Desarrollo de una aplicación móvil híbrida para el despliegue de una galería fotográfica utilizando los frameworks open source Ionic y Capacitor

Ángel Salvador López-Vásquez[1]*, José Francisco Delgado-Orta[1], Jorge Ochoa-Somuano[1], Omar Antonio Cruz-Maldonado[1], Ángel Antonio Ayala-Zúñiga[1], María Alejandra Menéndez-Ortiz[1], Juan Mario Martínez-Ruíz[2], Imelda Perales-Ambrosio[2], Karla Cristina Pacheco-De la Paz[2] & Marco Polo Reyes-Barrera[2]

Resumen

El presente trabajo aborda la construcción de una aplicación móvil de tipo galería de fotografías a partir del uso de los marcos de trabajo (frameworks) de tipo open source Ionic y Capacitor. *De la misma forma, se realiza un análisis de los trabajos relacionados que establecen las ventajas de estas tecnologías para su uso dentro del desarrollo de software para dispositivos móviles, haciéndo énfasis en las aplicaciones híbridas en cuanto a menores tiempos de programación con respecto a las aplicaciones nativas y web existentes para los dispositivos móviles. Por consiguiente, la construcción de la aplicación híbrida se presenta en una forma estructurada, partiendo desde los entornos de desarrollo web, y en donde los frameworks facilitan la reutilización del código fuente, permitendo también la programación del hardware del dispositivo y la generación de una aplicación nativa, la cual se puede instalar en los sistemas operativos Android e iOS, siendo los sistemas para dispositivos móviles con la mayor cantidad de usuarios a nivel mundial.*

Palabras clave: aplicación móvil, aplicación híbrida, framework, framework Ionic, framework Capacitor.

Recibido: 19 de enero de 2022.

Abstract

This work approaches the development of a mobile application as a photograph gallery. *The application is built through the open source frameworks Ionic and Capacitor to generate a hybrid application, a known type in mobile app development that presents low programming costs and optimized response times with regard to the web and native development approaches. Therefore, the frameworks allow the creation of the hybrid app in a structured way from the web to native development environments, reusing the web source code and permitting the construction of a native app that can be launched in mobile operating systems like Android and iOS, which currently have the largest number of users in the world.*

Key words: mobile application, hybrid application, framework, Ionic framework, Capacitor framework.

Aceptado: 21 de abril de 2022.

[1] *Instituto de Industrias, Universidad del Mar campus Puerto Escondido. Ciudad Universitaria, Vía Sola de Vega km 1.5 Carretera Puerto Escondido- Oaxaca. San Pedro Mixtepec-Juquila, Oaxaca, México, 71980.*

[2] *Licenciatura en Informática, Universidad del Mar campus Puerto Escondido. Ciudad Universitaria, Vía Sola de Vega km 1.5 Carretera Puerto Escondido- Oaxaca. San Pedro Mixtepec-Juquila, Oaxaca, México, 71980.*

* **Autor de correspondencia:** *angel.vasquez@zicatela.umar.mx (ASLV)*

Introducción

La tecnología móvil ha venido evolucionando en los últimos años, en sus inicios un dispositivo móvil se utilizaba para llevar a cabo labores esenciales, tales como realizar llamadas de voz y enviar mensajes de texto (como es el caso del teléfono móvil), mientras que en otros casos se le daba uso como un asistente electrónico (también llamado PDA o *Personal Digital Assistant*), en el cual una persona era capaz de procesar y utilizar aplicaciones sencillas como calendarios, listas de contactos, bloc de notas, aplicaciones de dibujo, entre otras. Hoy en día estos dispositivos han alcanzado un grado de desarrollo tal, que se utilizan para realizar una gran cantidad de tareas cotidianas, las cuales han extendido los usos iniciales, llegando a un punto que en la actualidad la tecnología se ha convertido en un elemento que acompaña a los usuarios en todo momento.

Actualmente, las aplicaciones móviles son utilizadas por la población para tareas cotidianas en diferentes entornos, por ejemplo dentro de los entornos productivos a partir de la inclusión de herramientas de oficina (u ofimáticas); en la preservación de la salud y el establecimiento de estilos de vida saludable con aplicaciones que llevan el control de peso, dieta y ejercitación corporal; en el ámbito del entretenimiento, como elementos de transmisión de contenidos digitales o *streaming*; en los rubros económicos y de comercio electrónico, a través de la realización de operaciones aplicaciones bancarias y transacciones entre negocios que funcionan en Internet; como elemento de ubicación de recursos a partir del uso de los dispositivos de geolocalización que acompañan a los dispositivos, cuyas aplicaciones de soporte permiten a los usuarios llegar a sus destinos en las rutas y tiempos más cortos; para ocio y difusión de la cultura, entre muchas otras actividades. Es importante resaltar que muchas de las personas que cuentan con un dispositivo móvil realizan algunas de éstas actividades a través de aplicaciones de software.

Y es precisamente debido a los vastos campos de aplicación del desarrollo de software para dispositivos móviles, que el número de profesionales en el área de tecnologías de la información se ha incrementado significativamente en los últimos diez años. Lo anterior obedece a que el uso de los dispositivos móviles ha desplazado hoy en día el uso de la computadora personal como dispositivo electrónico para este tipo de actividades (Fig. 1). Lo anterior se debe a la cantidad de personas dedicadas a los giros de desarrollo de software y programación, pues es un área emergente en donde se tienen mejores oportunidades de trabajo y en donde los dispositivos se renuevan periódicamente, abriendo paso a las nuevas tecnologías.

La figura 1 muestra que, a principios de la segunda década del siglo XXI, las computadoras personales tenían un porcentaje superior al 98% de utilización con respecto a los dispositivos móviles, tendencia que comenzó a cambiar a partir de la segunda mitad del año 2016, en el que la cantidad de usuarios de dispositivos móviles superó por primera vez a los usuarios de computadoras personales, este comportamiento se sigue conservando y va en aumento. En diciembre del año 2020, el uso de dispositivos móviles (entre los que se contabilizan los teléfonos móviles y las tabletas) ha alcanzado, en conjunto, el 59% de los electrónicos, mientras que el uso de las computadoras personales se ha reducido al 41%, lo que muestra la preferencia de los usuarios por los dispositivos móviles en los años más recientes, tendencia que se espera se incremente en los próximos años.

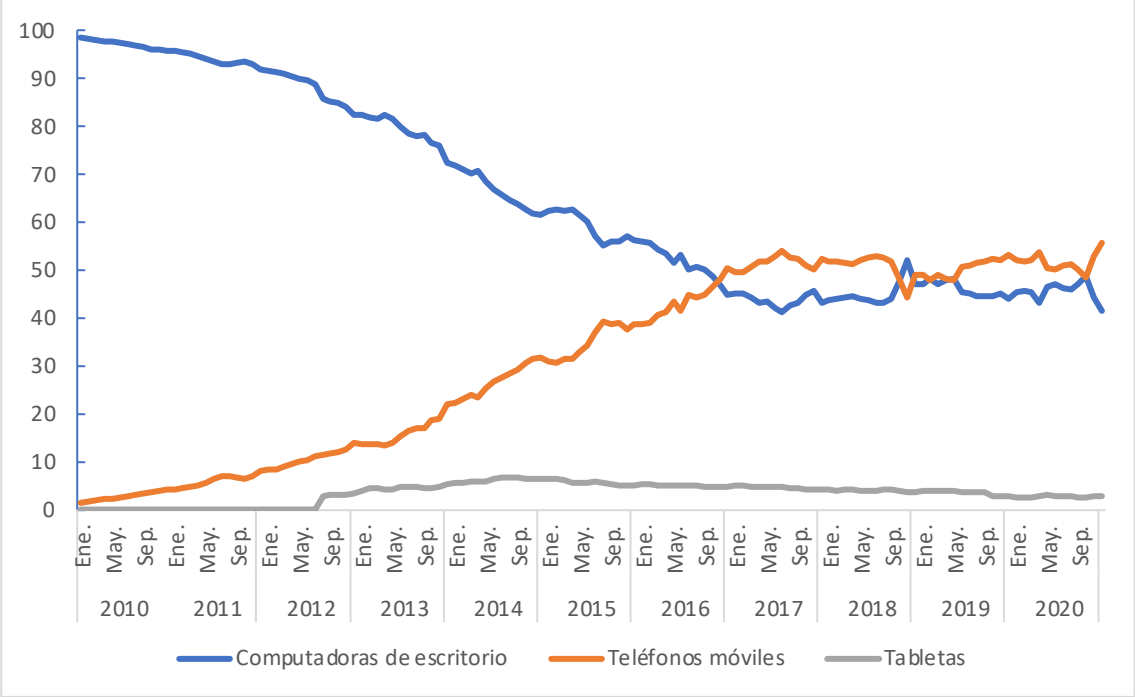

Figura 1. Relación de uso de los dispositivos móviles y las computadoras personales (2010-2020) (Fuente: StatCounter Global Stats).

En el mismo sentido, el desarrollo de software para los dispositivos móviles ha crecido de forma más acelerada que el software de las computadoras personales, lo cual ha sido posible en gran medida a las prestaciones del *hardware* de los dispositivos móviles, como las cámaras digitales, acelerómetros, GPS, sensores y otros, mismos que permiten su utilización de forma conjunta a través de las aplicaciones móviles, y que debido a la relación costo-beneficio de los dispositivos comúnmente no están disponibles en una computadora personal. Tanto el hardware como el *software* son gestionados a través de la microcomputadora del dispositivo y esta característica le otorga la denominación de "inteligente", siendo el caso más común el de los teléfonos inteligentes o smartphones. Este desarrollo fue posible, en gran medida, gracias al surgimiento de las tecnologías multimedia adquiridas desde la tercera generación (conocida como 3G), dentro del entorno de las telecomunicaciones, ocurriendo esto a finales de la primera década del siglo XXI. Los cambios más significativos de esta generación, se dieron con la introducción de los protocolos de transmisión de voz y datos en las redes de telefonía móvil, en la 3G se introdujo el uso del protocolo de Internet (IP) en la infraestructura de las redes , mismo que se ha estado mejorando para brindar soporte en el envío y recepción de grandes volúmenes de señales de voz y datos a través de las redes de telecomunicaciones. Estas tecnologías son el soporte para las aplicaciones móviles que se ejecutan en los *smartphones* y su evolución continúa hoy en día en las redes actuales 4G y 5G. De esta forma, los avances tecnológicos de las redes y las telecomunicaciones, así como el desarrollo de los sistemas operativos para los dispositivos móviles, propiciaron que el desarrollo de estas aplicaciones fuera un mercado abierto para todos los usuarios, quitando las barreras impuestas por los fabricantes en los inicios de la segunda década del siglo XXI.

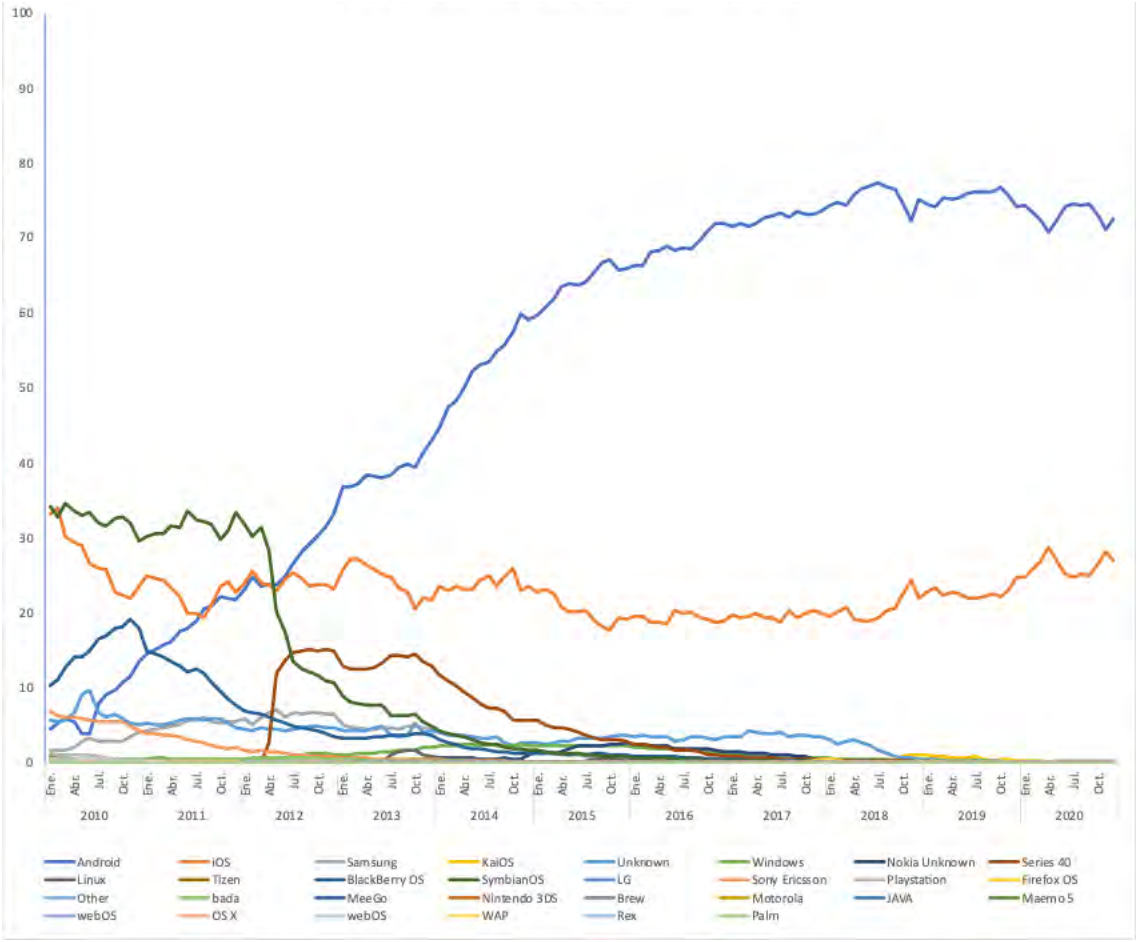

Figura 2. Usuarios de los sistemas operativos en los dispositivos móviles (2010-2020) (/Fuente: StatCounter Global Stats (https://gs.statcounter.com/).

A causa de esta situación, en los últimos diez años el desarrollo de los sistemas operativos móviles asociaba a cada sistema con un fabricante, quien dotaba de las interfaces de usuario a sus propios dispositivos, lo que ocasionó que existieran en el desarrollo histórico alrededor de 30 sistemas operativos distintos. La Figura 2 muestra la utilización mundial de los sistemas operativos en el periodo 2010-2020, de acuerdo con la información del sitio *StatCounter Global Stats* (StatCounter 2021).

La información de la figura 2 muestra que, a principios del año 2010, los sistemas que destacaron por la mayor cantidad de usuarios eran *Symbian OS* de la marca *Nokia*, *iOS* de *Apple* y *Blackberry OS* de su misma propiedad. Sin embargo, la robustez de estos y otros de sistemas operativos, así como la rápida evolución de las tecnologías del hardware y el software para los dispositivos móviles, ocasionó que muchos de estos fueran quedándose obsoletos, ya que el proceso de producción de nuevas versiones de los sistemas resultaba demasiado costoso para los fabricantes, reflejándose este costo en el precio que el consumidor paga por el dispositivo .

Para contrarrestar esta situación, muchos de los fabricantes adoptaron los sistemas operativos emergentes como

Android, debido a la característica open source, que permite adaptar el sistema para un dispositivo particular a partir de la distribución del código fuente, siendo la razón por la que hoy en día cuenta con la mayor cantidad de usuarios en el mundo, existiendo en alrededor del 72% de los dispositivos móviles, mientras que el sistema iOS de Apple se ha mantenido en el gusto del 27% de los usuarios, delegando ambos sistemas un porcentaje menor al 1% al resto de los usuarios de sistemas operativos móviles, hasta el mes de diciembre de 2020.

