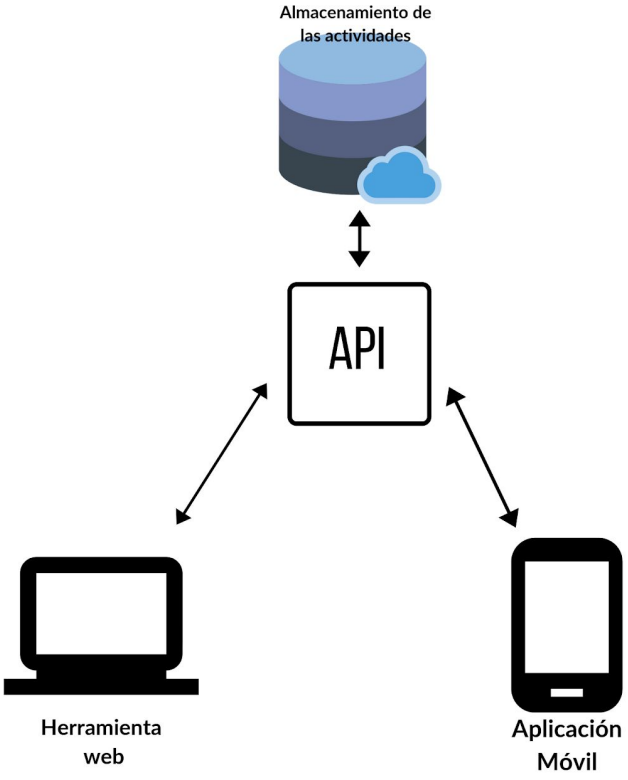
Trabajo Final Orientación a Objetos 1 y 2

Autores

Trabajo Orientación a Objetos 1: Mozzon Federico
Trabajo Orientación a Objetos 2: Butera Blas, Dal Bianco Pedro

Trabajo realizado

Se desarrolló una aplicación móvil configurable y genérica la cual permite descargar desde una API actividades educativas basadas en posicionamiento que se hayan generado previamente a través de una herramienta web y luego llevarlas a cabo a través de la misma aplicación (***Figura 1***). Es posible tener descargadas en la aplicación móvil múltiples actividades y ejecutar cada actividad, de manera independiente entre sí, repetidas veces.



La herramienta web permite generar una actividad educativa configurable, compuesta por distintas tareas para las cuales se especifica el tipo de tarea a desarrollar e información específica del tipo de tarea, además de la forma en que se realizarán las tareas.
La aplicación móvil permite descargar las actividades generadas por la herramienta web y llevarlas a cabo completando las distintas ***tareas***, las cuales pueden ser:

- ***Tareas de opción múltiple***: se define una consigna y luego se brindan distintas opciones, de las cuales sólo una es correcta.

- ***Tareas de respuesta libre***: se define una consigna y el usuario ingresa una respuesta para dicha consigna, luego, la aplicación le muestra la respuesta esperada.

La actividad está pensada para ser llevada a cabo en un espacio físico donde se distribuyen sus tareas y se pueda acceder a cada una de ellas cuando se alcance su respectiva posición. Para este desarrollo se decidió que para detectar que quien está realizando la actividad alcanzó la ubicación de una tarea, éste debe escanear un código QR correspondiente a dicha tarea.

