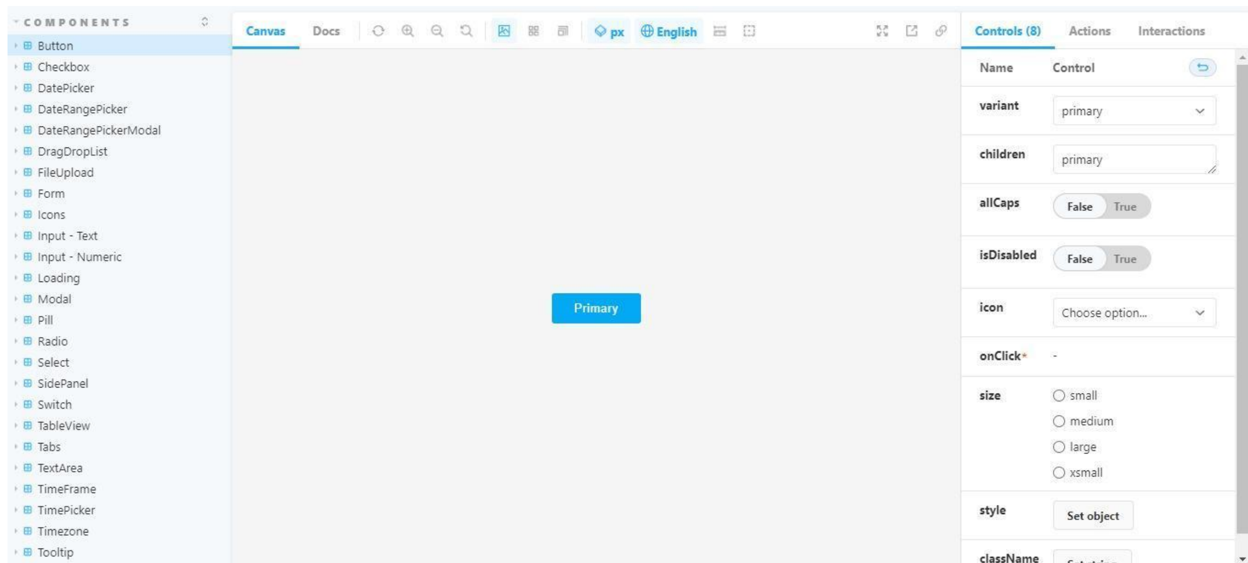



**Figura 45. Componente *Button* en *Storybook* con controles interactivos**

**Input:**

El componente Input en *React* es un componente que representa un campo de entrada de texto en un formulario. Este componente permite al usuario ingresar información, ya sea un texto simple, una dirección de correo electrónico, un número de teléfono, entre otros tipos de datos. Este componente para su diseño más completo puede contar con la integración de otros componentes que también se encuentran en el *storybook* por lo que lo hace un componente compuesto cuando se usan las *props* correspondientes como lo es con el caso del componente *Button*, icono, un tooltip, estos son componentes que también hacen parte de la librería *NPM* y se encuentran en la columna de la izquierda. En la figura 46 se puede observar la pestaña *Canvas* donde se puede interactuar con el componente de forma aislada por medio de los controles interactivos de sus props.