Hoy en día es común escuchar el término "aplicación móvil" o *app*, como un término asociado a una aplicación de software que se crea para desarrollar una tarea específica en un dispositivo móvil, en donde los recursos tecnológicos disponibles se utilizan para establecer tres tipos elementales de aplicaciones: aplicaciones web móviles, aplicaciones móviles nativas y aplicaciones híbridas, cada una de ellas se clasifica de acuerdo con las prestaciones disponibles del hardware , así como a las propias características asociadas al desarrollo de software. En la actualidad se distribuyen aplicaciones que aprovechan todos los recursos del hardware y el *software* de los teléfonos inteligentes de las múltiples gamas existentes.

Desarrollo

Aplicaciones web móviles

Las aplicaciones web móviles son diseñadas para ejecutarse dentro de un navegador, se desarrollan con tecnología *web* estándar (HTML, CSS y *JavaScript*), y cuentan con una serie de características favorables, entre las que se destacan que no necesitan adecuarse a ningún sistema operativo, son independientes de la plataforma y su puesta en marcha es rápida y sencilla. Sin embargo, sus tiempos de respuesta decaen debido a la interacción cliente-servidor natural que se da por el uso del protocolo HTTP (*Hypertext Transfer Protocol*), y al mismo tiempo resultan ser menos atractivas en cuanto a su diseño que las aplicaciones nativas. Además, las restricciones de seguridad impuestas a la ejecución de código por medio de un navegador limitan el acceso a todas las capacidades del dispositivo (Tracy 2012).

Morillo (2014) define una aplicación web móvil como un sitio web específicamente optimizado para desplegarse y ajustarse a las dimensiones de un dispositivo móvil. Las características que definen una aplicación web son las siguientes: la interfaz de usuario que se construye con tecnologías web estándar, está disponible en un URL (*Uniform Resource Locator*) público, privado o protegido por una contraseña, y está optimizada para los dispositivos móviles. La principal característica de una aplicación web móvil es que no está instalada en el dispositivo, sino que se ejecuta sobre el navegador web del dispositivo, siendo este un requerimiento tecnológico necesario para que los usuarios puedan ejecutarlas, ya sea en la computadora personal o en el dispositivo móvil. Algunas de las ventajas que presentan este tipo de aplicaciones son:

- Que los desarrolladores web pueden usar sus propias herramientas.
- Que se pueden aplicar los conocimientos de diseño y desarrollo web elementales.
- Que la aplicación será capaz de funcionar en cualquier dispositivo que tenga un navegador web.
- Que se pueden solucionar los errores de programación y diseño en tiempo real.
- Que el ciclo de desarrollo del producto de software es más rápido.

Entre las principales desventajas del desarrollo de aplicaciones web móviles se tienen:

• La falta de acceso a todas las características del dispositivo móvil.

• La dificultad para conseguir efectos sofisticados en la interfaz de usuario.

• La imposibilidad de ejecutar en segundo plano ni como aplicaciones *offline*, ya que requieren conexión a Internet para poder obtener el contenido desplegable desde un servidor de aplicaciones para los dispositivos.

De esta forma, las aplicaciones web representan una alternativa de desarrollo de software económica pero limitada, ya que estas aplicaciones no son capaces de administrar el *hardware* del la computadora personal o el dispositivo móvil, razón por la cual se crearon para este último tipo de dispositivo las aplicaciones nativas que proporcionan el soporte tecnológico necesario para este fin.

Aplicaciones nativas

El desarrollo móvil nativo es el desarrollo de las aplicaciones que son instaladas en el sistema operativo de cada dispositivo móvil. Estas aplicaciones poseen la ventaja de que cuentan con los mercados de distribución de los sistemas operativos móviles, como por ejemplo, en la *AppStore* (*iOS*) o en la *Play Store* (*Android*). De acuerdo con (Morillo 2014), al estar instaladas sobre el sistema operativo móvil, tienen acceso completo al hardware (altavoces, acelerómetro, cámaras, etc.), siendo ésta su principal ventaja de utilización, y están escritas en algún lenguaje de programación compilado (como, por ejemplo, el *Objective-C*, *Java*, *Windows Mobile*, etc.). Sin embargo, algunos de los inconvenientes que presenta el desarrollo de aplicaciones nativas son:

• Que la aplicación solo funcionará en la plataforma (sisema operativo) elegido.

• Que se requiere conocer detalladamente el lenguaje de programación que utiliza cada plataforma.

• La gestión de errores de programación y diseño es compleja, ya que para ello se requiere lanzar actualizaciones o parches desde las aplicaciones de distribución.

• Al ser un producto de *software* basado en aplicaciones de escritorio, su ciclo de desarrollo es más lento que el de las aplicaciones web.

Observando las ventajas y desventajas de las aplicaciones *web* y nativas, la industria del de software dio lugar al surgimiento de las aplicaciones híbridas, las cuales aprovechan las ventajas de ambos tipos de aplicaciones para mejorar el proceso de desarrollo, cuyo enfoque se basa en la reutilización de componentes de *software* .

Aplicaciones híbridas

Las aplicaciones híbridas, de acuerdo con (Peña 2017), consisten en código nativo que se crea a partir de un navegador web incorporado que se utiliza para procesar partes de la interfaz de usuario de la aplicación y que están escritas en formato estándar usando HTML, *JavaScript* y CSS. Partiendo de estas tecnologías, se tiene acceso a una buena parte de los dispositivos del hardware y a los sensores del teléfono a través de *frameworks* de desarrollo que se utilizan como una capa intermedia de la programación del dispositivo. Un ejemplo clásico de una aplicación híbrida es el caso de *Facebook*, la cual es la aplicación de mayor utilización en el mundo y que fue desarrollada bajo el principio descrito para contar con su disponibilidad en

Tabla I. Análisis de las características de las aplicaciones de los dispositivos móviles.

Característica	Aplicación web móvil	Aplicación nativa	Aplicación híbrida
Plataforma de desarrollo	Navegadores móviles	iPhone OS (iOS), Windows Mobile, Blackberry OS, Symbian, Android.	iPhone OS (iOS), Windows Mobile, Blackberry OS, Symbian, Android
Distribución	URL y códigos QR	Tiendas de aplicaciones según la plataforma.	Tiendas de aplicaciones según la plataforma.
Instalación	Se accede directamente y queda disponible mediante un launcher en el dispositivo.	Se realiza una vez y queda disponible.	Se realiza una vez y queda disponible para todas las plataformas.
Costos de desarrollo	Menores.	Mayores.	Menores que los de las nativas.
Rendimiento	HTML 5 mejora la infraestructura de red.	Más rápido, especialmente si requiere procesos gráficos pesados.	Se desarrolla como nativa cuando el rendimiento sea esencial.
Integración de hardware	Limitada.	Completa.	Buena.
Acceso fuera de línea	Solo en algunos dispositivos mediante HTML5.	Completo.	Completo.
Usabilidad	Buena.	Gran cantidad de efectos amigables en la interfaz, atractivos para el usuario.	Utiliza lo mejor de lo nativo y lo mejor de la red.

cualquier dispositivo móvil.

El objetivo que se busca con el desarrollo de las aplicaciones híbridas es el de poder llegar no solo a una plataforma, sino que la aplicación pueda extenderse a varias sin necesidad de lanzar periódicamente nuevas versiones. En ese sentido, en el trabajo de López Vásquez *et al*. (2022) se establece un análisis comparativo inicial de las características de los tres tipos de aplicaciones móviles con base en factores como la conectividad de red, la oferta de las aplicaciones en sus mercados de distribución, el acceso a las APIs (*Application Programming Interface* o interfaz de programación de aplicaciones) del sistema operativo, el uso de los dispositivos de captura de imagen y video de los dispositivos, el uso de elementos de geolocalización, el acceso a contactos y el sistema operativo, la comunicación entre aplicaciones, el almacenamiento local y el uso de notificaciones, observando que la aplicación híbrida compite en todos los rubros con la aplicación nativa, excepto con la capacidad de utilizar las APIs del sistema operativo. Lo anterior marca una pauta para que los programadores opten por una aplicación híbrida en el desarrollo móvil, por una parte, observando las ventajas del uso de los *frameworks* de tipo open source para que una aplicación híbrida cuente con las ventajas de las aplicaciones nativas que sean capaces de gestionar el *hardware* del dispositivo; mientras que por otra parte hace énfasis en el uso de la tecnología *web*, así como, los protocolos de redes para el envío y la

recepción adecuada de la información que se encuentra en cambio continuo a causa de la movilidad de los usuarios de los dispositivos. De esta forma, el análisis de la elección de un tipo de aplicación móvil se extiende a otras características necesarias que se suelen recomendar como parte del desarrollo de software profesional, por ejemplo, la tabla I muestra las características de los tres tipos de aplicaciones para los dispositivos móviles propuestas en (Angulo 2013), en cuanto a los elementos de plataforma, distribución, instalación, costos de desarrollo, rendimiento, integración de hardware, acceso fuera de línea y usabilidad.

Las características de plataforma y distribución presentadas en la tabla I, denotan que las aplicaciones nativas e híbridas cuentan con la ventaja de que pueden llevarse a las aplicaciones de distribución de los sistemas operativos móviles e instalarse de forma nativa en el dispositivo; a diferencia de una aplicación *web* móvil, a la que únicamente se tendrá el acceso si el servidor está disponible y la forma de acceder a ella es a través de un *launcher* (o acceso directo). En cuanto a costos de desarrollo, las aplicaciones híbridas representan un costo intermedio entre las aplicaciones web móviles y las nativas, esto debido a que la implementación de las vistas es sencilla, como consecuencia de utilizar HTML5, mientras que a este diseño hay que añadir la funcionalidad de los dispositivos, misma que se logra a través de *frameworks* de desarrollo. Por otro lado, en una aplicación web el soporte del hardware no está habilitado y en una aplicación nativa se requiere de conocimiento completo de las rutinas y del lenguaje de programación del entorno nativo para poder utilizar el hardware del dispositivo, lo cual se refleja en la integración con el hardware en donde una aplicación nativa brinda una funcionalidad completa, a diferencia de una aplicación híbrida que proporciona el acceso a muchos de los dispositivos.

De la misma forma, la característica correspondiente al funcionamiento de las aplicaciones sin conexión a Internet (acceso fuera de línea) muestra que para desarrollo de aplicaciones nativas e híbridas se agrega un soporte completo, mientras que una interfaz web se limita únicamente a los que ofrece HTML5. Por último, mientras que una aplicación *web* posee buena usabilidad, una aplicación nativa mejora las características de las interfaces web por la gran cantidad de efectos amigables que se pueden crear desde el lenguaje de programación. Del análisis anterior, la información de la tabla hace énfasis en que el desarrollo híbrido utiliza ambas características para la construcción de aplicaciones móviles. Obviamente, el análisis de las propias características y requerimientos de la aplicación son elementos esenciales para determinar cuál de los tres tipos de aplicaciones es el más viable de llevar a la fase de desarrollo. En ese sentido, Angulo (2013) presenta siete características (Tabla II) que establecen una directriz con base en los diferentes recursos de *hardware* y software necesarios para seleccionar un tipo de implementación, de acuerdo con requerimientos específicos de los usuarios.

De acuerdo con los criterios de selección del tipo de aplicación mostrados en la tabla II, el uso de aplicaciones *web* móviles se recomienda solamente cuando el presupuesto es reducido, ya que este tipo de desarrollos requieren solamente trabajar con los estándares de HTML, CSS y *JavaScript* para las aplicaciones *web*. Por otra parte, cuando se requiere del acceso al *hardware* del dispositivo las aplicaciones nativas e híbridas representan la mejor alternativa para desarrollar una

Tabla II. Criterios para seleccionar la mejor opción para el desarrollo de aplicaciones móviles (Angulo 2013).

Característica	Aplicación web móvil	Aplicación nativa	Aplicación híbrida
Acceso al hardware del dispositivo	Peor.	Mejor.	Mejor.
Funcionalidad sin conexión a internet.	Peor.	Mejor.	Intermedia.
Requerimiento de cálculos en tiempo real o gráficos 3D de alto rendimiento.	Peor.	Mejor.	Intermedia.
Presencia en sitios/ aplicaciones de distribución	Peor.	Mejor.	Mejor.
Cambios regulares en las reglas de negocio	Intermedia.	Peor.	Mejor.
Presupuesto reducido	Mejor.	Peor.	Intermedia.
Constante conexión con el servidor.	Intermedia.	Intermedia.	Mejor.

aplicación móvil, debido a que una aplicación *web* móvil presenta restricciones de seguridad establecidas por el protocolo HTTP para los navegadores *web*. De la misma forma, si se requiere que la aplicación funcione sin conexión a Internet para que ésta despliegue su contenido, la mejor alternativa es la aplicación nativa, ya que tanto las aplicaciones híbridas como las aplicaciones *web* móviles requieren que el dispositivo (cliente) se conecte a Internet para obtener el código desplegable desde el servidor de aplicaciones web.

Si se requiere que la aplicación procese grandes volúmenes de datos o realice una gran cantidad de cálculos, la aplicación nativa representa la mejor alternativa con respecto a las aplicaciones *web* móviles e híbridas, ya que utiliza la imagen del sistema operativo para procesar las instrucciones de entrada-salida requeridas, a diferencia de las aplicaciones *web* móviles e híbridas que requieren de los procesos de transporte de los datos, así como de la implementación del código seguro que permita el procesamiento de transacciones y consultas desde el dispositivo a través del transporte de la información y el uso de los protocolos de comunicaciones para que el código del cliente se envíe y se ejecute en el servidor y se procese para posteriormente retornarlo al dispositivo.

En cuanto a la distribución, las aplicaciones nativas e híbridas cuentan con una mayor cobertura debido a que este tipo de aplicaciones se pueden distribuir a través de aplicaciones como *AppStore*, *PlayStore*, etc., alcance que no tienen las aplicaciones web móviles, ya que su código se ejecuta y despliega sobre el navegador web, por lo que su distribución se da únicamente a través del URL (*Uniform Resource Locator*) que difunde el creador de la aplicación en su servidor *web*, lo que sujeta la

disponibilidad de la app a que el servidor web se encuentre en ejecución.

Cuando se analiza el presupuesto disponible para crear una aplicación móvil, una aplicación *web* móvil es menos costosa ya que solamente requiere conocimiento de los comandos HTML, contenido que se extiende en las aplicaciones híbridas con el uso de los *frameworks* para el despliegue del contenido en los dispositivos móviles a partir del diseño *web*. Por último, con respecto a la característica correspondiente a la conectividad con el servidor, la aplicación híbrida representa la mejor opción, ya que presenta la forma más eficiente de desplegar contenido cuando se requiere conectividad a Internet utilizando los recursos del servidor.

Del análisis anterior, se observan las ventajas del desarrollo híbrido sobre el resto de los tipos de aplicaciones, ya que para crear una *app* de tipo galería de fotografías se requiere solamente que la aplicación implemente las rutinas de lectura y escritura de los archivos digitales de las fotografías que se adquieren desde las cámaras del dispositivo móvil, por lo que se utilizará este enfoque, siendo ésta la alternativa que representa un esquema de trabajo más equilibrado, y en cuanto a conectividad y distribución de una app hacia una mayor cantidad de usuarios. Debido a que el desarrollo híbrido requiere del uso de capas de desarrollo intermedia, el proceso requiere de la integración de los frameworks necesarios para el manejo del hardware del dispositivo, donde el caso de tecnologías *open source* son viables para poder trabajar las versiones iniciales de la aplicación, para posteriormente distribuirla como una aplicación nativa que se ejecute en un sistema operativo móvil particular.

