Práctica 1 - React.js - IWEB

Octubre 2019

Introducción

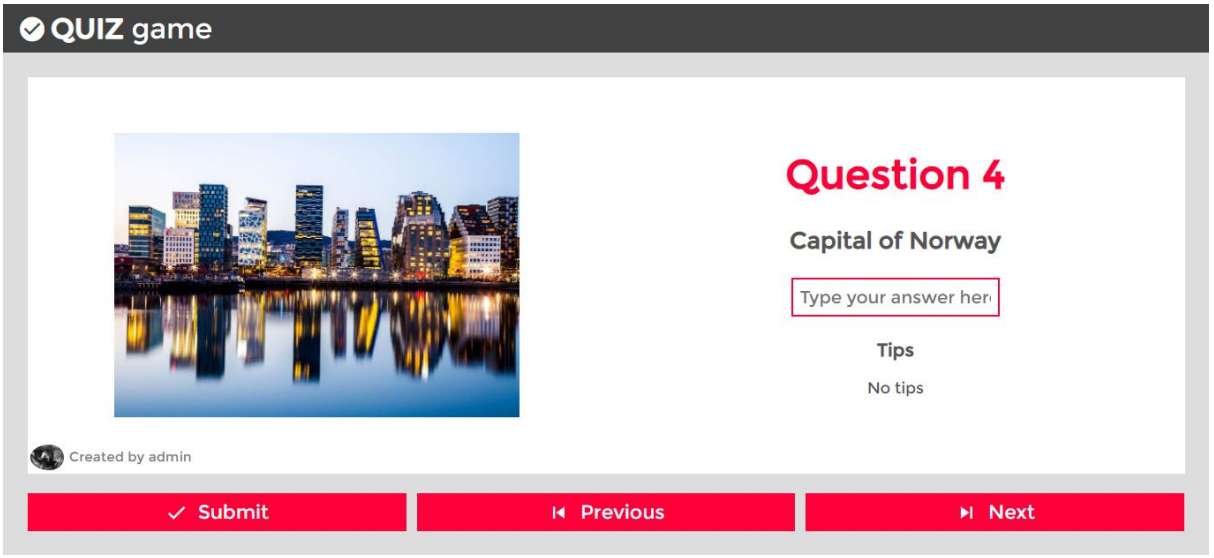
Requisitos:

- Nociones básicas de las tecnologías explicadas en clase
- Editor como Sublime Text, Atom o Webstorm (con correo UPM un año de licencia).
- Tener instalado node.js (10+), git y yarn
- Recomendable: React DevTools en el navegador

El objetivo de esta práctica es realizar un juego de preguntas utilizando las tecnologías vistas en clase. Para ello se desarrollará una Single Page Application que permita jugar al juego creado en el Quiz de la asignatura de CORE.

El juego consistirá en contestar 10 preguntas aleatorias procedentes del juego de Quiz. Para ello, es necesario descargarse las preguntas y sus respuestas de un servidor web similar al realizado en CORE. Se mostrarán sucesivamente las distintas preguntas al usuario, que podrá ir contestando a cada una de ellas. Al terminar, el usuario pulsará el botón de “Submit” para evaluar sus respuestas y obtener su porcentaje de aciertos. La evaluación de las respuestas se realizará en la propia aplicación web, no en el servidor.

En la siguiente imagen se muestra un ejemplo de esta aplicación:



Primero se recomienda programar la aplicación con datos estáticos (mock data) y después modificar el código para bajarse los datos actualizados del servidor. Hay un ejemplo de

datos de mock en Moodle (`mock-data.js`). Estos datos son un extracto de los datos provenientes de :

https://quiz.dit.upm.es/api/quizzes/random10wa?token=0123456789

Nota: Es necesario sustituir el token por el correspondiente a cada pareja de laboratorio. Puede consultarse el token en https://quiz.dit.upm.es/, iniciando sesión con las credenciales creadas para cada pareja y accediendo al perfil de usuario en la esquina superior derecha de la página.

Primeros pasos: Preparación del entorno de desarrollo

Tradicionalmente los desarrollos de Frontend manejaban ficheros estáticos (HTML, CSS, JS...). En la actualidad, los desarrollos son cada vez más complejos; ya no son simples páginas que muestran información al usuario, sino que son webapps interactivas. Para facilitar el desarrollo de estas webapps, han surgido numerosos frameworks y librerías diferentes, como Angular o React. Estas tecnologías añaden un montón de nuevas funcionalidades al desarrollo web tradicional. Una de ellas es el uso de servidores de desarrollo, como Webpack, que nos permite inyectar el CSS directamente en JavaScript, transpilar nuestro código con Babel cada vez que hacemos un cambio en nuestros ficheros, y actualizar estos cambios en el navegador sin necesidad de refrescar la página. Cuando el desarrollo de termina, Webpack nos permite empaquetar y transformar la webapp en los ficheros HTML y JS a los que estamos acostumbrados.