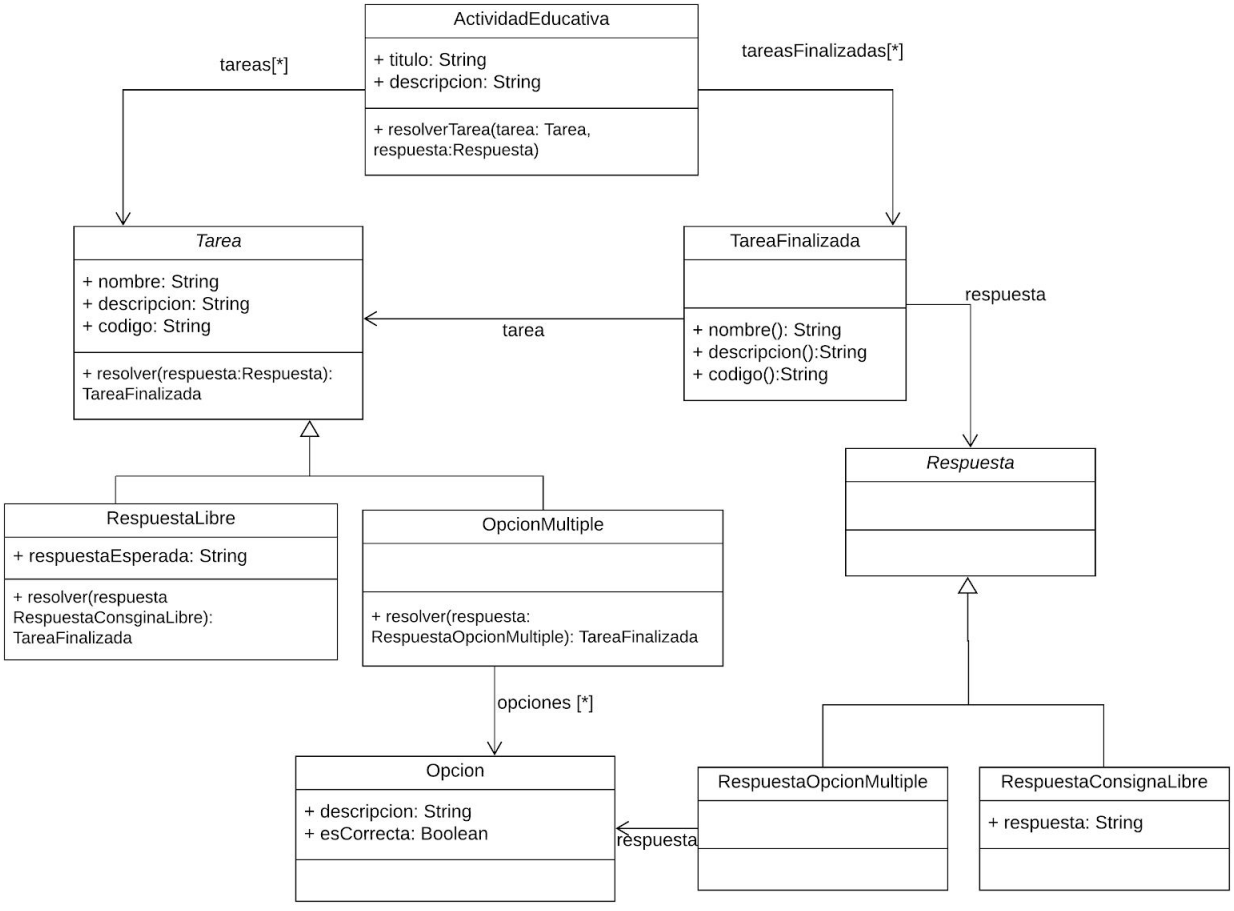
En este trabajo se desarrolla lo correspondiente a la aplicación móvil, y conceptos de las materias Orientación a objetos 1 y 2, el desarrollo completo del trabajo se puede encontrar en este artículo.
El código del desarrollo documentado en este informe se encuentra disponible en este repositorio.

Objetos 1 conceptos aplicados

Introducción

JavaScript es un lenguaje de programación **orientado a objetos** pero a diferencia del visto durante la materia es un lenguaje de objetos basado en **prototipos**, no en clases.
La programación basada en prototipos es un estilo de programación orientada a objetos donde los objetos no se crean a partir de clases sino a través de la clonación de otras instancias.