Los frameworks de desarrollo de aplicaciones móviles

La elección de una plataforma de desarrollo por parte del equipo del proyecto de *software* se enfoca, de acuerdo con (Puvvala *et al*. 2016) en cuatro factores: el primer factor asocia los costos de desarrollo asociados con las licencias del software e infraestructura; el segundo factor se enfoca en el retorno de inversión esperado por los programadores con el software y los tiempos del proceso de manufactura del *software*; el tercer factor asocia la facilidad en la programación ligada a los entornos de desarrollo, la disponibilidad de herramientas para construir prototipos, y las restricciones del código fuente; y el cuarto factor se asocia con el soporte brindado por la comunidad a partir de kits de desarrollo de *software* (*Software Development Kit* o SDK) disponibles para generar aplicaciones en contextos específicos.

En el caso de las aplicaciones web, la producción de aplicaciones móviles fue posible gracias a la evolución de los lenguajes de hipertexto como HTML (*HiperText Markup Language*), destacándose por la facilidad del manejo de contenido digital que es parte de muchas de las aplicaciones de software, permitiendo hoy en día el desplegado de cualquier contenido multimedia en entornos orientados a la *web*, mismos que se han replicado hacia los dispositivos móviles. Estos han sido acompañados de poderosos lenguajes gráficos para la gestión y organización de contenidos como CSS (*Cascade Style Sheet*), ambos tipos de recursos han sido de gran apoyo para que el contenido de sitios y aplicaciones *web* lleguen a los dispositivos móviles en forma de aplicaciones. En este campo, se pueden observar *frameworks* de diseño como Materialize, *Material Design* y *Bootstrap*, en donde un programador de páginas *web* es capaz de generar diseños

avanzados para desplegar el contenido de páginas web en dispositivos móviles, el cual se adapta a las dimensiones del dispositivo para desplegar los contenidos de la misma forma en que se muestran en una computadora personal. El contenido que se despliega a partir de estos *frameworks* se complementa desde el lado de la aplicación web con la interacción con servidores de aplicaciones como *Apache* y sus extensiones para propósitos específicos bajo el esquema *open source* o *Internet Integration Services* para tecnologías de Microsoft.

De la misma forma que el caso de la web, los desarrolladores han creado *frameworks* enfocados al desarrollo de aplicaciones híbridas que simplifiquen las desventajas que presenta el desarrollo nativo, destacándose entre los principales utilizados por los desarrolladores los siguientes de código abierto:

• *Flutter* (Flutter 2021), que se basa en el lenguaje de programación Dart. Su principal ventaja reside en que permite desarrollar aplicaciones móviles escribiendo solo una versión de la aplicación, siendo ejecutada en los sistemas *Android* e *iOS*. Las aplicaciones creadas en este marco se basan en *widgets* visuales, los cuales actúan como componentes independientes que conforman la interfaz de usuario de la aplicación.

• *Ionic*, un *framework* que habilita el Desarrollo de aplicaciones móviles híbridas usando tecnologías como HTML, CSS y *JavaScript*, las cuales se adaptan y optimizan para su operación en dispositivos móviles, lo que permite el desarrollo de aplicaciones móviles altamente interactivas. Las aplicaciones creadas en Ionic se ejecutan dentro de una vista *web* y pueden usar las funcionalidades, la cámara y los dispositivos de medición integrados en los dispositivos móviles modernos (Griffith 2017, Ravulavaru 2017).

• R*eact Native* es un *framework* de desarrollo de aplicaciones híbridas que se basa en *React*.js, se utiliza para el desarrollo de aplicaciones *front-end*, o interfaces gráficas de usuario (Masiello & Friedman 2017). Se diferencia de *Ionic* en el uso de componentes nativos y que éstos no se implementan en las vistas *web*. El código del programa de la aplicación se escribe en *JavaScript* y se mapea en componentes nativos de la interfaz de usuario, lo que mejora el desempeño y agrega un sentido de autenticidad en comparación con las aplicaciones basadas en la *web* (Eisenman 2017). Al igual que las aplicaciones creadas en *Ionic*, éstas tienen acceso a las funcionalidades de la cámara y los sensores del dispositivo (Masiello & Friedmann 2017).

• NativeScript es un *framework* para desarrollo de aplicaciones móviles basado en el lenguaje de programación *JavaScript*, en el cual se construye la lógica del programa de la aplicación. En este entorno de desarrollo, las vistas se definen en el lenguaje XML (*eXtensible Markup Language*) y se les puede dar formato con CSS (Branstein & Nick 2017). Al igual que las aplicaciones creadas en *Ionic* y *React Native*, las aplicaciones pueden tener acceso a las funcionalidades del sistema operativo, sensores y otros dispositivos de medición integrados en el dispositivo.

Estos cuatro *frameworks* fueron probados en (Denko *et al*. 2021), para mostrar su desempeño como parte de su uso en el proceso de desarrollo de una aplicación móvil, la cual es sometida a cuatro tareas de programación que se desarrollaron para aplicaciones que se ejecutaron en tres dispositivos móviles de tres diferentes fabricantes. La prueba de las *apps* se realiza en cuatro fases del proceso de desarrollo: la primera fase evalúa el tiempo en que se construye un paquete de instalación nativo para el sistema operativo *Android*,

así como el tamaño de los paquetes de instalación y las aplicaciones instaladas; la prueba concluye con el análisis del tiempo de instalación y el tiempo de carga de las aplicaciones. La segunda fase se enfocó en medir el uso de la CPU (Unidad Central de Procesamiento) y el consumo de la memoria RAM (Memoria de Acceso Aleatorio) en el caso del desplazamiento vertical en la pantalla al utilizar la aplicación. La tercera fase consistió en medir el tiempo de ejecución y el uso de la CPU de algoritmos clásicos de ordenamiento; mientras que la cuarta fase consistió en medir el uso de la CPU durante un monitoreo de recursos definido en un periodo de tiempo.

Los resultados obtenidos luego de llevar a cabo las cuatro fases se describen a continuación. En la primera fase de pruebas, *Ionic* obtuvo el tiempo más bajo en la creación del paquete de instalación, así como del menor tamaño y tiempo de instalación, incluso mejorando los tiempos de un entorno nativo. En lo referente al tiempo de carga de la aplicación, los entornos nativos son los más rápidos, mientras que, dentro de los *frameworks* de desarrollo híbrido, *React Native* obtiene los mejores resultados. En la segunda fase de pruebas, se observa un desempeño variante, en donde los f*rameworks Native Script* y *Flutter* tienden a ser los más eficientes cuando la aplicación se desplaza verticalmente a lo largo de la pantalla del dispositivo en los tres dispositivos usados para las pruebas. Por otra parte, en la tercera fase de pruebas, los frameworks mejoran los tiempos de respuesta de las aplicaciones nativas, destacándose *Native Script* en cuanto a tiempo de ejecución, mientras que Ionic se posicionó como el mejor con respecto al uso de la CPU, observando que el grado de optimización de su código fuente hace que la ejecución de aplicaciones desarrolle una baja exigencia a los recursos del procesador para poner en ejecución aplicaciones robustas como ordenamientos o una generación de números Fibonacci. Por último, la cuarta prueba mostró que *Flutter* es el más eficiente cuando se trata de compartir los recursos por intervalos de tiempo prolongados.

Si bien se tiene un comportamiento mixto en cuanto a la eficiencia de los frameworks bajo las condiciones descritas, se recomienda el uso de Ionic para aquellas aplicaciones que incorporen elementos de aprendizaje automático e inteligencia artificial, elementos que se pueden aplicar sobre las imágenes que se capturen desde las cámaras del dispositivo móvil (caso de aplicación del presente trabajo) y otros dispositivos de entrada, dado el grado avanzado de optimización que incorpora para la realización de tareas específicas, como las que se muestran en la tercera fase de pruebas presentadas por (Denko *et al*. 2021).

Implementación de la aplicación híbrida "Galería de fotos"

La implementación de la aplicación de "Galería de Fotos" consta de cinco etapas de desarrollo: en la primera etapa se establecen los requerimientos tecnológicos para el desarrollo de la aplicación; posteriormente en la segunda etapa se construye el proyecto de *software* asociado y las interfaces gráficas principales para su despliegue en el navegador *web*; la tercera etapa comprende la programación de la cámara y las rutinas de entrada-salida para la administración de los archivos de las fotografías desde el dispositivo hacia el sistema operativo móvil; la cuarta etapa comprende el desarrollo de las interfaces híbridas de soporte, las cuales permiten hacer la distribución de una aplicación móvil a los entornos nativos *iOS* y *Android*,

siendo los de mayor utilización a nivel mundial.

Toda aplicación móvil que se implementa en la modalidad híbrida requiere del soporte de los *frameworks* de despliegue para establecer los elementos del diseño de las interfaces gráficas de usuario (*Graphical User Interfaces* o GUI), siendo esta tarea la que se establece en la primera etapa de la construcción del proyecto.

Etapa 1. Implementación de las tecnologías de soporte

En esta etapa se establecen todos los elementos tecnológicos que darán el soporte a la aplicación, siendo el caso de los servidores web, así el medio que hace posible el procesamiento y despliegue de la aplicación en los dispositivos móviles, siendo utilizados para la aplicación *Node.js* e *Ionic*, siendo *Node.js* un entorno de programación que permite introducir comandos del lenguaje *JavaScript* en una aplicación móvil, mientras que Ionic permite establecer los complementos para visualizar los contenidos de la aplicación híbrida en simuladores que se pueden cargar en una computadora personal, con el objetivo de visualizar el diseño de las interfaces de usuario en la fase previa a su exportación al dispositivo móvil.

En el caso de *Ionic* (*Ionic* 2020), se selecciona como una herramienta de desarrollo debido a su característica *open source* (esto es, que no requiere licencia y se cuenta con el código del *framework* para adaptarlo y personalizarlo). Además, ofrece los complementos de desarrollo móvil que permiten implementar *scripts* de HTML5, CSS3 y componentes *JavaScript* para construir aplicaciones interactivas y con diseños agradables para los usuarios finales, así como de otras tecnologías de código abierto que permiten crear y mantener aplicaciones *web* como *Angular*, *React*, y *Vue*, todas estas herramientas ajustadas en un mismo paquete con lo que se puede lograr una buena arquitectura de desarrollo, diseño e integración con el dispositivo que puede ser *Android* o *iOS*.

Adicionalmente, la instalación del *framework Ionic* requiere de algunas herramientas que permitirán construir aplicaciones web, entre ellas, un editor de código como *Visual Studio Code*, una interfaz o terminal de línea de comandos (CLI), la creación de usuarios del sistema operativo *Windows*, *Mac OS* o *Linux*, siendo la interfaz recomendada la línea de comandos (cmd) o la CLI (*Command Line Interface*) de *Powershell*, que se ejecuta en modo administrador. Debido a la naturaleza multiplataforma que se busca con las aplicaciones híbridas, se requiere de un entorno de código abierto que permita trabajar con *Ionic*, siendo *Node.js* una alternativa que cumple con la integración de las aplicaciones en las diferentes etapas de desarrollo. *Node.js* es un framework que permite implementar operaciones de entrada/salida en sistemas informáticos utilizando lenguaje *JavaScript* que se ejecuta en el lado del servidor de una aplicación web. Para lograr este propósito, trabaja de manera asíncrona para que las respuestas del servidor desde una petición de un cliente web se ejecuten con mayor rapidez. El código *JavaScript* que corre en el backend es ejecutado gracias a la máquina virtual de *Google*, llamada V8, que es utilizada por el navegador *Google Chrome*. De esta forma, *Node.js* es un *framework* de código abierto que se compone de un entorno de ejecución y de una librería que se ejecuta en distintos sistemas operativos, la cual posee un gran rendimiento y permite la escalabilidad de las aplicaciones que se construyen con las tecnologías del lado del servidor, siendo esta

la razón de su selección como elemento para el desarrollo de la aplicación híbrida.

Instalación de Node.js

La instalación del entorno *Node.js* se realiza siguiendo nueve pasos, los cuales se describen a continuación. El primer paso consiste en ingresar al sitio *web* de *Node.js* (https://nodejs.org/es/), en el sitio *web* se mostrará una ventana del navegador *web* predeterminado como el de la figura 3. En la figura se muestran dos versiones del instalador: la 14.16.0 LTS y la 15.13.0 Actual, de éstas *Ionic* recomienda que cuando se instale *Node.js*, se seleccione la versión más reciente, *Long Term Support* (LTS), ya que es funcional en la mayoría de los dispositivos móviles.

El segundo paso consiste en seleccionar la versión del entorno de programación, y descargarla en el disco duro local. Una vez que se pulsa con el ratón esta opción 14.16.0 LTS., se inicia en breve la descarga del paquete de instalación (Figura 4).

Una vez que se ha descargado el paquete de instalación, el tercer paso consiste en abrir el archivo del paquete de instalación descargado en el paso 2. La ventana del programa de instalación mostrará el diálogo que se muestra en la figura 5, en donde la instalación continuará al presionar el botón de "*Next*".

El comando "*Next*" del tercer paso, dirige el instalador a la ventana de la Figura 6a), para continuar con la instalación hay que proceder con el cuarto paso que involucra la aceptación de los términos de la licencia del *software*, de manera que para habilitar el comando "*Next*" y continuar con la instalación, debe pulsarse la caja de verificación (*I accept the terms in the License Agreement*), tal y como se muestra en la figura 6b).

Una vez que se han aceptado los términos del uso de la licencia, se procede al quinto paso que es la selección del directorio de instalación dentro del sistema operativo en donde se realiza la instalación. En este caso, se observa en la figura 7, el ejemplo del directorio "C:\Program

Figura 3. Distribución del entorno *Node*.js.

Figura 4. Descarga del paquete de instalación de *Node*.js.

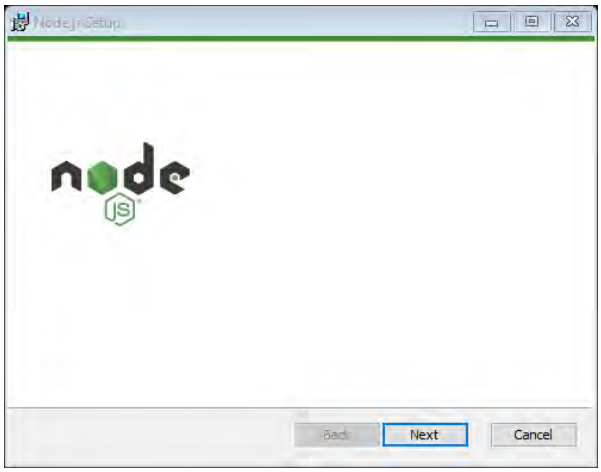

Figura 5. Diálogo inicial de instalación del entorno *Node*.js.