Adicionalmente, la tecnología React nos proporciona una herramienta de CLI, denominada Create React App, para facilitar las tareas de desarrollo y empaquetamiento del código. Además, facilita la configuración inicial que se debe llevar a cabo cada vez que se inicia un proyecto y abstrae la complejidad introducida al añadir Webpack, Babel y otras herramientas necesarias para el desarrollo.

Pasos a seguir:

1. Instala node.js (10+) y git sino no has hecho ya
2. Instala yarn
3. Instala Create React App: `yarn add global create-react-app`
4. Comienza un nuevo proyecto: `yarn create react-app my-app`
5. Entra en el directorio creado: `cd my-app`
6. Inicia el servidor de desarrollo: `yarn start`

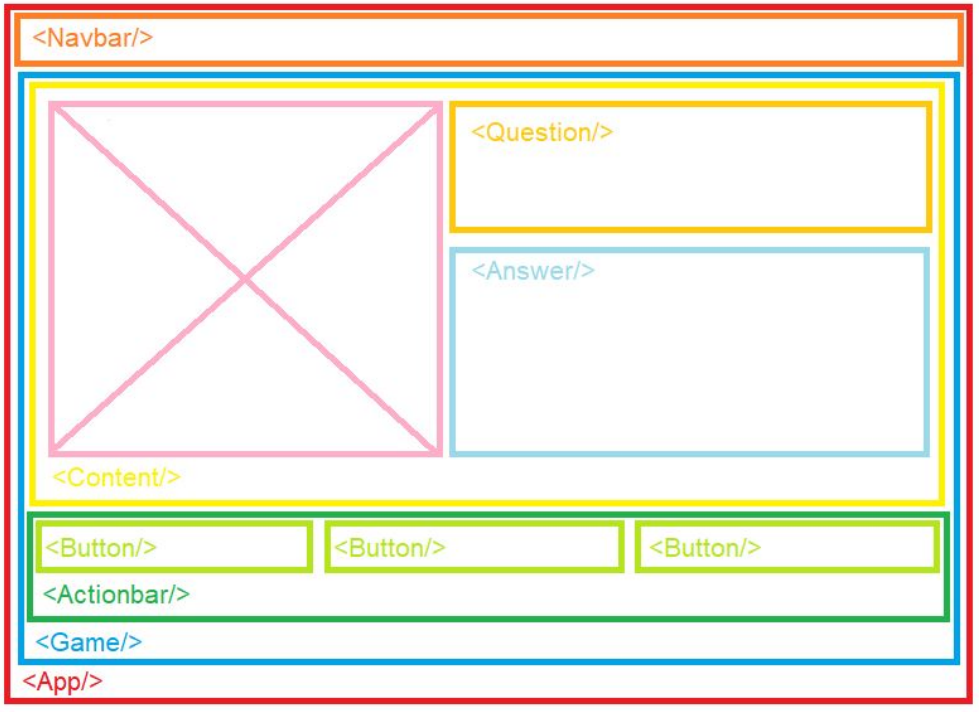
A partir de ahora si abrimos el navegador en la URL http://localhost:3000, podemos visualizar la aplicación en tiempo real mientras desarrollamos. Cuando queramos parar el desarrollo basta con hacer `Control + C` en el terminal.

Al utilizar Create React App se crea una estructura de directorios en la carpeta `my-app`, que contiene todos los ficheros necesarios para comenzar el desarrollo. Para entender mejor cómo funciona el servidor de desarrollo, observa el contenido de `public/index.html`. Se trata de la página web que renderizará nuestra aplicación de React. Como ves, apenas tiene contenido; sólo tiene un elemento `div` con id `root`.

En la carpeta `src` se encuentran todos los componentes de React del proyecto. El fichero `src/index.js` es el que sirve de puente entre React y el HTML de la página. En él, se renderiza el componente `App` (el componente principal de nuestra aplicación) en el elemento con `id="root"` de `index.html`, y dentro de él, todos los componentes que se definan.