Objetos



En primer lugar se desarrolló un diagrama UML con los objetos que componen la aplicación. **Actividad** es el responsable principal de la aplicación, éste posee un nombre que será el que mostrará la aplicación móvil, un descripción, indicando cuál es el objetivo de la actividad a desarrollar. Tareas, que son las que aún les estudiantes no han desarrollado, estas **Tareas** pueden estar conformadas por un grupo heterogéneo de tareas donde cada una de ellas puede ser de un diferente tipo, como se ha mencionado anteriormente, puede ser del tipo de **Opción Múltiple**, donde a su vez tienen ligado una colección de opciones.
El objeto opción posee una descripción, es decir la información que propone esa opción y un boolean para verificar si era o no correcto para esta tarea.
La tarea también puede ser una **Respuesta Libre**, la cual sólo posee una respuesta esperada como las respuestas que puede dar un usuario son muy diversas, se utiliza la respuesta esperada a modo de dar una devolución sobre la respuesta que ingresó y que utilice como guía para corroborar su respuesta.
Existe el objeto **Tarea Finalizada**, el cual es una tarea, que agrega una **Respuesta**, la tarea es un objeto a forma de señalar que la respuesta podría cambiar dependiendo del caso de cada tarea, por ejemplo para el caso de respuesta libre, simplemente sería un String pero en las tareas de opción múltiple será un objeto opción. Esto es pensado teniendo en cuenta que el objetivo de la aplicación es poder extender los tipos de tarea lo más que se pueda. Teniendo esto en cuenta creemos que es más conveniente crear una herencia de respuesta ya que eventualmente una respuesta podría complejizarse e involucrar más objetos aún no creados.

Polimorfismo y Herencia

En el caso mencionado de la tarea, que pueden existir distintos tipos de tareas las cuales tengan información y comportamiento similar es un buen caso para usar **herencia**, es decir agrupar el comportamiento similar en un objeto tarea y dejar que cada una de los diferentes tipos de tarea, ya sea de respuesta libre o de opción múltiple redefina el comportamiento y tenga nuevos atributos. También vemos el uso de herencia en el caso de una respuesta, donde tenemos una respuesta propia para cada tipo de tarea, se vuelve a aplicar herencia por lo mencionado anteriormente del reuso y pensando en la extensión tanto de las tareas como de sus respuestas
Por ejemplo, una tarea va a tener una consigna y una respuesta pero es propio de cada tarea en particular ver cómo será la respuesta, en el caso de la opción múltiple, será sólo un string con una respuesta esperada mientras que en la de opciones múltiples la respuesta podría ser una colección de opciones, es decir que hay varias opciones correctas, esto nos deja lugar para extender las tareas a futuro, en el caso de que apareciese un nuevo tipo de tarea, por ejemplo de tomar una foto que cumpla con una consigna, sea fácil de extender.
También, dado que cada tarea podrá resolverse por sí misma vemos un caso donde se puede aplicar polimorfismo. Los distintos tipos de tarea, como poseen distintos tipos de atributos de respuesta, variará su forma de corregirse, entonces aplicamos polimorfismo para ver si cada tarea fue resuelta correctamente y devolver si fue resuelta de manera adecuada o no, según su criterio de corrección.

Objetos 2 conceptos aplicados

Frameworks

Para el desarrollo de este proyecto se utilizó el framework ***Expo***[1], que a su vez está montado sobre el framework ***React Native***[2], y brinda herramientas para facilitar el desarrollo de aplicaciones móviles con este último.

React native

React Native es un framework JavaScript para diseñar aplicaciones nativas tanto para dispositivos Android o IOs mayor a 10.0.
Es un framework basado en componentes, es decir que cada elemento visual de una aplicación *React Native* será un componente, es decir una clase que herede de *React.Component*[3] y nos permite redefinir, entre otros, los siguientes métodos:

- **constructor()**

- **render()** es el unico metodo que se debe redefinir obligatoriamente, y debe devolver un *JSX*[4] que describa cómo será la interfaz de usuario del componente.

- **componentDidMount()** se invoca inmediatamente después de que un componente se monte (se inserte en el árbol). Si se necesitara cargar datos desde un *endpoint* remoto, este es un buen lugar para instanciar la solicitud de red.[5]

- **componentWillUnmount()** se invoca inmediatamente antes de desmontar y destruir un componente. Realiza las tareas de limpieza necesarias en este método, como la invalidación de temporizadores, la cancelación de solicitudes de red o la eliminación de las suscripciones que se crearon en *componentDidMount()*.[6]

- **componentDidUpdate()** se invoca inmediatamente después de que la actualización ocurra. Este método no es llamado para el renderizador inicial.[7]

Aquí se puede observar el uso de ***Template Method*** para acceder a los *hotspots* del frameworks, tal como se vio en la materia.