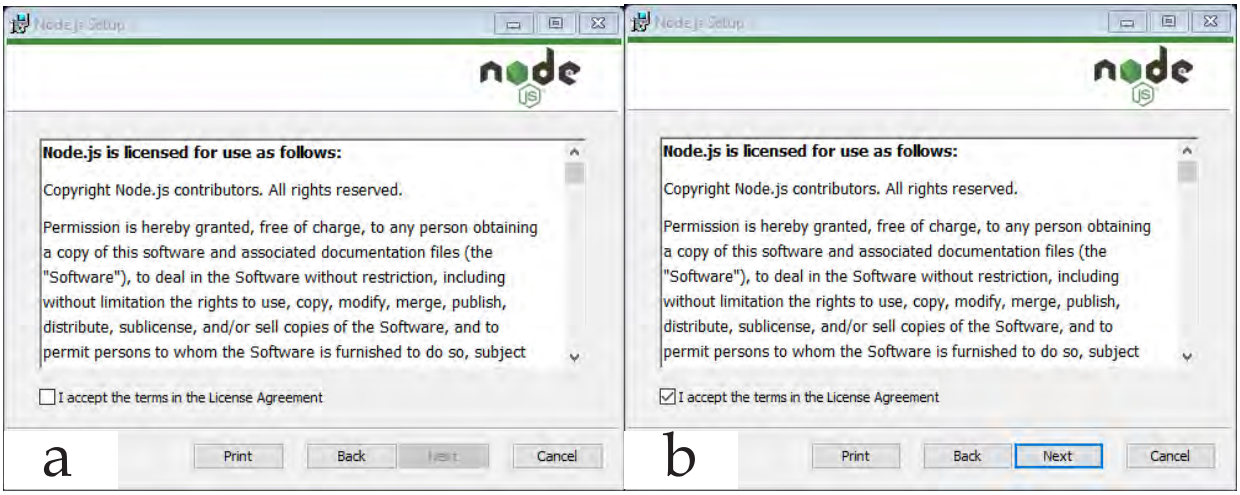

Figura 6. Aceptación de acuerdo de licencia para uso de *Node*.js.

Files\nodejs", opción que se establece por defecto, en caso de desear que la instalación se realice sobre otro directorio, se pulsa el botón "Change" para seleccionar otro directorio de trabajo correspondiente al sistema operativo Windows 10.

Una vez finalizado el quinto paso, se mostrará la ventana de la figura 8, en la cual se seleccionarán las opciones de instalación del entorno de programación, en donde se incluye un entorno de desarrollo en tiempo de ejecución (*Node*.js *runtime*), el sistema gestor de paquetes npm (npm *package manager*), accesos directos hacia la documentación disponible en Internet (*Online documentation shortcuts*), así como del establecimiento de variables de entorno para el sistema operativo contenedor del entorno (*Add to PATH*), variables que se utilizan para dirigir el flujo de instrucciones desde los accesos directos del sistema operativo hacia las aplicaciones para invocarlas desde eventos del ratón o teclado en los diferentes sistemas operativos. Para continuar con la instalación, se selecciona el botón "*Next*".

El séptimo paso consiste en la selección (opcional) de los módulos npm que requieren ser compilados desde compiladores de C/C++, los cuales son necesarios

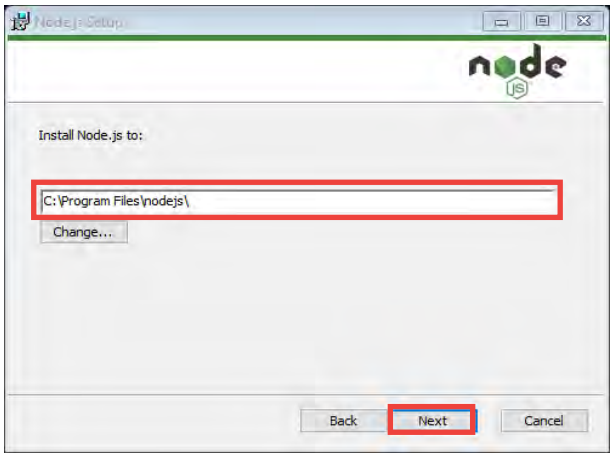

Figura 7. Selección del directorio de instalación de *Node*.js.

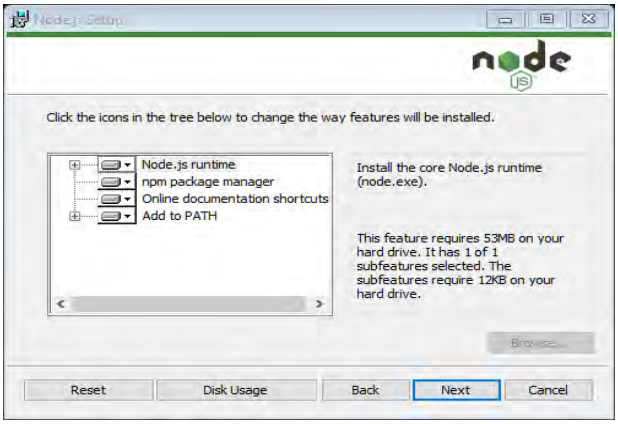

Figura 8. Selección de componentes de instalación del entorno de programación.

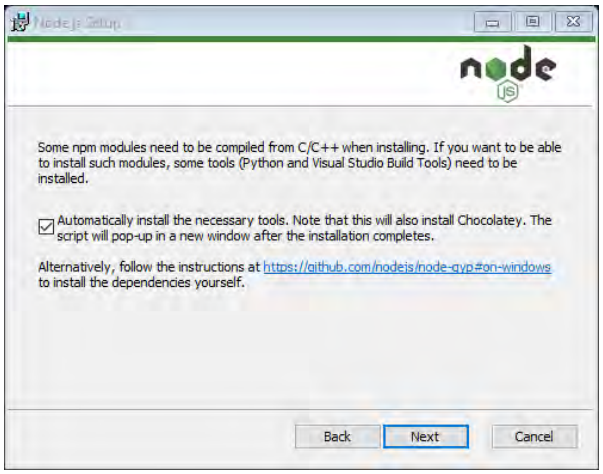

Figura 9. Selección de módulos de soporte de los lenguajes C/C++.

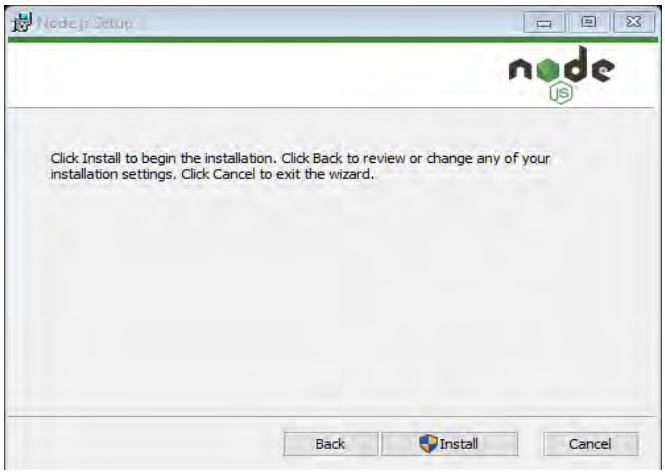

Figura 10. Inicio del proceso de instalación de *Node*.js.

para dar soporte a algunos componentes adicionales que forman parte del entorno de programación. Se recomienda que se habilite la opción de instalarlos automáticamente (Fig. 9), y continuar con la instalación pulsando el botón "*Next*".

Con el séptimo paso, se ha completado la configuración inicial del instalador, con lo que el proceso mostrará el diálogo de la figura 10, en donde al presionar el botón "*Install*", se inicia con el copiado de los archivos necesarios (Fig. 11), para contar con el entorno de ejecución disponible en el dispositivo de almacenamiento local, proceso que se finaliza en la pantalla que se muestra en la figura 12.

La instalación finaliza cuando se pulsa el botón "*Finish*", en donde una vez pulsado el botón se ejecutará una ventana del entorno MS-DOS con una pantalla que muestra el proceso de instalación de las herramientas adicionales de *Node*.js, tal y como se observa en la figura 13. Una vez finalizado el programa, se cierra la ventana emergente y con esto se da por finalizado el proceso de instalación, teniendo a partir de este momento las herramientas del lenguaje *JavaScript* que permiten gestionar los eventos de la aplicación *web* híbrida en la interfaz de usuario de la galería fotográfica que se implementará en la segunda etapa de la construcción de la aplicación móvil.

Instalación de las herramientas del framework Ionic

Como el desarrollo de la aplicación se realizará inicialmente en el entorno web de una computadora personal, es necesario que los diseños de las interfaces de usuario se puedan visualizar tal y como se mostrarán en los dispositivos móviles. Por lo anterior, se requiere establecer un esquema de tipo "vista del dispositivo", mismo que se proporciona en el framework Ionic. Para instalar esta herramienta se requiere la instalación del gestor de paquetes npm, a partir del cual se ejecuta el comando en la interfaz de línea de comandos:

npm install -g @ionic/cli native-run cordova-res

El comando utiliza el gestor de paquetes npm para instalar *Ionic* CLI (*Command-Line Interface*), que permite ejecutar una aplicación nativa del sistema operativo móvil en entornos *web*. Este *software* se utiliza para ejecutar los archivos binarios nativos en dispositivos, así como los simuladores / emuladores, a través del generador *Cordova*-res, el cual es una extensión

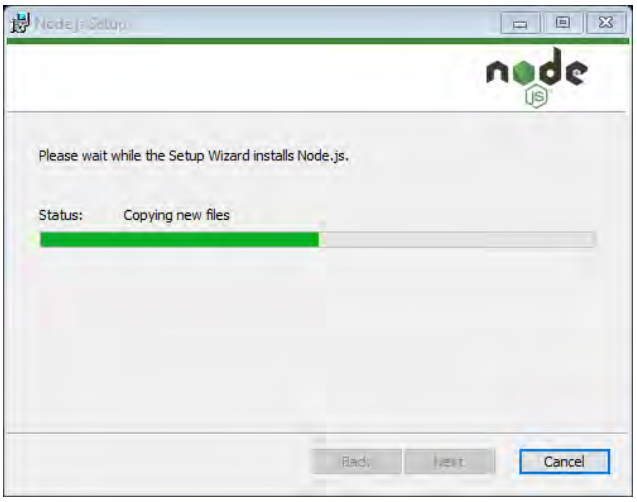

Figura 11. Ejecución del proceso de instalación.

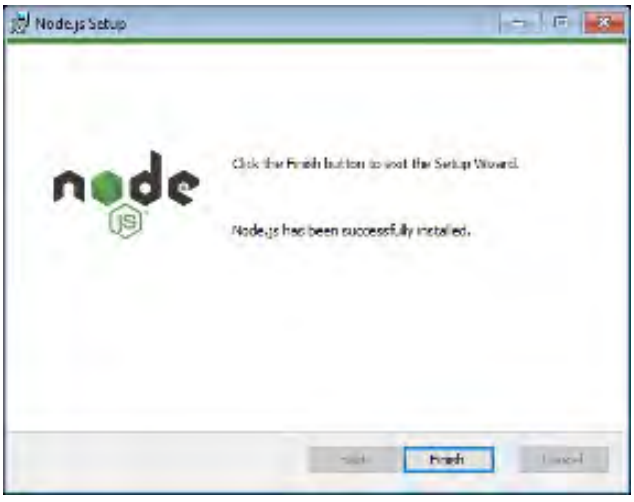

Figura 12. Vista inicial de la aplicación de la galería fotográfica.

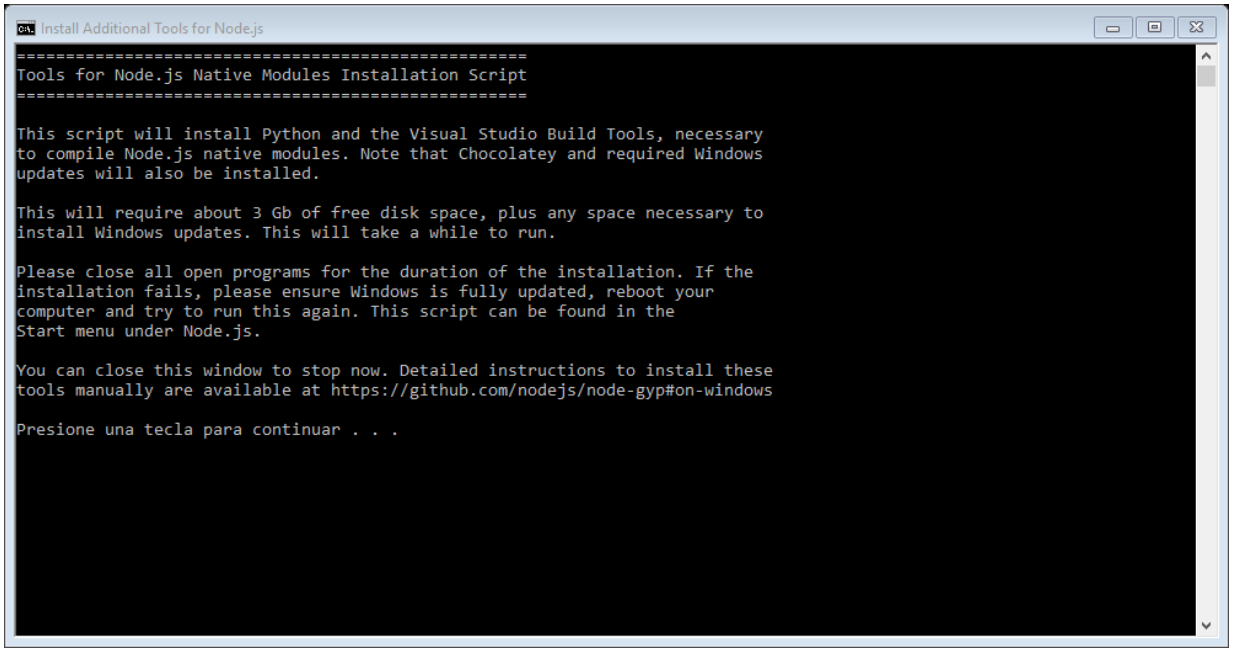

Figura 13. FInstalación de las herramientas adicionales de Node.js.

del servidor *Apache* (denominada *Apache Cordova*), que se utiliza para generar los íconos de las aplicaciones nativas y las pantallas iniciales de diseño en entornos *web* a través de código HTML, JavaScript y CSS.

Cordova cuenta con un conjunto de *APIS* que permiten desarrollar aplicaciones híbridas que se ejecutan en diferentes sistemas operativos como *Android*, *iOS*, *Windows Mobile* y *BlackBerry OS*. Una vez que se han instalado las herramientas ionicas y los entornos de trabajo de la aplicación, ahora se puede generar un proyecto para comenzar la programación de la aplicación híbrida.

Etapa 2. Construcción del proyecto de software de la aplicación híbrida

En esta etapa se crea una aplicación de la pantalla principal utilizando Angular, el cual es un *framework* de código abierto

desarrollado por *Google* para construir aplicaciones *web* sobre el lenguaje *JavaScript* con programación del lado del cliente (Basalo 2014). *Angular* utiliza el patrón MVC (Modelo-Vista-Controlador) para el desarrollo de aplicaciones web dinámicas, extendiendo la sintaxis de HTML para hacerlo más funcional. De esta forma, el servidor proporciona el contenido estático en forma de plantillas que contienen la información que se despliega (o los modelos), mientras que el cliente se encarga de reunir la información y el contenido, dando como resultado la vista.