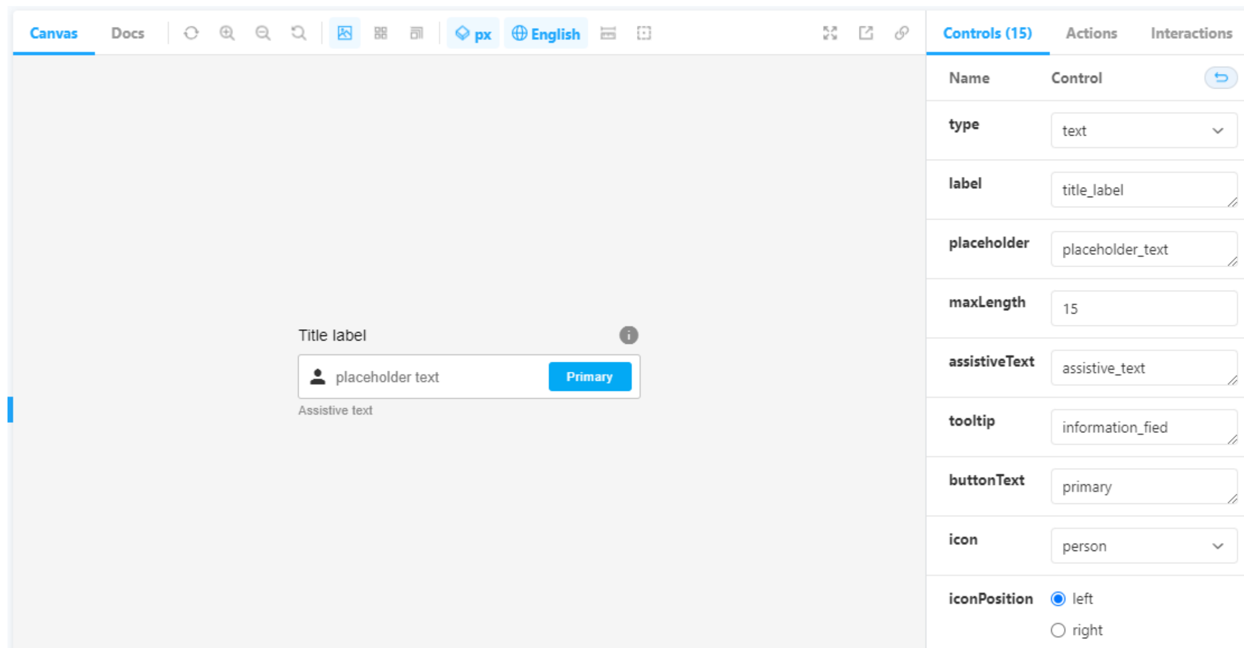



**Figura 46. Componente Input en *Storybook* con controles interactivos**

**Modal:**

Un componente *modal* es un tipo de componente visual utilizado en el desarrollo de interfaces de usuario que se muestra en la parte superior de la interfaz actual y requiere que el usuario interactúe con ella antes de continuar con la aplicación.

Normalmente, los modales se utilizan para mostrar información importante al usuario, como mensajes de error, confirmaciones, alertas, ventanas de autenticación, formularios y otros elementos que requieren una respuesta inmediata del usuario.

En la figura 47, se puede apreciar la pestaña *Canvas* del componente Modal. En este ejemplo, se utilizaron componentes como el input, *Button*, *checkbox* y *Radiobutton* como contenido del modal para ejemplificar un formulario. En la columna de la derecha, el usuario puede encontrar los controles interactivos que le permiten controlar el tamaño del modal mediante la propiedad "*size*", controlar si se renderiza o no el encabezado del modal (ejemplificado con el texto *"Report Setup"*) mediante la propiedad *"hasHeader"*. También puede abrir y cerrar el modal a través de la propiedad

*"isOpen"*, y mediante la propiedad *"isLoading"* puede simular la acción de carga de contenido que pueda ser demorado en descargar desde una base de datos, viendo cómo se comportan los componentes mientras la carga ocurre.