Expo

Es un framework que funciona sobre *React native* que brinda un conjunto de herramientas para facilitar el desarrollo de aplicaciones. Entre estas se utilizaron la interfaz de línea de comandos *expo-cli*, para inicializar la estructura de carpetas de la aplicación, con el archivo

[1] https://expo.io/learn
[2] https://facebook.github.io/react-native/
[3] https://es.reactjs.org/docs/react-component.html
[4] https://es.reactjs.org/docs/introducing-jsx.html
[5] Definición extraída de la documentación de React
[6] Definición extraída de la documentación de React
[7] Definición extraída de la documentación de React

App.js como componente raíz y los módulos *expo-permission* para el manejo de permisos y *expo-barcode-scanner* para la lectura de códigos QR.

Patrones

Redux

Para el manejo del estado de la aplicación se utilizó el patrón de diseño *Redux*[8], a través de la librería *redux.js*.
Este patrón define:

- La existencia de un único estado global para toda la aplicación con la forma de un objeto plano *javascript*.

- Que dicho estado debe ser de solo lectura, y solo puede ser modificado a través de la emisión de una *'acción'*, objetos planos que indican que es lo que sucedió en su propiedad de tipo, y que pueden contener datos adicionales. Esto garantiza que ningún componente va a poder modificar el estado directamente, y se puede generar una historia de la aplicación a través de las *acciones* que fueron emitidas.

- Los cambios a este estado se realizan con funciones puras, que toman el estado previo y una acción, y devuelven el nuevo estado. Estas funciones puras se llaman *reducers*.

Es importante mencionar entonces que para adaptar la aplicación a este patrón de diseño se migró del modelo original planteado a un estado con objetos *javascript* planos.
En este proyecto se definieron dos módulos, uno con la lógica correspondiente a una actividad, y otro correspondiente a una tarea en particular, cada uno con sus respectivas acciones y reducers correspondientes.
Las *acciones* se encuentran en la carpeta *'src/actions'* y allí se definen las distintas formas de interactuar con el estado de la aplicación, entre las que se encuentran establecer una nueva actividad, resolver una tarea, establecer un nuevo código de tarea escaneado, entre otras.
Los *reducers* se encuentran en la carpeta *'src/reducers'*, donde se define como se modifica el estado en función de cada acción definida anteriormente.

Presentational and Container Components[9]

Este patrón sugiere dividir los componentes en dos categorías: Componentes de contenedor y componentes de presentación.

Los **componentes de presentación** son componentes principalmente ligados a la interfaz de usuario, que casi no contienen una lógica ni estado propios más allá de cuestiones relacionadas con la interfaz gráfica.
Estos reciben los datos y *callbacks* a través de las *props* y no especifican cómo se cargan o mutan los datos, y suelen escribirse en forma de una función que recibe *props* y devuelve un JSX.

[8] https://redux.js.org/
[9] https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

En este proyecto, se encuentran en la carpeta *'src/components'*, y algunos ejemplos son el encargado de mostrar cómo se renderiza una tarea de opción múltiple (*src/components/multipleChoiceComponent/multipleChoiceComponent.js*), el encargado de renderizar una tarea de respuesta libre (*src/components/freeAnswerComponent.js*), o incluso algunos más sencillos como el de cada opción a seleccionar en una tarea de opción múltiple.

Los **componentes de contenedor** son quienes contienen la lógica de la aplicación, y suelen representar pantallas o secciones completas de esta.
Pueden contener a su vez componentes de presentación u otros componentes de contenedor y son los encargados de conectarse con el estado y las acciones de la aplicación y enviarselos a los componentes de presentación en forma de datos o *callbacks*.
Para esto suelen generarse a través de componentes de alto orden (definidos en el inciso siguiente), cómo *connect()*, de la librería *'react-redux'*, que le permite acceder al estado y las acciones de la aplicación.
En este proyecto los componentes de contenedor se encuentran en la carpeta *'src/containers'*, y en este caso se corresponden con las distintas pantallas de la aplicación, siendo por ejemplo *'WelcomeScreen.js'* la pantalla de bienvenida, *'HomeScreen.js'* la pantalla que principal cuando se está realizando una actividad, entre otras.