Así, a través de *Ionic* Angular se puede crear una plantilla de inicio basada en pestañas que agregue *Capacitor*, un ejecutor nativo multiplataforma que facilita la creación de aplicaciones híbridas en entornos web, las cuales se ejecutan de forma nativa en *iOS*, *Android* y la *web*. En otras palabras, *Capacitor* proporciona en un proyecto *Ionic* un contenedor nativo que permite desplegar el contenido de una aplicación móvil primeramente en la *web*, antes de distribuirla a los sistemas operativos móviles. Para crear la pantalla principal de la aplicación, se ejecuta el comando ionic start en la interfaz CLI de Ionic:

ionic start photo-gallery tabs –type=angular –capacitor

El comando anterior generará un entorno de trabajo que almacenará la configuración básica de la aplicación móvil, la cual se almacenará en un directorio denominado "*photo-gallery*". El directorio contendrá los archivos de las vistas (documentos HTML) asociados, así como los archivos de soporte para la gestión de eventos de la aplicación híbrida (archivos ts), en donde se implementará la funcionalidad de la galería. Para trabajar sobre las vistas de la interfaz principal generada, se debe cambiar de directorio a la carpeta de la aplicación, para lo cual se ejecuta el comando:

cd photo-gallery

A continuación, se instalará *PWA Elements,* un paquete que es parte de *Ionic* que agrega algunos complementos de *Capacitor* al proyecto, entre estos complementos destacan los de la gestión de las cámaras y los sensores del dispositivo, siendo estos los que se utilizarán para ejecutar las aplicaciones móviles en entornos *web*. Para instalarlo se captura la siguiente instrucción en la interfaz de línea de comandos:

npm install @ionic/ pwa-elements

Una vez que se instalan *PWA elements*, el comando de instalación creará el archivo *main*.ts, el cual se almacenará en el directorio *@ionic/pwa-elements*. El archivo tendrá el contenido de la tabla III con la definición de los eventos necesarios para ejecutar la aplicación.

Ahora la aplicación está lista para ejecutarse. Para lanzar la ejecución, se introducirá el comando:

ionic serve

En la interfaz de línea de comandos utilizada, una vez que se compile el proyecto y se encuentre libre de errores, se mostrará la vista inicial de la interfaz gráfica de la aplicación, mostrando la estructura base como la que se muestra en la figura 15.

Cabe mencionar que cuando se genera un proyecto *Ionic* se invoca un método *Init* automáticamente, por lo que no será necesario que se efectúe en *Git*. El sistema *Git* se utiliza en los entornos de programación para establecer el control de las versiones del código fuente de la aplicación. *Git* es un *software* de código abierto (*open source*) que permite controlar las versiones de un proyecto, manteniendo los archivos de las aplicaciones organizados en un equipo de cómputo, e incluso permitiendo el trabajo de un proyecto de manera distribuida con

Tabla III. Definición inicial del archivo *main*.ts

```
1  import { enableProdMode } from '@angular/core';
2
3  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
4  import { AppModule } from './app/app.module';
5  import { environment } from './environments/environment';
6  //Línea agregada
7  import { defineCustomElements } from '@ionic/pwa-elements/loader';
8  if (environment.production) {
9   enableProdMode();
10 }
11
12 platformBrowserDynamic().bootstrapModule(AppModule)
13  .catch(err => console.log(err));
14
15 defineCustomElements(window);//Línea agregada
```

los integrantes de un equipo de programadores de aplicaciones de *software*.

De esta forma, *Git* permite auditar el código fuente, controlando los accesos y estableciendo que personal de desarrollo del proyecto ha realizado cambios en él, en qué instancia de tiempo y qué línea se ha modificado. Este tipo de herramientas es útil para poder realizar respaldos del código fuente de la aplicación y regresarlo a versiones anteriores para restaurar un proyecto de software antes de la ocurrencia de algún error o cambio significativo, por lo que contar con esta tecnología es de utilidad para gestionar todo el contenido de una aplicación, la administración de los paquetes o las actualizaciones que se requieran para un proyecto de software. Para introducir el control de versiones, basta con ejecutar los siguientes comandos desde la interfaz de línea de comandos:

git status
git add .
git commit –m "Agregando complementos de capacitor"

Creación de la interfaz gráfica principal de la galería de fotos

La ventana principal de la aplicación generada en la figura 14 posee tres pestañas, donde se programará a continuación la pestaña "Tab2" para que el evento clic despliegue la galería de todos, la cual se inicializa como un lienzo en blanco, el cual se establecerá como el lugar perfecto para transformarlo en una galería de fotos. La CLI de Ionic presenta Live Reload, una opción que permite visualizar los cambios realizados en una aplicación justo en el momento de modificarla, por lo que cuando en el código fuente se realizan los cambios y se guardan mostrado en la Figura 16, la aplicación se actualiza de inmediato en el navegador web.

Para visualizar el contenido de la figura 14, se utiliza el navegador *web Mozilla Firefox*, al cual se puede acceder al modo vista de teléfono mediante la siguiente sucesión de teclas *Ctrl* + *Shift* + *M*. A continuación, se abrirá la carpeta de la aplicación de la galería de fotos en el editor de código que elija. Posteriormente se

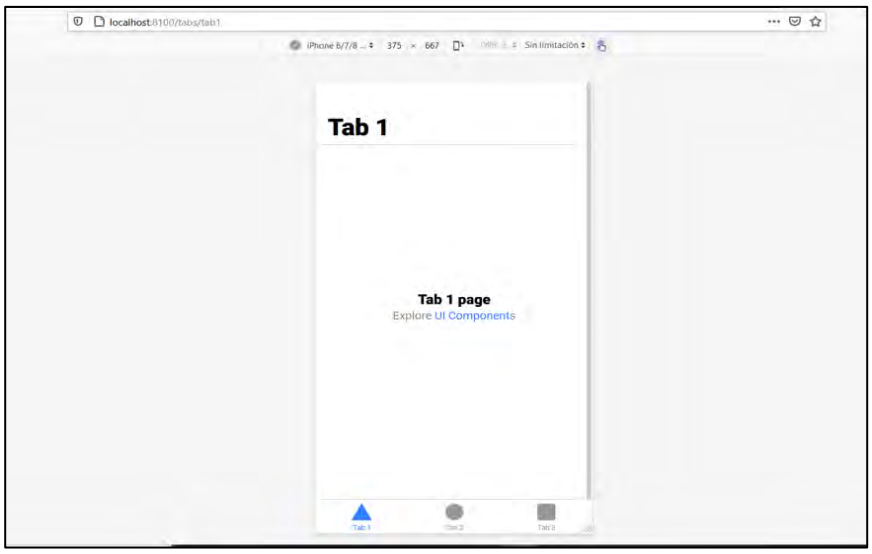

Figura 13. Vista inicial de la aplicación de la galería fotográfica.

seleccionará el archivo tab2.page.html (Fig. 14), el cual se localiza en el directorio /src/app/tab2/, y se le cambiará el título por "Galería de Fotos" (líneas 3-5). Se deberá abrir el archivo y editarlo, agregando el fragmento de código de las líneas 14 – 19. , se utiliza el navegador *web Mozilla Firefox*, al cual se puede acceder al modo vista de teléfono mediante la siguiente sucesión de teclas *Ctrl* + *Shift* + *M*. A continuación, se abrirá la carpeta de la aplicación de la galería de fotos en el editor de código que elija. Posteriormente se seleccionará el archivo tab2.page.html (Tabla IV), el cual se localiza en el directorio /src/app/tab2/, y se le cambiará el título por "Galería de Fotos" (líneas 3-5). Se deberá abrir el archivo y editarlo, agregando el fragmento de código de las líneas 14 – 19.

A continuación, se deberá abrir el archivo *tabs.page.html* ubicado en el directorio src/app / tabs/, y cambiar la etiqueta a "Fotos" y el nombre del ícono a "images", tal y como se muestra en la tabla V.

Se deberán guardar todos los cambios e inmediatamente se actualizará el diseño de la ventana principal en el navegador, tal y como se muestra en la figura 14.

Ahora se guardarán los cambios en el control de versiones de la aplicación de galería de fotos, basta con aplicar el

Figura 14. Establecimiento del comando "Fotos" del menú principal de la galería fotográfica.

Tabla IV. Código de la vista de ventana principal de la galería de fotos (archivo tab2.page.html)

```
<ion-header [translucent]="true">
  <ion-toolbar>
    <ion-title>
      Galería de Fotos <!--Cambio de título-->
    </ion-title>
  </ion-toolbar>
</ion-header>

<ion-content [fullscreen]="true">
  <!--Lo que se encontraba anteriormente en esta sección se elimina -->
  <!--Sección agregada-->
  <ion-fab vertical="bottom" horizontal="center" slot="fixed">
    <ion-fab-button>
      <ion-icon name="camera"></ion-icon>
    </ion-fab-button>
  </ion-fab>
  <!--Sección agregada-->
</ion-content>
```

Tabla V. Modificación del archivo tabs.page.html.

```
<ion-tab-button tab=”tab2”>
      <ion-icon name=“images“></ion-icon>
      <ion-label>Fotos</ion-label>
    </ion-tab-button>
```

comando “git” sobre el directorio de trabajo del proyecto, tal y como se muestra en los siguientes comandos introducidos en la interfaz de línea de comandos:

git status

git add .

git commit –m “Se añadio el icono de la camara y se cambio el nombre e icono del segundo tab”

Una vez que se ha habilitado la nueva versión del código fuente, se tiene la aplicación principal contenedora de la galería, a la cual se le añadirá la funcionalidad para poder capturar fotos con la cámara del dispositivo móvil en la siguiente fase de esta etapa en la construcción de la aplicación móvil.

Etapa 3. Programación de la cámara del dispositivo móvil

Una vez que se ha programado la interfaz principal de la galería, ahora se agregará a la aplicación la capacidad de tomar fotos con la cámara del dispositivo. Para ello se utilizará la API de cámara desde Capacitor, la cual se debe compilar para poder agregarla a la página web, y posteriormente se le deben realizar algunos ajustes complementarios para que sea funcional en los dispositivos móviles que utilizan los sistemas *iOS* y *Android*, proceso que sigue cuatro fases: la creación del servicio de fotografía, la preparación de la

aplicación para que la aplicación sea habilitada para tomar fotos usando la cámara del dispositivo móvil, el establecimiento de las rutinas de guardado y carga de las fotografías desde el sistema de archivos del dispositivo móvil.

Creación del servicio de fotografía

Toda la lógica del condensador (que comprende el uso de la cámara y otras características nativas de los dispositivos móviles) se encapsulará en una clase de servicio. Para ello, se creará el servicio denominado PhotoService, el cual se genera usando el comando ionic generate:

ionic g service services/photo

Una vez que se ha generado el servicio, este comando creará el archivo photo.service.ts que se almacenará en el directorio services de la aplicación. A continuación, se abre el archivo y se agrega la lógica que impulsará la funcionalidad de la cámara, tal y como se observa en la tabla VI. Primero, se deberán importar las dependencias de *Capacitor* y obtener referencias a los complementos de Cámara, Sistema de archivos y Almacenamiento (líneas 1-4).

A continuación, se modificará el archivo tab2.page.ts mostrado en la tabla VII, y se definirá un nuevo método de clase llamado *addNewToGallery* (líneas 11 – 18), que contendrá la lógica central para tomar una fotografía del dispositivo y guardarla en el sistema de archivos, comenzando con la apertura del dispositivo (línea 13). Una vez generados los comandos para invocar la cámara del dispositivo, se abre el archivo de soporte de las clases de la ventana principal (tab2.page.ts de la tabla VI) y se importa la clase *PhotoService* (línea 1) y se agrega el método *addPhotoToGallery*() definido en las líneas 16-18. Este método deberá definir la llamada al método *addNewToGallery* (línea 17), en el cual se programará la funcionalidad para que la fotografía capturada se agregue a la galería.

Ahora se creará un evento *script* en la

Tabla VI. Creación de la clase PhotoService (archivo photo.service.ts).

```
1  import { Injectable } from '@angular/core';
2  import { Plugins, CameraResultType, Capacitor, FilesystemDirectory,
3    CameraPhoto, CameraSource } from '@capacitor/core';
4  const { Camera, Filesystem, Storage } = Plugins;
5
6  export class PhotoService {
7
8    constructor() { }
9
10   /*Método agregado a la clase*/
11   public async addNewToGallery() {
12
13     const capturedPhoto = await Camera.getPhoto({
14       resultType: CameraResultType.Uri,
15       source: CameraSource.Camera,
16       quality: 100
17     });
18   }
19 }//End of class
```

página de la vista HTML de la ventana principal (archivo tab2.page.html), donde se llamará a la función *addPhotoToGallery* () del código fuente de la tabla VII, cuando se pulsa en el botón de acción flotante de la ventana principal, definiendo en el archivo el código fuente de la tabla VIII.

Guarde el archivo y, si aún no se está ejecutando, reinicie el servidor de desarrollo en su navegador ejecutando el comando ionic serve. En la pestaña Galería de fotos, se deberá hacer clic en el botón Cámara. Si la computadora tiene una cámara *web* de algún tipo, aparece una ventana modal, la cual se muestra al frente de todas las ventanas y adquiere el control principal de la pantalla del dispositivo. Ahora la aplicación se encuentra lista para poder tomar una *selfie* o autorretrato o una captura con las cámaras del dispositivo, por ejemplo la que se observa en la figura 15. Nota: Para crear una aplicación para iOS, se necesitará una computadora Mac.

Tabla VII. Creación de la clase PhotoService (archivo photo.service.ts).

```
1  import { PhotoService } from '../services/photo.service';/**Importanción agregada */
2  import { Component } from '@angular/core';
3
4  @Component({
5    selector: 'app-tab2',
6    templateUrl: 'tab2.page.html',
7    styleUrls: ['tab2.page.scss']
8  })
9
10 export class Tab2Page {
11   /**
12    * Bloque de código agragdo
13    */
14   constructor(public photoService: PhotoService) {}
15
16   addPhotoToGallery() {
17     this.photoService.addNewToGallery();
18   }
19 }
```

Tabla VIII. Código de evento clic del documento de vista tab2.page.html.

```
1  <ion-fab vertical="bottom" horizontal="center" slot="fixed">
2
3    <ion-fab-button (click)="addPhotoToGallery()">
4      <ion-icon name="camera"></ion-icon>
5    </ion-fab-button>
6  </ion-fab>
```

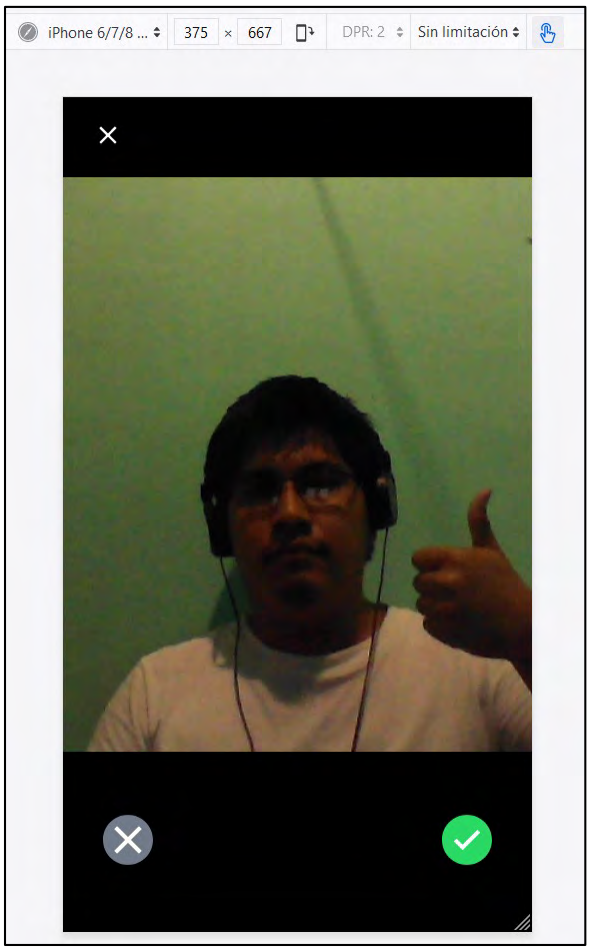

Figura 14. Prueba de ejecución de captura de una *selfie*.

Después de tomar una fotografía, la ventana de la cámara desaparecerá de inmediato, por lo que los datos de la fotografía se perderán a menos que se introduzcan en la aplicación y se guarden para poder visualizarla en la galería, siendo esta funcionalidad la siguiente en definirse en la elaboración de la aplicación.