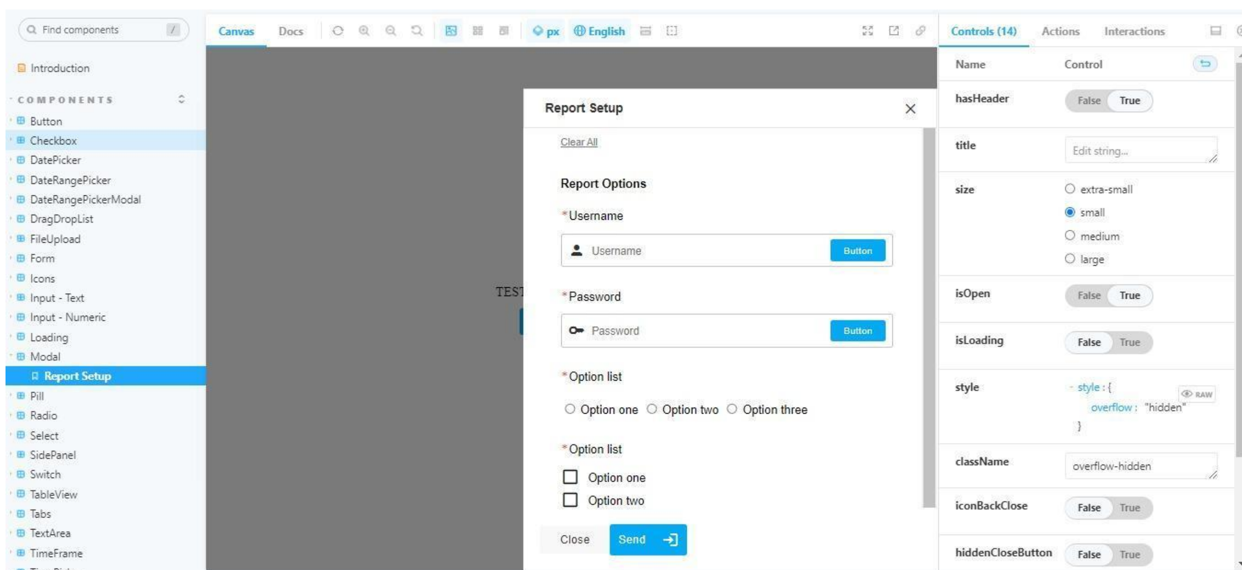



**Figura 47. Componente modal con controles interactivos**

### 6.1.2.2 Documentación

Al momento de seleccionar la pestaña Docs, el usuario se encuentra con la documentación de las *props* que posee el componente. Por medio de la tabla que se observa en la figura 48 el usuario puede conocer la funcionalidad de cada una de ellas, puede conocer si estas contienen un valor por defecto (en caso de que no se pasen al componente) y al igual que en la pestaña de *Canvas*, el usuario tiene la posibilidad de interactuar con las *props* por medio de los controles.

Por medio del botón *“Show code”* la documentación también cuenta con la posibilidad de generar el código (figura 49) necesario para instanciar el componente tal y como lo personaliza el usuario en esta pestaña y se puede garantizar que el resultado será igual siempre y cuando la librería *NPM* sea instalada en el proyecto donde se espera instanciar dicho componente.