Componentes de alto orden

El patrón de diseño de **componentes de alto orden**[10] (higher order components o HOC) sirve para compartir datos o funcionalidad entre distintos componentes evitando la repetición de código.
Esto se logra creando un componente que recibirá un componente como parámetro, le inyectara los datos o funcionalidad deseada a través de sus *props*, y en su método *render()* devuelve este mismo componente con las propiedades inyectadas, de ahí el nombre de componente de alto orden.
En este proyecto, si bien no se definieron componentes de alto orden, si se utilizaron varios a los que se les pasaron componentes de contenedor como parámetro.
Este es el caso de *connect()*, componente de alto orden provisto por la librería *'react-redux'*, que le permite a un componente acceder al estado de la aplicación y disparar acciones. Para esto se deben definir las funciones *'mapStateToProps'* y *'mapDispatchToProps'* que indican de qué forma se va a acceder al estado y a las acciones desde las props del componente pasado como parámetro. Un ejemplo del uso de este **HOC** se ve en el archivo *'src/containers/MainScreen.js'*, entre otros.
Otro ejemplo de componentes de alto orden utilizados en este proyecto son el uso de *withNavigationFocus()*, de la librería *'react-navigation'*, que le permite al componente pasado como parámetro acceder a la prop *isFocused*, que indica si en ese momento se está haciendo foco en ese componente.

10 https://medium.com/@pramonowang/advanced-react-component-patterns-higher-order-component-hoc-9d4d9077b05f

Testing

En una aplicación React que utiliza Redux se pueden realizar tests de unidad sobre los componentes, las acciones y los reducers, para esto se utilizó *jest*[11], la principal librería para realizar este tipo de tests en *javascript*.

Tests sobre *Redux*

Una de las ventajas que encontramos al realizar tests habiendo utilizado el patrón *redux* fue que al estar definidas a través de acciones todas las formas de acceder al estado, se dispone de un buen punto de partida para cubrir gran parte de la lógica de la aplicación realizando tests sobre todas las acciones, y sobre el reducer aplicado a cada una de dichas acciones.
Un ejemplo de test de unidad de una acción sobre un reducer tiene la siguiente forma:

- Definidos un estado previo y una tarea:

```
const state = {
 ready:true,
 title:'Actividad nueva',
 description:'Descripción nueva',
 tasks:[
   {
     name:'Tarea 1',
     description:'Descripción de la tarea 1',
     code:'t1'
   },
   {
     name:'Tarea 2',
     description:'Descripción de la tarea 2',
     code:'t2'
   }
 ],
 finishedTasks:[]
}

const task = {
 name:'Tarea',
 description:'Descripción de la tarea',
 code:'t1',
}
```

[11] https://jestjs.io/

- Se testea el reducer sobre el estado *state* y la acción *solveTask* que, dada en este caso la tarea *t1* y una respuesta a dicha tarea, debería mover la tarea *t1* de la lista *tasks* a la lista *finishedTasks*, con su respuesta correspondiente, como se muestra en la figura siguiente:

```
it('Debería mover una tarea a resuelta', () => {
   expect(
     reducer(state, solveTask(task, 'Una respuesta'))
   ).toEqual({
     ...state,
     tasks:[
       {
         name:'Tarea 2',
         description:'Descripción de la tarea 2',
         code:'t2'
       }
     ],
     finishedTasks: [
       {
         answer: 'Una respuesta',
         task: {
           name:'Tarea',
           description:'Descripción de la tarea',
           code:'t1',
         }
       }
     ]
   })
 })
```

En este proyecto, este y otros tests se encuentran en los archivos dentro de la carpeta *'src/tests/redux'*.

Snapshot tests

La librería *Jest* nos permite también escribir **snapshot tests**, un tipo de test orientado probar que la UI no cambia de forma inapropiada.
Un snapshot test con *Jest* en React permite obtener el **JSX** generado al renderizar un componente con determinadas *props* y testear que se corresponda con el esperado.

Utilizando esta metodología se realizaron tests sobre los componentes *'freeAnswerComponent.js'* y *'multipleChoiceComponent.js'*, con dos tareas de ejemplo.
Estos tests se encuentran en la carpeta *'src/tests/snapshots'*, y los **JSX** generados dentro de la carpeta *'/__ snapshots __'*.