Definición del contenedor para visualización de las fotografías

Para lograr la visualización de las fotografías capturadas en la aplicación, hay que establecer en la clase *PhotoService* (Tabla IX), la adición de una nueva interfaz externa a la definición de la clase, la cual se denominará *Photo* (líneas 6-10), y que contendrá los metadatos de las fotografías almacenados en el vector photos, el cual se utilizará más adelante. En la función *addNewToGallery* (líneas 16-27), se agrega la fotografía recién capturada al comienzo de la matriz de fotos (líneas 23 - 26).

A continuación, se modifica el archivo tab2.page.html mostrado en la tabla X, para que se muestre la imagen capturada en la pantalla. Para ello se debe agregar un componente de cuadrícula para que cada foto se muestre conforme se capturan las fotos y se agreguen a la galería. Una vez que se añada, la aplicación recorrerá cada foto en la matriz de fotos de la *PhotoServices*, agregando un componente de imagen (<*ion-img*>) para cada una, incluyendo la referencia hacia el directorio src (fuente) con la ruta del archivo de la imagen.

Una vez que se hayan guardado todos todos los archivos, se puede probar en el navegador *web* la interacción para visualizar el evento del botón cámara y tomar una fotografía, mostrándola en la galería tal y como se ejemplifica en la figura 15.

Una vez que se han aplicado los cambios, se guarda la siguiente versión del código con la funcionalidad de tomar fotos y se muestran en la pantalla a través de los siguientes comandos Git:

git status

git add .

git commit –m "Se añadio la funcionalidad de tomar fotos y mostrarlas en pantalla"

A continuación, se agregará el soporte para guardar las fotos en el sistema de archivos, para que puedan recuperarse y mostrarse en la aplicación en un momento en el momento deseado.

Tabla IX. Adición de la visualización de las fotos en la aplicación móvil del archivo photo.service.ts.

```
import { Injectable } from '@angular/core';
import { Plugins, CameraResultType, Capacitor, FilesystemDirectory,
  CameraPhoto, CameraSource } from '@capacitor/core';
const { Camera, Filesystem, Storage } = Plugins;

//Código agregado
export interface Photo {
   filepath: string;
   webviewPath: string;
}

export class PhotoService {
 //Código
 constructor() { }

 public async addNewToGallery() {
  // Código
  const capturedPhoto = await Camera.getPhoto({
    resultType: CameraResultType.Uri,
    source: CameraSource.Camera,
    quality: 100
  });
  this.photos.unshift({
    filepath: "soon…",
    webviewPath: capturedPhoto.webPath
  });
 }
}//End of class
```

Tabla X. Código de la vista de ventana principal de la galería de fotos (archivo tab2.page.html)

```
<ion-grid>
   <ion-row>
   <ion-col size="6"
     *ngFor="let photo of photoService.photos; index as position">
       <ion-img [src]="photo.webviewPath"></ion-img>
   </ion-col>
   </ion-row>
  </ion-grid>
  <!--Más código-->
</ion-content>
```

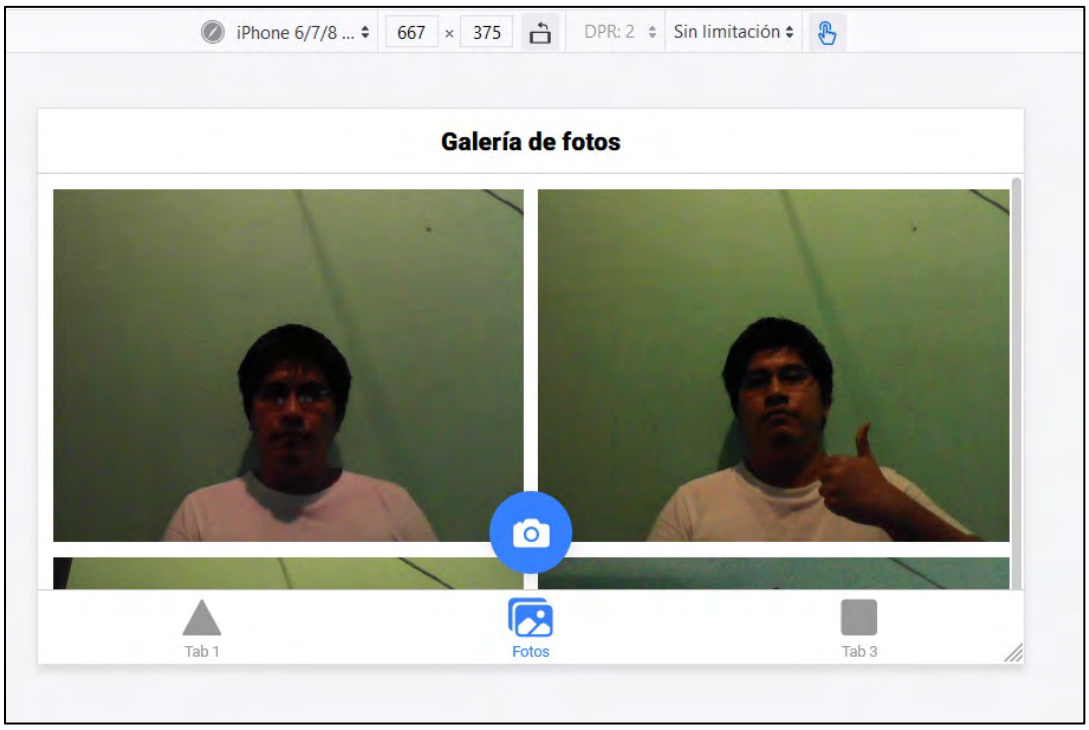

Figura 15. Vista de la galería de fotos en la ventana principal de la aplicación móvil.

Almacenamiento de las capturas de la cámara en el sistema de archivos

Ahora se pueden tomar varias fotografías y mostrarlas en una galería de fotos en la segunda pestaña de la aplicación. Sin embargo, estas fotos no se almacenan de forma permanente, por lo que cuando se cierre la aplicación, se eliminarán. Para guardarlas en el sistema de archivos se requiere de la secuencia de pasos que se muestran en el código fuente de la tabla XI, en los cuales se utiliza la *API Filesystem*. El proceso comienza con la creación de un nuevo método de clase, denominado *savePicture*( ) que se define en la clase *PhotoService* (del archivo src/app/services/photo.service.ts), el cual se invoca en la línea 9. A este método se le pasa como parámetro en el objeto *cameraPhoto* capturado, inicializado en las líneas 2-6, que representa la foto del dispositivo recién capturada. El procesamiento de la operación de guardado se implementa en la definición del método *addNewToGallery* (), tal y como se muestra en la tabla X, mientras que el cuerpo del método *savePicture* ( ) se observa en el código fuente de la figura 15.

El código de la tabla XII utiliza la API del sistema de archivos *Capacitor* para guardar la foto capturada en el sistema de archivos. Para ello, se requiere convertir la foto al formato base64 (línea 4), para posteriormente introducir los datos convertidos en el formato de imagen JPEG (línea 7) en la función *writeFile*( ) del sistema de archivos que se invoca en la líneas 8-12. Con ello, se mostrará cada foto en la pantalla estableciendo la ruta de origen de cada imagen en el atributo src en la línea <ion-img [src]="photo.webviewPath"></ion-img> en tab2.page.html en la propiedad *webviewPath*.

La función *readAsBase64* () de la línea 4 del código de la tabla XII es una función auxiliar que se define a continuación. Esta función se agrega en el código fuente para organizar el flujo de la aplicación a

Tabla XI. Implementación de la rutina de guardado de las imágenes en la galería (archivo photo.service.ts).

```
1  public async addNewToGallery() {
2    const capturedPhoto = await Camera.getPhoto({
3      resultType: CameraResultType.Uri,
4      source: CameraSource.Camera,
5      quality: 100
6    });
7
8    // Guarda la imagen y la agrega en la colleción de fotos
9    const savedImageFile = await this.savePicture(capturedPhoto);
10   this.photos.unshift(savedImageFile);
11 }
```

Tabla XII. Implementación de la rutina savePicture (código añadido al archivo photo.service.ts)

```
1  private async savePicture(cameraPhoto: CameraPhoto) {
2    // Convierta la foto al formato base64, requerido por la API del sistema de
3   archivos para guardar
4    const base64Data = await this.readAsBase64(cameraPhoto);
5
6    // Escribe el archivo en el directorio de datos.
7    Const fileName = new Date().getTime() + '.jpeg';
8    const savedFile = await Filesystem.writeFile({
9      path: fileName,
10     data: base64Data,
11     directory: FilesystemDirectory.Data
12   });
13
14   // Use webPath para mostrar la nueva imagen en lugar de base64 ya que
15   // ya esta cargada en la memoria
16   return {
17     filepath: fileName,
18     webviewPath: cameraPhoto.webPath
19   };
20 }
```

través de un método separado, ya que en ella se establece una pequeña cantidad de lógica específica para adaptar el tipo definido para las fotografías de la aplicación en las distintas plataformas en que ésta se ejecuta (por ejemplo, página *web* frente a dispositivos móviles). El detalle de esta lógica para ejecutar la aplicación en la *web* se observa en el código fuente de la tabla XIII.

Obsérvese que obtener la foto de la cámara en formato base64 en la web parece ser un poco más complicado que en un dispositivo móvil, aunque en realidad,

Tabla XIII. Lógica de guardado de las fotografías para los dispositivos móviles.

```
1  private async readAsBase64(cameraPhoto: CameraPhoto) {
2    // Obtenga la foto, léala como un blob y luego conviértala al formato base64
3    const response = await fetch(cameraPhoto.webPath!);
4    const blob = await response.blob();
5
6    return await this.convertBlobToBase64(blob) as string;
7  }
8
9  convertBlobToBase64 = (blob: Blob) => new Promise((resolve, reject) => {
10   const reader = new FileReader;
11   reader.onerror = reject;
12   reader.onload = () => {
13     resolve(reader.result);
14   };
15   reader.readAsDataURL(blob);
16 });
```

solo se están utilizando las API web integradas: fetch () de la línea 3 como una forma ordenada de leer el archivo en formato de blob (línea 4), posteriormente el método readAsDataURL () de FileReader de la línea 15 se utiliza para convertir el tipo blob contenedor de las fotos en el tipo base64. Con este código implementado, cada vez que se toma una nueva foto ahora se guarda automáticamente en el sistema de archivos, por lo que se confirma que se ha actualizado la nueva versión en el directorio de trabajo con los comandos:

git status

git add .

git commit –m "Se guarda la foto en formato Base64"

Hasta este momento, se ha generado la galería de fotos, se capturan las fotografías desde la cámara y se almacenan en el sistema de archivos del dispositivo que la ejecuta. En este punto, la aplicación no ha sido dotada aún de la capacidad de cargar las fotografías desde el sistema de archivos para desplegarlas en la aplicación, por lo que se requiere de la programación de las rutinas necesarias para alojarlas en la vista de la galería, siendo esto la siguiente acción a programar en la aplicación.

Carga de las fotos a la aplicación desde el sistema de archivos

La siguiente fase de implementación de la aplicación galería consiste en la implementación de la estructura de memoria de la aplicación que almacene las fotografías que se tomaron desde la cámara y las cargue desde el almacenamiento en el sistema de archivos. Esto se logra con la construcción de la API de almacenamiento que implementa Capacitor (definida en la tabla XIV), la cual permite almacenar la galería de fotos en el vector *Photo* definido en la línea 2, para poder acceder a ellas usando un valor de clave como índice.

Para definir la funcionalidad restante, se agrega al final de la función *addNewToGallery* del archivo *photo.service.ts* una llamada al método *Storage.set* () de la tabla XV. Este método permite que se guarde una nueva foto capturada con la cámara

Tabla XIV. Definición de la memoria de almacenamiento de la galería de fotos (archivo photo.service.ts).

```
1  public async addNewToGallery() {
2    const capturedPhoto = await Camera.getPhoto({
3      resultType: CameraResultType.Uri,
4      source: CameraSource.Camera,
5      quality: 100
6    });
7
8    // Guarda la imagen y la agrega en la colleción de fotos
9    const savedImageFile = await this.savePicture(capturedPhoto);
10   this.photos.unshift(savedImageFile);
11 }
```

Tabla XV. Establecimiento de la memoria persistente de la galería de fotos.

```
1  Storage.set({
2      key: this.PHOTO_STORAGE,
3      value: JSON.stringify(this.photos)
4    });
```

Tabla XVI. Implementación de la rutina loadSaved para inicializar la memoria persistente de la galería.

```
1  public async loadSaved() {
2    // Retrieve cached photo array data
3    const photoList = await Storage.get({ key: this.PHOTO_STORAGE });
4    this.photos = JSON.parse(photoList.value) || [];
5
6    // más...
7  }
```

en la matriz de fotos. De esta forma, la memoria de la galería almacenada en el vector persiste aún cuando el usuario cierra o cambia el flujo de instrucciones del dispositivo a una aplicación diferente.

Con los datos de las imágenes capturadas de la cámara en el vector de fotos, se define ahora una función llamada *loadSaved* (), que permite recuperar los datos de las fotos en el momento deseado. El método utiliza la misma clave para recuperar el vector de fotos en formato JSON (*JavaScript Object Notation*) con el método *parse* () de la línea 4 del código de la tabla XVI.

Esta inicialización de la estructura de memoria permitirá establecer directamente en los dispositivos móviles la fuente de una imagen en una para cada archivo de foto existente en el sistema de archivos, mostrándolos automáticamente en la galería. En la *web;* sin embargo, no ocurre de la misma forma sino que para que el código sea funcional se debe leer cada imagen del sistema de archivos en formato *base64*, usando una nueva propiedad *base64* en el objeto *Photo*. Esto se debe a que la API del sistema de archivos usa *IndexedDB* para almacenar datos dentro del navegador.

Tabla XVII. Inicialización de la galería de fotos (archivo photo.service.ts).

```
1    for (let photo of this.photos) {
2      const readFile = await Filesystem.readFile({
3        path: photo.filepath,
4        directory: FilesystemDirectory.Data
5      });
6
7      photo.webviewPath = `${readFile.data}`;
8    }
```

Tabla XVIII. Inicialización de la galería de fotos en la carga de la aplicación (archivo tab2.page.ts).

```
1    async ngOnInit() {
2     await this.photoService.loadSaved();
3    }
```

A continuación, el código de la tabla XVII implementa esta funcionalidad dentro de la definición de la función *loadSaved* () del archivo *photo.service.ts*.

Para que el código de la tabla XVII se refleje en la vista principal de la galería, se debe invocar este nuevo método en el archivo tab2.page.ts, para que cuando el usuario navegue por primera vez a la pestaña 2 (la galería de fotos), todas las fotos se carguen y se muestren en la pantalla, lo cual se establece con la llamada al método *ngOnInit*() de la tabla XVIII.

Con las implementaciones de las dos etapas se tiene una aplicación *Ionic* que es funcional en entornos *web*. Para continuar con el desarrollo de la galería se actualizan las versiones de los archivos de código fuente en el directorio contenedor con los siguientes comandos:

git status

git add .

git commit –m "Se cargan las fotos tomadas en la pantalla"

Ahora la aplicación se encuentra lista para ejecutarse en entornos *web* con todas las funcionalidades deseadas. En este punto, la aplicación híbrida desarrollada en el entorno web ya se puede transferir a una aplicación nativa que sea capaz de ejecutarse sobre un dispositivo móvil *iOS* y *Android*. Sin embargo, para ello requiere de una API que establezca que el código *web* desarrollado en el *framework* sea ejecutable en entornos nativos, así como de la configuración que permita administrar los sistemas de archivos de forma adecuada para la lectura y almacenamiento de las imágenes en los entornos *web* (caso híbrido) y en los sistemas operativos móviles (caso nativo), elementos que se implementan en la cuarta etapa del desarrollo de la aplicación móvil.