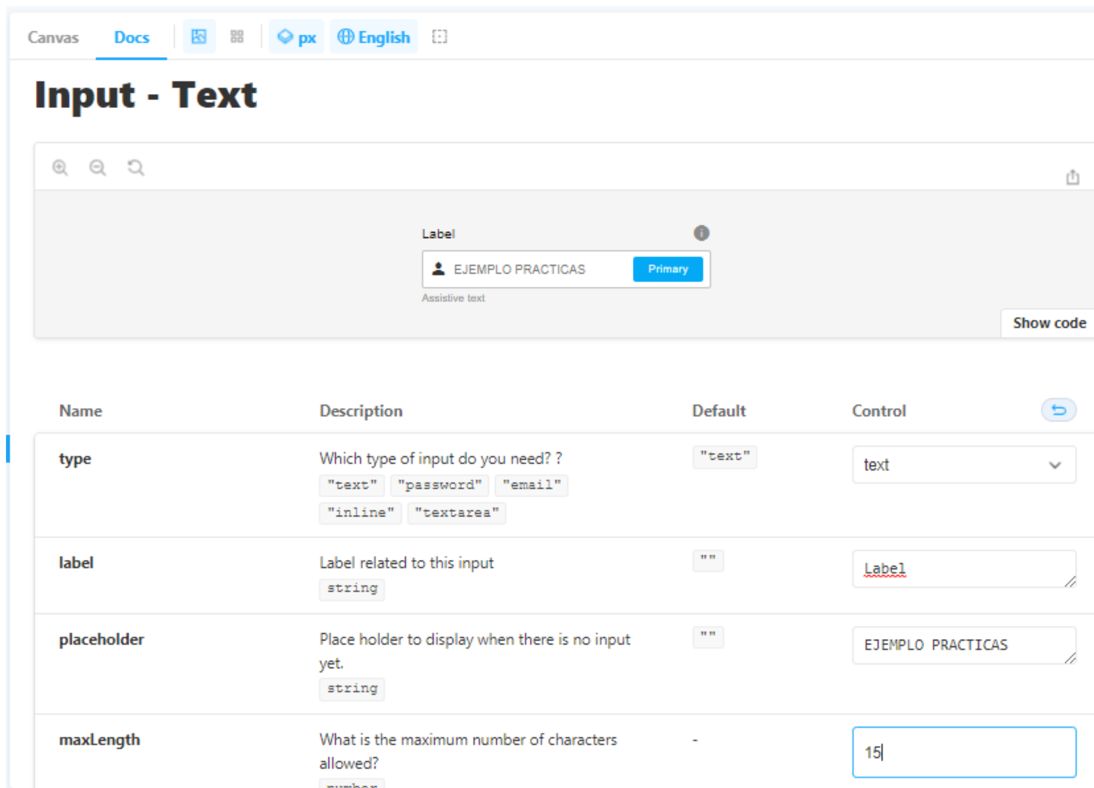



Figura 48. Documentación de componente Input

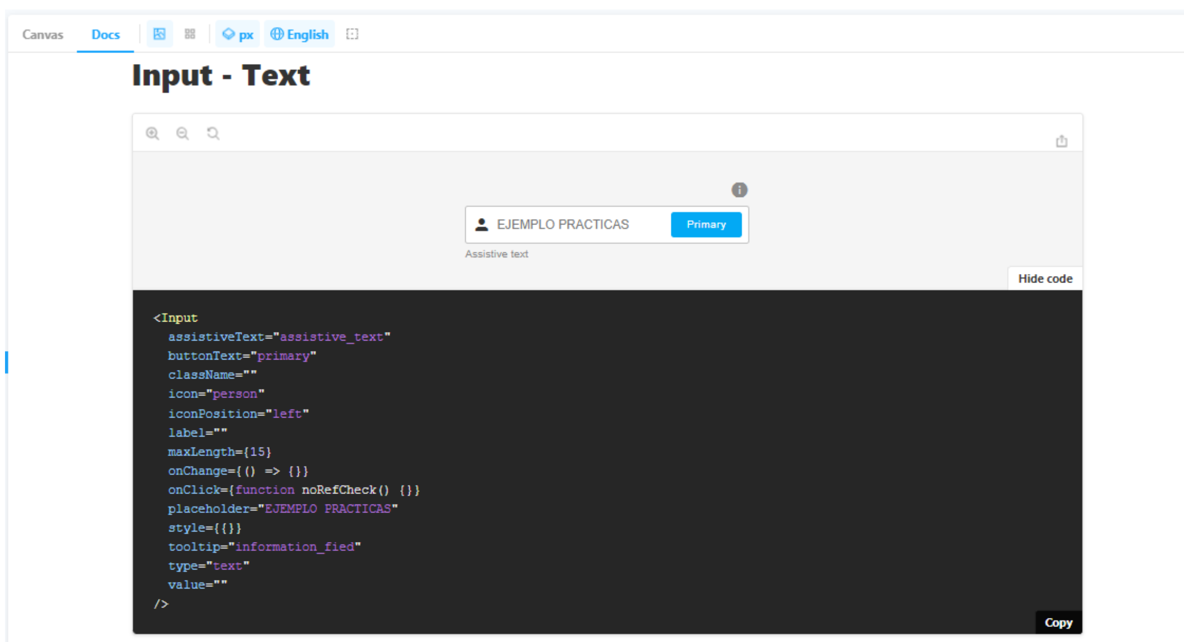


**Figura 49. Generación de código desde documentación de *Storybook***

### 6.1.3 Pruebas unitarias

En la figura 50 se aprecia el resultado de la ejecución de todas las pruebas unitarias de los componentes de la sección 6.1.2. En total se realizaron 68 pruebas unitarias para un promedio entre 7 y 8 pruebas por componente con el propósito de evaluar el correcto comportamiento y visualización de cada uno de los componentes.

```
> \react-controls
> jest

PASS  src/components/TextArea/TextArea.test.tsx (21.289 s)
PASS  src/components/Modal/Modal.test.tsx (27.119 s)
PASS  src/components/Checkbox/Check.test.tsx (27.147 s)
PASS  src/components/Tooltip/Tooltip.test.tsx (27.191 s)
PASS  src/components/SidePanel/SidePanel.test.tsx (27.466 s)
PASS  src/components/Input/Input.test.tsx (27.71 s)
PASS  src/components/Button/Button.test.tsx (27.828 s)
PASS  src/components/Switch/Switch.test.tsx (6.619 s)
PASS  src/components/Radio/Radio.test.tsx

Test Suites: 9 passed, 9 total
Tests:       68 passed, 68 total
Snapshots:   0 total
Time:        31.894 s, estimated 63 s
Ran all test suites.
```

**Figura 50. Resultado de ejecución de pruebas unitarias de los componentes de sección 6.1.2**

### 6.1.4 Librería *NPM*

En la figura 51 se presenta una captura de pantalla de la librería privada de *NPM* alojada en la página npmjs.com. Ingresando a esta página con credenciales de la empresa se puede notar que la librería se encuentra en la versión 0.0.4, que proporciona el comando necesario para instalar la librería en un proyecto de *React* y contiene un apartado para información adicional como instrucciones de instalación.

npm
Search packages
Search

0.0.4 • Private • Published 2 months ago

Readme | Code Beta | 6 Dependencies | 0 Dependents | 4 Versions | Settings

Installation:

This is a Node.js module available through the npm registry.

Before installing, download and install Node.js. Node.js 14.17.3 (LTS) or higher is recommended.

If this is a brand new project, make sure to create a package.json first with the npm init command.

Installation is done using the npm install command:

```
npm install
```

Install
> npm i

Repository
github.com/

Homepage
github.com/

Version
0.0.4

License
ISC

**Figura 51. Librería *NPM* de los componentes visuales alojada en npmjs.com**

## 6.2. Análisis

### Flujo actual de trabajo con componentes visuales

Actualmente en la empresa cuando un componente visual requiere ser modificado, se le asigna la tarea a un desarrollador del equipo y este se encarga de modificar el código existente en el repositorio del proyecto, estas modificaciones pasan por el equipo de QA de la empresa que se encarga de garantizar la calidad de los productos o servicios que ofrece una empresa y de revisar que los cambios cumplan con las funcionalidades especificadas, y posteriormente otro desarrollador diferente al que realizó las modificaciones del componente puede ser el encargado de implementarlo en la página web o plataforma web de la empresa.

Este flujo de trabajo cuenta con la desventaja de que los cambios de los componentes visuales pueden pasar por diversos desarrolladores y llegar a no ser consistentes en el tiempo, pueden llegar a ser difíciles de mantener y crean problemas de colaboración ya

que la información de los cambios puede quedarse en unas cuantas personas sin ser socializados con el resto del equipo.

Debido a la creación de una librería *NPM* de los componentes visuales y la centralización de estos, se genera un cambio en el flujo de trabajo previamente mencionado.

Cuando los proyectos dependan de esta librería el flujo de trabajo sería el siguiente:

1. El equipo de diseño pasa la solicitud al equipo de desarrollo del cambio que requiere en un componente o de la creación de un componente nuevo.
2. El caso le es asignado a un desarrollador con conocimientos del *Storybook* en cuanto al manejo del lenguaje TypeScript, como documentar el componente y creación de pruebas unitarias en Jest.
3. Cuando el componente es creado o modificado, pasa por el equipo de QA para validar sus funcionalidades y estilos.
4. Se publica una nueva versión de la librería en caso de que sea urgente y en caso de que no lo sea, se espera a que haya varios componentes a cambios significativos para publicar una nueva versión de Liberia *NPM*.
5. Una nueva tarea es creada para la implementación del nuevo componente usando la nueva versión de la librería.
6. En caso de que los cambios de la versión solo sean estéticos, solo es necesario la actualización a la nueva versión para ver cambios en los componentes inmediatamente en el proyecto que usa la librería.

## 7. Conclusiones

- Se cumplió con el objetivo de migrar los componentes propuestos inicialmente del lenguaje JavaScript al lenguaje TypeScript y en los casos necesarios los

componentes de tipo clase a tipo función permitiendo así la implementación de la herramienta llamada *Storybook* para la creación de un catálogo de componentes visuales con los que cuenta la empresa.

- Se puede evidenciar la ventaja de utilizar TypeScript en lugar de JavaScript para la generación de la librería *NPM* ya que al momento de utilizar la librería en un proyecto de la empresa se cuenta con la opción de identificar y prevenir errores en tiempo de ejecución ya que estos pueden ser encontrados en tiempo de compilación debido a la definición de un tipo de variable para cada *prop* de los componentes.

- Una de las ventajas de centralizar los componentes en una librería de *NPM* es que se promueve la reutilización de código y la consistencia en la interfaz de usuario en todo el proyecto. Si todos los componentes visuales están en un lugar centralizado, es más fácil para los desarrolladores encontrar y reutilizar los componentes existentes en lugar de crear nuevos desde cero.