Etapa 4. Adaptabilidad de la aplicación híbrida para su uso en los sistemas operativos de los dispositivos móviles

Para que la aplicación de galería de fotos se ejecute en *iOS*, *Android* y la *Web* de forma similar en cuanto al diseño y la funcionalidad, se utiliza la base de código fuente generada hasta este punto; aunque para que la aplicación se pueda distribuir hacia entornos nativos *iOS* e *Android*, se requiere de la implementación de algunos

Tabla XIX. Uso de la API de importación de la plataforma (archivo photo.service.ts).

```
1     // Otro código
2  import { Platform } from '@ionic/angular';
3  // Otro código
4  export class PhotoService {
5    public photos: Photo[] = [];
6    private PHOTO_STORAGE: string = "photos";
7    private platform: Platform;
8
9    constructor(platform: Platform) {
10     this.platform = platform;
11   }
12 // Otro código
13
14 }
```

cambios lógicos para que la interfaz de la galería sea admitida por las plataformas móviles. Para lograr esta adaptación del código fuente se requiere de la instalación de algunas herramientas nativas que permitirán ejecutar la aplicación en un dispositivo móvil, lo cual se logra a través de la creación de una API de plataforma de importación.

Construcción de una API de plataforma de importación

Para que la aplicación generada se pueda ejecutar en un dispositivo móvil, se modifica el archivo photo.service.ts como observa en la tabla XIX, importando la API de la plataforma Ionic (línea 2), que se utiliza para recuperar la información sobre el dispositivo actual en tiempo de ejecución a través del objeto Platform definido en la línea 7. De esta forma, el código del método constructor utiliza la información proporcionada por este objeto de la API para que la aplicación detecte de forma automática qué código debe ejecutar según la plataforma en la que se ejecuta la aplicación (web o móvil), funcionalidad que se implementa en la definición del método constructor () líneas 9-11.

Una vez que la aplicación es capaz de identificar la plataforma que la ejecuta en tiempo de ejecución, ahora se debe establecer la lógica para que las rutinas creadas en el entorno web sean funcionales en los sistemas operativos móviles, por lo que se debe generar la lógica específica para estas plataformas, siendo las que brindan la capacidad "híbrida" a la galería implementada en la aplicación.

Creación de la lógica específica de la plataforma

Para generar la lógica de la aplicación híbrida, primero se debe actualizar la funcionalidad para que la rutina de guardado de las fotografías sea compatible con dispositivos móviles. Para ello, se edita la función *readAsBase64* () del archivo *photo.service.ts* tal y como se muestra en la tabla XX. En este método se añaden las sentencias de control que verifiquen en qué plataforma se ejecuta la aplicación. Si la plataforma es "híbrida" (líneas 3-9)

Tabla XX. Implementación de la lógica de la plataforma en el archivo photo.service.ts.

```
1  private async readAsBase64(cameraPhoto: CameraPhoto) {
2    // "hybrido"
3    if (this.platform.is('hybrid')) {
4      // Lee el archivo en formato base64
5      const file = await Filesystem.readFile({
6        path: cameraPhoto.path
7      });
8      return file.data;
9    }
10   else {
11     //La convierte en formato base64
12     const response = await fetch(cameraPhoto.webPath);
13     const blob = await response.blob();
14     return await this.convertBlobToBase64(blob) as string;
15   }
16 }
```

(se implementan las *APIS* de *Capacitor* o *Cordova*, para utilizar los entornos nativos en tiempo de ejecución), entonces se lee el archivo de la foto en formato *base64* (líneas 5-7) usando el método *readFile* () del sistema de archivos. De lo contrario, se utiliza la lógica implementada para ejecutar la aplicación en la *web*, la cual se define en esta modificación del archivo en las líneas 10-16.

De la misma forma, se actualiza el método *savePicture*() tal y como se muestra en la tabla XXI, para que cuando la aplicación se ejecute en un dispositivo móvil (líneas 14-20), se establezca la ruta de archivo en el resultado de la operación *writeFile*(): *SavedFile.uri*, utilizando el método especial *Capacitor.convertFileSrc*() para configurar el *webviewPath* de la vista principal de la galería (en el archivo tab2. page.html).

En caso de que la aplicación se ejecute sobre entornos *web*, se ejecutan las líneas 21 - 26 con la funcionalidad definida para la aplicación. Ahora, al igual que en la rutina *savePicture*(), se edita la función *loadSaved*() implementada para que la aplicación se ejecute en la *web*, añadiendo la sentencia de control que permita validar la plataforma que ejecuta la aplicación, tal y como se muestra en la tabla XXII. En el caso de que la aplicación se ejecute en dispositivos móviles (líneas 8-20), se establecen directamente las fuentes de las imágenes en una etiqueta - - para cada archivo de foto existente en el sistema de archivos, mostrándolos automáticamente en la galería. En el caso de la web (líneas 16 -17), solo se requiere leer cada imagen del sistema de archivos en formato base64. Actualice esta función para agregar una sentencia de control (líneas 8-15) alrededor del código que gestiona los accesos al sistema de archivos.

Una vez que se haya generado la lógica de la plataforma, al guardar los cambios en el archivo photo.service.ts, la galería de fotos ahora contendrá una base de código que se ejecutará de la misma forma en la web, Android e iOS, por lo que se recomienda que se actualicen las versiones del

Tabla XXI. Establecimiento de la lógica de la plataforma para la rutina de almacenamiento de las fotografías (archivo photo.service.ts).

```
1  private async savePicture(cameraPhoto: CameraPhoto) {
2    // Convierta la foto al formato base64, requerido por la API del sistema de archivos
     para guardar
3    const base64Data = await this.readAsBase64(cameraPhoto);
4    const fileName = new Date().getTime() + '.jpeg';
5    const savedFile = await Filesystem.writeFile({
6      path: fileName,
7      data: base64Data,
8      directory: FilesystemDirectory.Data
9    });
10
11   if (this.platform.is('hybrid')) {
12     // Muestra la nueva imagen reescribiendo la ruta 'file: //' a HTTP
13     return {
14       filepath: savedFile.uri,
15       webviewPath: Capacitor.convertFileSrc(savedFile.uri),
16     };
17   }
18   else {
19     return {
20       filepath: fileName,
21       webviewPath: cameraPhoto.webPath
22     };
23   }
24 }
```

código fuente utilizando los siguientes comandos git:

git status

git add .

git commit –m "Se añadio la logica para Android e iOS"

Ahora la aplicación está lista para poder distribuirse en las plataformas de los dispositivos móviles, requiriendo para ello del soporte del entorno de ejecución Capacitor que permite la migración de la aplicación desde Ionic hacia las plataformas de los sistemas operativos nativos, siendo este proceso la tercera etapa de la implementación de la galería.

Etapa 5. Distribución de la aplicación hacia los sistemas operativos móviles iOS y Android

Desde que se agrega *Capacitor* al proyecto cuando se creó por primera vez, solo quedan unos pocos pasos hasta que la aplicación Galería de fotos esté en nuestro dispositivo.

Configuración del condensador. *Capacitor* es el entorno de ejecución de aplicaciones oficial de Ionic que facilita la implementación de aplicaciones *web* en plataformas nativas como *iOS*, *Android* y más. Si todavía se está ejecutando servicio iónico en la terminal, se deberá cancelar. Se completará una nueva compilación del

Tabla XXII. Implementación de la lógica de la plataforma (archivo photo.service.ts).

```
public async loadSaved() {
  // Retrieve cached photo array data
  const photoList = await Storage.get({ key: this.PHOTO_STORAGE });
  this.photos = JSON.parse(photoList.value) || [];

  // Easiest way to detect when running on the web:
  // “when the platform is NOT hybrid, do this”
  if (!this.platform.is('hybrid')) {
    // Display the photo by reading into base64 format
    for (let photo of this.photos) {
      // Read each saved photo's data from the Filesystem
      const readFile = await Filesystem.readFile({
          path: photo.filepath,
          directory: FilesystemDirectory.Data
      });
      // Web platform only: Load the photo as base64 data
      photo.webviewPath = `${readFile.data}`;
    }
  }
}
```

proyecto Ionic, corrigiendo los errores que informa:

ionic build

A continuación, se cran los proyectos *iOS* y *Android* de la aplicación:

ionic cap add ios

ionic cap add android

Se crean las carpetas *Android* e *iOS* en el directorio raíz del proyecto, siendo estas carpetas contenedoras de los proyectos nativos de ambos sistemas operativos móviles, los cuales son totalmente independientes que deben considerarse parte de su aplicación *Ionic* (es decir, es necesario verificarlos en el control de código fuente, editarlos usando sus herramientas nativas, etc.).

Cada vez que se realice una compilación (por ejemplo, una compilación iónica) que actualice els directorio *web* (predeterminado: compilación), se deberán copiar esos cambios en sus proyectos nativos:

ionic cap copy

Después de realizar actualizaciones a la parte nativa del código (como agregar un nuevo complemento), se debe usar el comando de sincronización:

ionic cap sync

Implementación de la aplicación para iOS

Las aplicaciones de *iOS* de capacitor se configuran y administran a través de Xcode, el entorno de desarrollo integrado de los sistemas *iOS* y *Mac* de *Apple*, el cual utiliza con dependencias administradas por *CocoaPods*. Antes de ejecutar la aplicación en un dispositivo *iOS*, se debe contar con la configuración adecuada, , primero se ejecutará el comando capacitor open, que abre el proyecto nativo de *iOS* en

Xcode:

ionic cap open ios

Para que funcionen algunos complementos nativos, se deben configurar los permisos de usuario tal y como se muestran en la figura 16. En nuestra aplicación de galería de fotos, esto incluye el complemento de la cámara: *iOS* muestra un cuadro de diálogo modal automáticamente después de la primera vez que se llama a *Camera.getPhoto* (), solicitando al usuario que permita que la aplicación use la cámara. El permiso que impulsa esto se denomina "Privacidad: uso de la cámara". Para configurarlo, se debe modificar el archivo *Info.plist*. Para acceder a él, se deberá hacer clic en "Información", luego expandir "Propiedades de destino personalizadas de *iOS*".

Cada ajuste en *Info. plist* tiene un nombre de parámetro de bajo nivel y un nombre de alto nivel. De forma predeterminada, el editor de la lista de propiedades muestra los nombres de alto nivel, pero a menudo es útil cambiar para mostrar los nombres sin formato de bajo nivel. Para hacer esto, se deberá hacer clic con el botón derecho en cualquier lugar del editor de lista de propiedades y cambiar "Claves / valores sin procesar". Ahora se deberán buscar la clave NSCameraUsageDescription y establecer el valor en algo que describa por qué la aplicación necesita usar la cámara, como "Para tomar fotos". El campo Valor se muestra al usuario de la aplicación cuando se abre la solicitud de permiso. A continuación, se deberá hacer clic en Aplicación (*App*) en el Navegador de proyectos en el lado izquierdo, luego dentro de la sección Firma y capacidades (*Signing & Capabilities*) y se deberá seleccionar el equipo de desarrollo, tal y como se observa en la figura 17.

Con los permisos establecidos y el equipo de desarrollo seleccionado, se puede probar la aplicación en un dispositivo real. Para ello solo se debe conectar un dispositivo *iOS* a la computadora *Mac*, selecciónelo (*App -> Matthew's iPhone for me*), posteriormente se pulsa elbotón "*Build*" tal y como se muestra en la figura 18 para construir, instalar e iniciar la aplicación en su dispositivo:.

Al pulsar el botón con el ícono de la cámara en la pestaña Galería de fotos, se mostrará el mensaje solicitud de permisos (figura 40). Una vez que se pulsa el botón "Aceptar", se habilita la aplicación para poder tomar fotografías con la cámara, para luego mostrarla en la aplicación.

Implementación de la aplicación para Android

Las aplicaciones de Android Capacitor

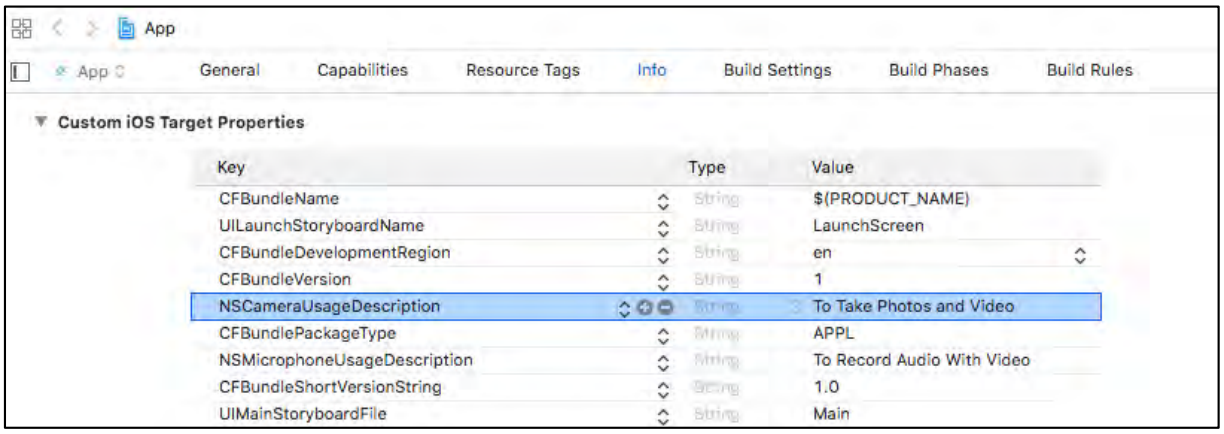

Figura 16. Propiedades de la aplicación nativa generada para iOSe.

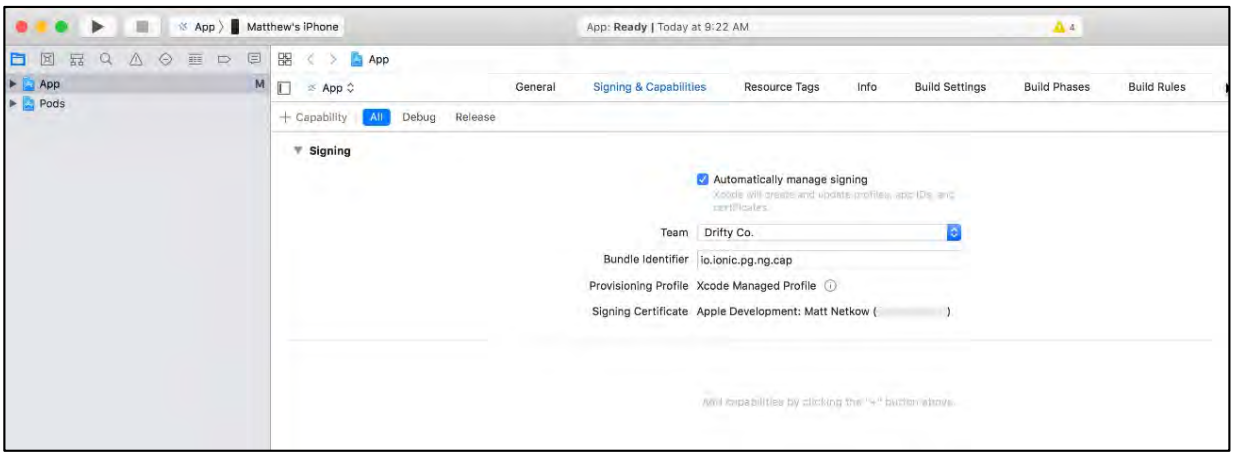

Figura 17. Configuración de la ejecución de la aplicación en Xcode.

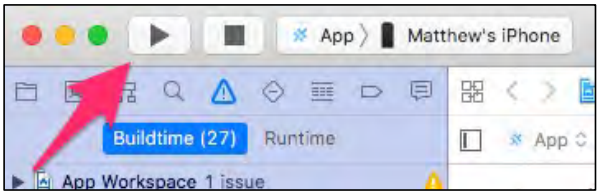

Figura 18. Creación de la aplicación nativa iOS.

se configuran y administran a través de *Android Studio*. Antes de ejecutar esta aplicación en un dispositivo *Android*, se deben completar un par de pasos. Primero, se ejecutará el comando capacitor open, que abre el proyecto nativo de Android en *Android Studio*.

Figura 18. Prueba de funcionalidad de la aplicación en un dispositivo con sistema operativo iOS.

ionic cap open Android

Al igual que en *iOS*, se deben habilitar los permisos para poder utilizar la cámara. Éstos se configuran en el archivo AndroidManifest.xml visualizado en la figura 19. Es probable que *Android Studio* abra este archivo automáticamente, aunque en caso de que no se muestre de forma automática, se puede ubicar en la barra de tareas del panel superior de *android studio*, en la opción: *android / app / src / main /*, o desde el panel de explorador de proyectos en la opción */app/ manifest/*.

Es necesario desplazarse hasta la sección Permisos (etiquetas uses-permission), y asegurarse de que se incluyan las siguientes entradas:

<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>

<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

Estas dos etiquetas permitirán que la aplicación pueda utilizar la cámara para leer y almacenar las imágenes desde el almacenamiento secundario del dispositivo. A continuación se deberá guardar el archivo para que se apliquen los cambios realizados. Ahora con los permisos de lectura/escritura establecidos, al igual que en el caso de *iOS*, se puede probar la aplicación en un dispositivo real (véase Fig. 20), simplemente conectando el dispositivo *Android* a su computadora, y en la aplicación *Android Studio* se deberá pulsar el botón "Ejecutar", seleccionando el dispositivo *Android* adjunto, para posteriormente pulsar el botón "Aceptar", acción con la que se compilará, instalará e

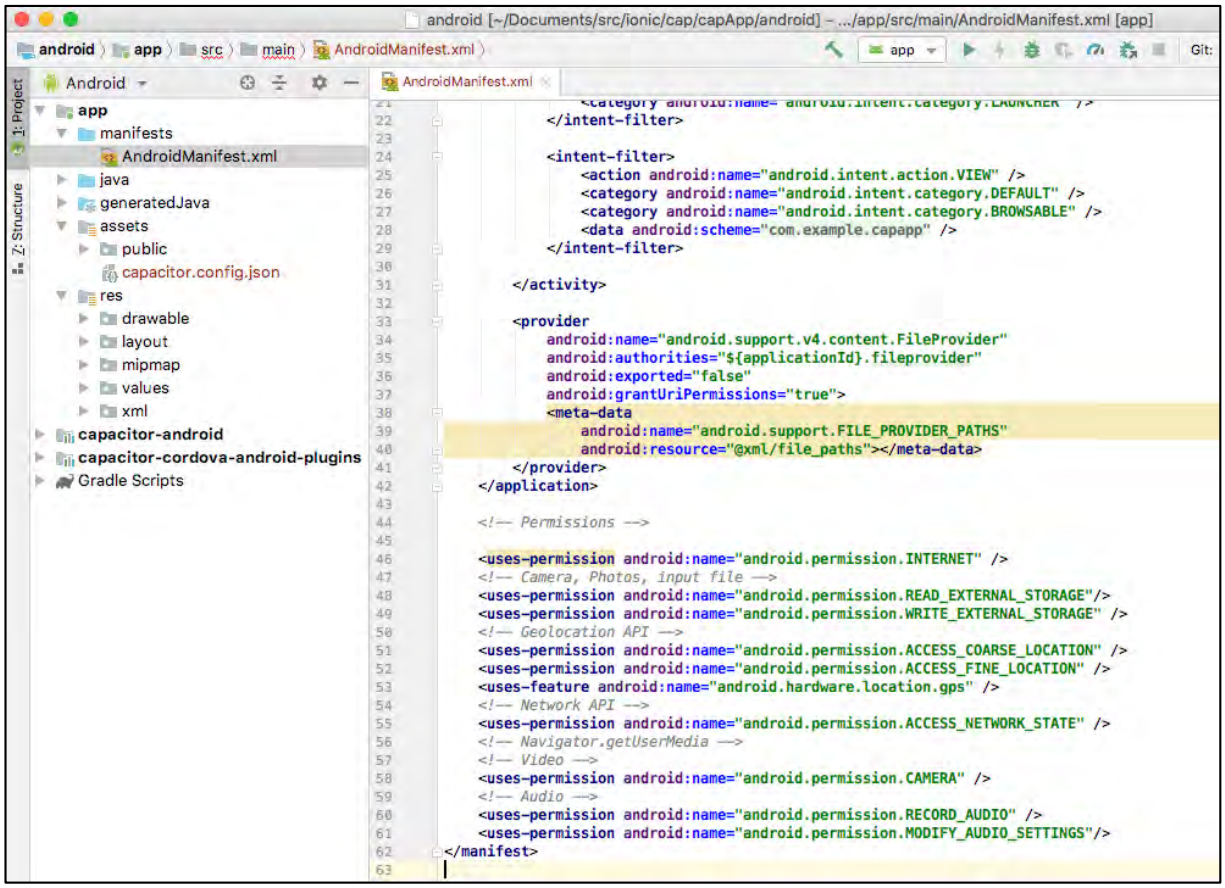

Figura 19. Habilitación de soporte del dispositivo en la aplicación Android de la galería de fotos.

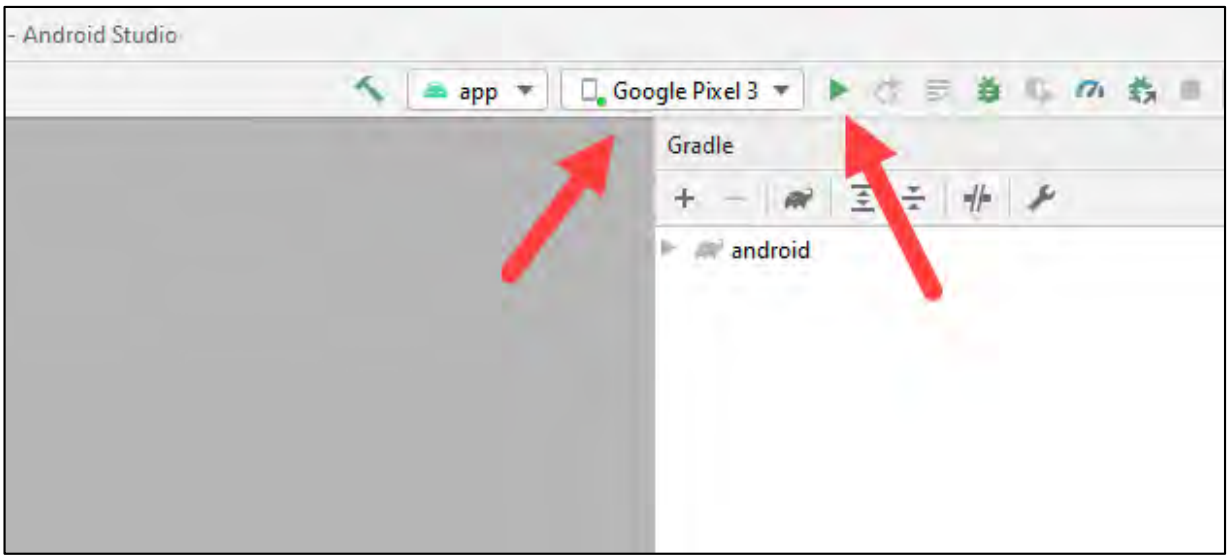

Figura 20. Selección del dispositivo Android de prueba

iniciará la aplicación en el dispositivo.

Una vez más, al pulsar el botón Cámara en la pestaña Galería de fotos, se debe mostrar la solicitud de permiso (Fig. 21). Se deberá seleccionar la opción "Aceptar", para poder tomar una fotografía con la cámara del dispositivo, la cual debería aparecer en la aplicación una vez capturada.

Ahora la aplicación Galería Fotográfica se ha instalado en dispositivos con los sistemas *Android* e *iOS*, logrando una migración exitosa de la aplicación híbrida hacia los entornos nativos.

Conclusiones

El presente trabajo muestra el desarrollo de una aplicación móvil híbrida en entorno *web*, la cual se distribuye hacia los sistemas operativos de mayor uso en el mundo (*Android* e *iOS*), desarrollo que se logra a través de los *frameworks Ionic* y *Capacitor*, mismos que se utilizan para habilitar las vistas nativas desde código HTML desplegable en el navegador, así como para establecer la comunicación entre la aplicación y el *hardware* del dispositivo de forma sencilla.

Siguiendo el esquema de construcción de una aplicación propuesto en este documento, el proceso se resume en cinco etapas de construcción que permiten: establecer los requerimientos tecnológicos (etapa 1), establecer el proyecto de *software* y los diseños iniciales de las interfaces de usuario (etapa 2), establecer las rutinas de entrada-salida de la cámara digital del dispositivo (etapa 3), la construcción de la lógica adaptable para la distribución a plataformas nativas (etapa 4), y la distribución (etapa 5) de la aplicación hacia los sistemas operativos nativos.

Tanto *Ionic* como *Capacitor*, *frameworks* utilizados en la aplicación de la galería de fotos, permiten trabajar el modelo vista controlador (MVC), utilizado en el desarrollo de aplicaciones web para adaptar este modelo de desarrollo de *software* y reutilizarlo en el desarrollo de aplicaciones móviles, permitiendo la creación de aplicaciones multiplataforma, simplificando los costos de desarrollo nativo y reutilizando el código generado para la aplicación híbrida en entornos *web*, lo que

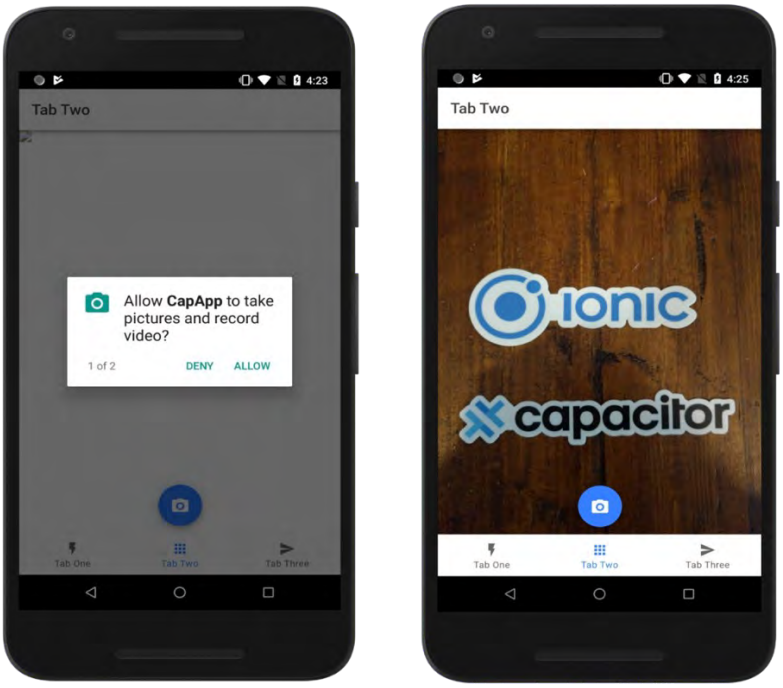

Figura 21. Prueba de funcionalidad de la aplicación de galería de fotos en el sistema operativo Android

permite desarrollar aplicaciones en un tiempo menor de programación.

Agradecimientos

Al Programa para el Desarrollo Profesional Docente, para el Tipo Superior (PRODEP), por el apoyo y las facilidades brindadas en el marco del proyecto de fortalecimiento de cuerpos académicos (UMAR-CA-38), convocatoria 2020. A los estudiantes de la Licenciatura en Informática colaboradores del cuerpo académico de Sistemas Inteligentes (UMAR-CA-38), por su esfuerzo y dedicación en la elaboración del presente trabajo.

Referencias

Angulo, R. 2013. Aplicaciones móviles híbridas: lo mejor de dos mundos. DEBATES IESA. XVIII(1): 78-79.

Basalo, M. A. 2014. Que es AngularJS. Consultado el 7 de julio de 2021: http://www.desarrolloweb.com/articulos/que-es-angularjs-descripcion-framework-javascript-conceptos.html

Branstein, M., & B. Nick. 2017. The nativescript book. Brosteins, Louisville, Kentucky, 462 pp.

Denko, B., S. Pecnik & I. Fister. 2021. A Comprehensive Comparison of Hybrid Mobile Application Development Frameworks. International Journal of Security and Privacy in Pervasive Computing. 13(1):78-90.

Eisenman, B. 2017. Learning React Native. O'Reilly Media, Sebastopol, California, 242 pp.

Ionic Drifty, C. 2020. The Ionic Framework. Consultado 2 de julio de 2021: http://ionicframework.com

Flutter. 2021. Flutter architectural overview. Consultado el 27 de junio de 2021: https://flutter.dev/docs/resources/technical-overview

Griffith, C. 2017. Mobile App Development with Ionic, Revised Edition. O'Reilly Media, Sebastopol, California, 292 pp.

Lopez-Vásquez, A. S., Delgado-Orta, J. F., Ochoa-Somuano, J., Cruz-Maldonado, O. A., Ayala-Zúñiga, A. A., Menéndez-Ortiz, M. A., Martínez-Ruíz, J. M., Perales-Ambrosio, I., Pacheco-De la Paz, K. C. & Reyes-Barrera, M. P. 2022. Aplicaciones para dispositivos móviles utilizando frameworks de software libre: caso de estudio IONIC y CAPACITOR. Revista Ciencia y Mar

26 (77): 1-10.

Masiello, E. & J. Friedmann. 2017. Mastering React Native leverage frontend development skills to build impressive iOS and Android applications with Native React. Packt Publishing, Birmingham, UK, 496 pp.

Morillo, J. 2014. Entornos de programación móviles. Consultado el 22 de junio de 2021: https://www.exabyteinformatica.com/uoc/Informatica/Tecnologia_y_desarrollo_en_dispositivos_moviles/Tecnologia_y_desarrollo_en_dispositivos_movil es_(Modulo_3).pdf

NativeScript. 2020. Native mobile apps with Angular, Vue.js, TypeScript, JavaScript -

NativeScript. Consultado el 25 de junio de 2021: https://www.nativescript.org/

Peña, I. 2017. Hybrid development platforms. Proyecto Fin de Carrera / Trabajo Fin de Grado, E.T.S.I. de Sistemas Informáticos (UPM), Madrid.

Puvvala, A., A. Dutta, R. Roy & P. Seetharaman. 2016. Mobile Application Developers' Platform Choice Model. Proceedings 49th Hawaii International Conference on System Sciences 2016, p: 5721-5730.

Ravulavaru, A. 2017. Learning Ionic 2, Second Edition. Packt Publishing, Birmingham, UK, 378 pp.

StatCounter. 2020. Mobile Operating System Market Share Worldwide. Consultado el 28 de junio de 2021: https://gs.statcounter.com/os-market-share/mobile/worldwide

Tracy, K.W. 2012. Mobile Application Development Experiences on Apple's iOS and Android OS. Potentials, IEEE. 31(4): 30-34.