- Por medio de la herramienta *Storybook* se logra crear un catálogo interactivo de los componentes visuales con los que cuenta la empresa ayudando a la centralización de la información tanto del código como de la documentación. Por lo que también se logra tener un punto de referencia entre el equipo de diseño y de desarrollo permitiendo así evitar ambigüedades en los diseños en las implementaciones requeridas en las plataformas web.

- Se logró la creación de pruebas unitarias que se encargan de la validación de las funcionalidades y estilos de los componentes migrados, permitiendo verificar rápidamente que la funcionalidad existente no es afectada cuando se modifica el código, o se crean nuevas funcionalidades.

## 8. Referencias bibliográficas


1. TSO Mobile Colombia. (2021). Empresa de Rastreo Satelital - GPS. Disponible en: https://tsomobile.com.co/empresa-de-rastreo-satelital/
2. Atlassian. (2020). BitBucket. Disponible en: https://bitbucket.org/
3. Cloudflare. (s.f) What Do Client-Side and Server-Side Mean? Disponible en: https://www.cloudflare.com/learning/serverless/glossary/client-side-vs-server-side
4. Clark, J. (2016). What is the Internet of Things, and how does it work? IBM. Disponible en: https://www.ibm.com/blogs/internet-of-things/what-is-the-iot/
5. Amazon AWS. (s.f.). What is AWS. Disponible en: https://aws.amazon.com/what-is-aws/
6. Microsoft Azure. (n.d.). What is Azure—Microsoft Cloud Services. Disponible en: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/
7. Microsoft Azure. (s.f.). What is the Cloud - Definition. Disponible en: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-the-cloud/
8. MDN Web Docs. (2023). HTML: Lenguaje de etiquetas de hipertexto. Disponible en: https://developer.mozilla.org/es/docs/Web/HTML
9. MDN Web Docs. (2023). CSS. Disponible en: https://developer.mozilla.org/es/docs/Web/CSS
10. Amazon AWS. (s.f.). ¿Qué es JavaScript? - Explicación de JavaScript (JS). Disponible en: https://aws.amazon.com/es/what-is/javascript/
11. Microsoft. (s.f.). ¿Qué es TypeScript? - Información general de TypeScript. Disponible en: https://learn.microsoft.com/es-es/training/modules/typescript-get-started/2-typescript-overview
12. NeoAttack. (2020). ¿Qué es un Framework y para qué sirve?. Disponible en: https://neoattack.com/neowiki/framework/

13. React. (s.f.). Una biblioteca de JavaScript para construir interfaces de usuario. Disponible en: https://es.reactjs.org/
14. Meta. (2021). The Facebook Company Is Now Meta. https://about.fb.com/news/2021/10/facebook-company-is-now-meta/
15. Meta. (s.f.) React | Meta Open Source. Disponible en: https://opensource.fb.com/projects/react/
16. React. (s.f.). Componentes y propiedades - React Disponible en: https://es.reactjs.org/docs/components-and-props.html
17. React. (s.f.). Virtual DOM and Internals. Disponible en: https://reactjs.org/docs/faq-internals.html
18. Ugwu, R. (2022). What is the virtual DOM in React? LogRocket Blog. Disponible en: https://blog.logrocket.com/virtual-dom-react/
19. Ahmed, A. (2021). Differences Between React Class Components Vs. Functional Components. Medium. Disponible en: https://medium.com/swlh/differences-between-react-class-components-vs-functional-components-2468e90f3857
20. React. (s.f.). El ciclo de vida del componente. Disponible en: https://es.reactjs.org/docs/react-component.html#the-component-lifecycle
21. React. (n.d.). Built-in React Hooks. Disponible en: https://react.dev/reference/react
22. Storybook. (s.f.). Storybook: Frontend workshop for UI development. Disponible en: https://storybook.js.org/
23. Storybook. (s.f.). Why Storybook? Disponible en: https://storybook.js.org/docs/react/get-started/why-storybook
24. IONOS (2019). El papel del unit test en el desarrollo de software. Disponible en: https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/el-papel-del-unit-test-en-el-desarrollo-de-software/
25. Ugwu, R. (2022). Jest testing: Top features and how to use them. LogRocket Blog. Disponible en: https://blog.logrocket.com/jest-testing-top-features/
26. Granados, A. (2020). ¿Cómo empezar a hacer Unit Testing con Jest? Guía básica. Medium. Disponible en:

https://medium.com/@angelygranados/c%C3%B3mo-empezar-a-hacer-unit-testing-con-jest-gu%C3%ADa-b%C3%A1sica-ca6d9654672
27. NPM (s.f.). About npm - npm Docs. Disponible en: https://docs.npmjs.com/about-npm
28. Atlassian (s.f.). Qué es el control de versiones. Disponible en: https://www.atlassian.com/es/git/tutorials/what-is-version-control
29. Atlassian (s.f.). Flujo de trabajo de Gitflow. Disponible en: https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow
30. Node.js (s.f.). About Nodejs. Disponible en: https://nodejs.org/en/about

## 9. Anexos

Como se expresó en la sección 6.1.2 acerca de los resultados de Storybook, a continuación se presenta la documentación generada por esta herramienta para cada uno de los componentes mencionados en la sección 4.1.4. que fueron migrados de JavaScript a TypeScript y ahora se encuentran en la librería NPM liberada en este proyecto.

### 9.1. Checkbox

Este componente permite al usuario seleccionar una o varias opciones de una lista de opciones disponibles. En la documentación de este componente mostrada en la figura 52 se pueden observar algunas de las *props* cuyos tipos de datos se definieron y también se documentaron tales como *“label”*, *“checked”* o *“isDisabled”.*

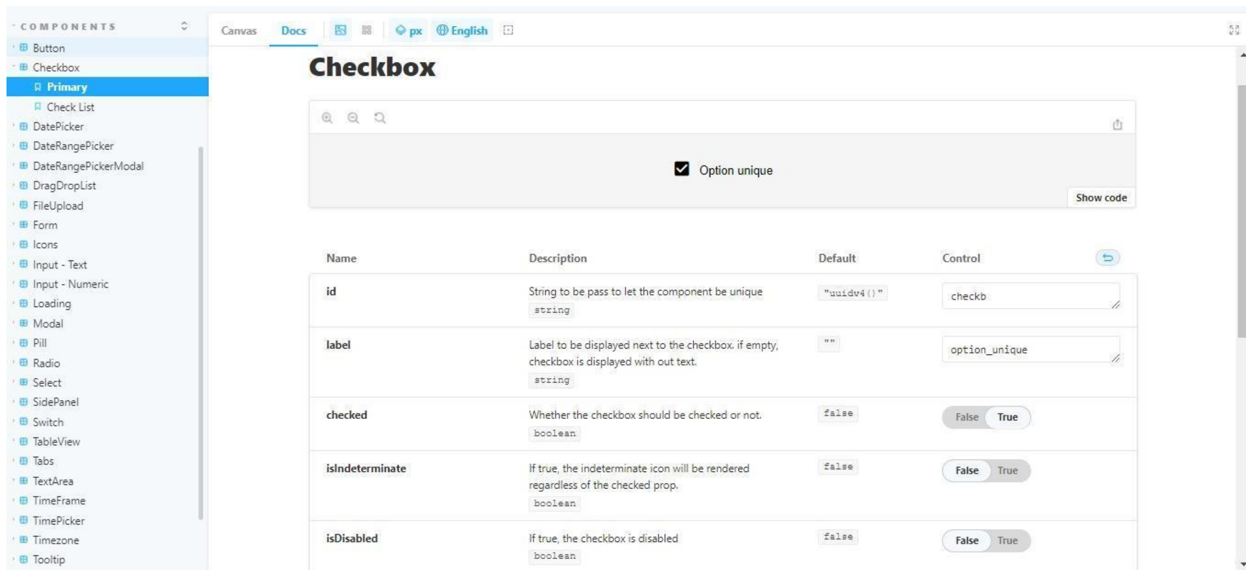



**Figura 52. Documentación de componente *Checkbox***

## 9.2. Radio *Button*

Este componente permite al usuario seleccionar una única opción de una lista de opciones disponibles. En la documentación de este componente mostrada en la figura 53 se pueden observar algunas de las *props* cuyos tipos de datos se definieron y también se documentaron tales como *"label"*, *"checked", "value"* o *"group".*

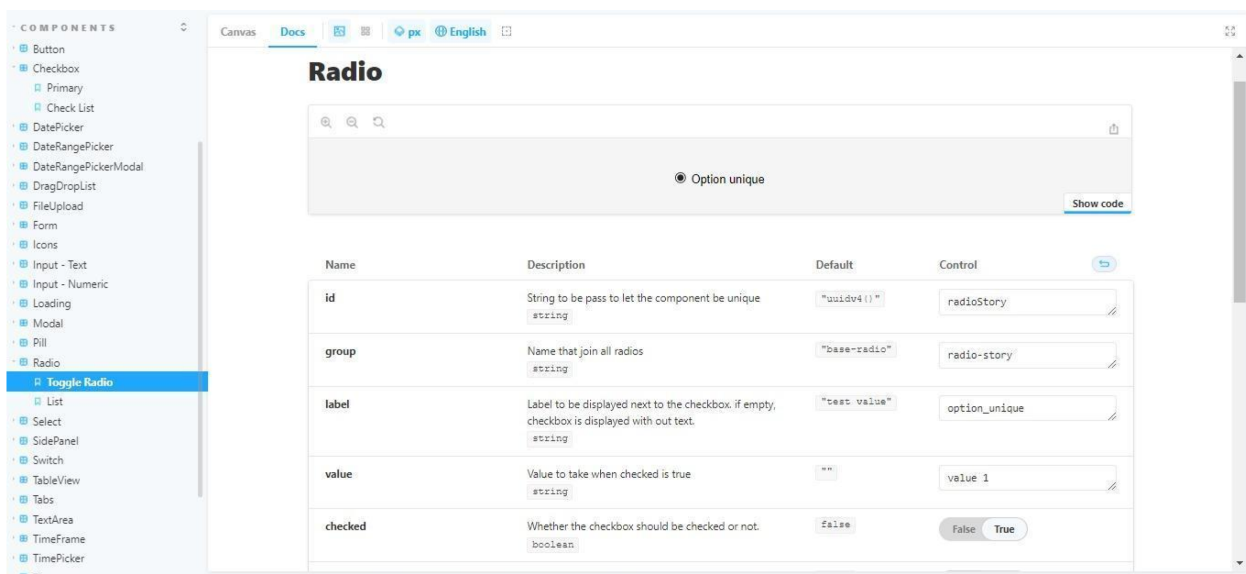


**Figura 53. Documentación de componente *Radio Button***

## 9.3. *TextArea*

Este componente permite al usuario introducir y editar grandes cantidades de texto en una página web o en una aplicación. En la documentación de este componente mostrada en la figura 54 se pueden observar algunas de las *props* cuyos tipos de datos se definieron y también se documentaron tales como *"name"*, *"label"*, *"style"* o *"placeholder"*.

**Figura 54. Documentación de componente *TextArea***

## 9.4. *SidePanel*

Este componente se utiliza para mostrar información adicional en un panel lateral de la pantalla y a menudo se utiliza en aplicaciones web para permitir al usuario interactuar con la aplicación de manera más eficiente. En la documentación de este componente mostrada en la figura 55 se pueden observar algunas de las *props* cuyos tipos de datos se definieron y también se documentaron tales como *"hiddenButton"*, *"style"* o *"classSidePanel"*.

Canvas Docs px English

# SidePanel

(Click the arrow on the top right.)

Show code

| Name | Description | Default | Control |
|---|---|---|---|
| hiddenButton | If true, the floating button to open the side panel is not rendered `boolean` | `false` | False True |
| style | Object with styles to be applied to the SidePanel `CSSProperties` | `{ }` | style : { width : "350px" transform : "translateX(350px)" } RAW |
| classSidePanel | Specify an optional className to be added to the SidePanel `string` | `""` | Set string |
| classPanelContent | Specify an optional className to be added to the SidePanel content | `"gtc-rc-side-panel-content"` | Set string |

**Figura 55. Documentación de componente *SidePanel***

## 9.5 *Switch*

Este componente de interfaz de usuario se utiliza para alternar entre dos estados o valores como "aceptar/denegar" o "activar/desactivar". En la documentación de este componente mostrada en la figura 56 se pueden observar algunas de las *props* cuyos tipos de datos se definieron y también se documentaron tales como *"label", "optionLabels"* o *"labelPosition".*

**Figura 56. Documentación de componente *Switch***