Comenzando a desarrollar

Antes de empezar a programar, uno debe pensar cómo estructurar la aplicación, y decidir *grosso modo* qué componentes va a necesitar y la jerarquía de los mismos. Un posible diseño es el siguiente (se valorará la originalidad del alumno al proponer el suyo propio):



No hace falta que el diseño sea el definitivo, se puede ir cambiando durante el desarrollo.

Integrar Redux

Para manejar el estado y la lógica de nuestra aplicación vamos a utilizar Redux. Primero vamos a instalar Redux y configurar el entorno para poder emplearlo. Desde el terminal ejecutamos:

```
yarn add redux react-redux
```

En el directorio `src` creamos una nueva carpeta llamada `redux`, donde situaremos todos los archivos necesarios para utilizar esta librería. En ella, creamos un nuevo fichero llamado `ReduxProvider.js`, que servirá de puente entre React y Redux, con el siguiente código[1].

```
import { Provider } from 'react-redux';
import GlobalState from './reducers';
import { createStore } from 'redux';
```

[1] Se recomienda no copiar y pegar código del enunciado ya que en ocasiones los ficheros PDF introducen caracteres invisibles que dan problemas a la hora de transpilar.

```
import React from 'react';
import App from '../App';

export default class ReduxProvider extends React.Component {
  constructor(props) {
    super(props);
    this.initialState = {   };
    this.store = this.configureStore();
  }

  render() {

    return (
        <Provider store={ this.store }>
          <div style={{ height: '100%' }} >
            <App store={ this.store } />
          </div>
        </Provider>
    );
  }

  configureStore() {
    return createStore(GlobalState, this.initialState);

  }
}
```

Este código inicializa un *store* de Redux con el estado inicial indicado (por ahora un objeto vacío). Ahora el componente principal de nuestra aplicación (`App`), estará interceptado por `ReduxProvider`, que le pasará el estado a `App` como *props*. Para conseguir esto, debemos modificar el código de `App` para hacer uso de la función `mapStateToProps` de Redux:

```
import React, { Component } from 'react';
import './App.css';
import { connect } from 'react-redux';

function App() {
    return (
     ……
    );

}
function mapStateToProps(state) {
  return {
    ...state
  };
}

export default connect(mapStateToProps)(App);
```

Como hemos enganchado nuestra aplicación con Redux, el componente que renderiza `index.js` ya no es `App`, sino `ReduxProvider`

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import ReduxProvider from './redux/ReduxProvider';
import * as serviceWorker from './serviceWorker';

ReactDOM.render(<ReduxProvider />, document.getElementById('root'));
```

A continuación debemos crear el fichero `reducers.js` (dentro de la carpeta `redux`), donde implementaremos la lógica de la aplicación. En este fichero asociaremos cada parte del estado a la función que implementa la lógica para actualizarla. Por ahora sólo necesitamos el esqueleto del fichero:

```
import { combineReducers } from 'redux';

const GlobalState = (combineReducers({

}));

export default GlobalState;
```

Por último, creamos el archivo `actions.js` (también dentro de la carpeta `redux`), en el que definiremos las distintas acciones que permiten modificar el estado. Por ahora, lo podemos dejar vacío.

Estado de la aplicación

Una vez creados los ficheros necesarios para emplear Redux, ya podemos definir el estado de nuestra aplicación. Esto lo haremos en `ReduxProvider`, en el constructor, dentro de `this.initialState`. Dadas las características del juego, la parte principal del estado será la lista de preguntas del Quiz (`questions`). Además, puesto que las preguntas se muestran de una en una, debemos llevar la cuenta de la pregunta que se está visualizando en cada momento (`currentQuestion`). También tenemos que saber cuando el usuario ha pulsado el botón de “Submit" y terminado el juego (`finished`) y recoger la puntuación final (`score`). Un posible estado con todos estos requisitos sería de la siguiente manera:

```
this.initialState = {
     score: 0,
     finished: false,
     currentQuestion: 0,
     questions: [

     ]
}
```

El siguiente paso es importar los datos de las preguntas del quiz a la aplicación. Como ya se ha mencionado, se comenzará con datos estáticos de ejemplo disponibles en Moodle, que almacenaremos en `src/assets/mock-data.js`. Para tener acceso a este fichero desde nuestra app, primero debemos importarlo. En la parte superior del fichero de `ReduxProvider.js` añadimos:

```
import { questions } from "../assets/mock-data";
```

Ahora ya tenemos acceso a `questions` desde nuestro componente. En el estado inicial debemos asignar el valor de `questions` a la parte del estado correspondiente:

```
questions: [ ...questions ]
```

Una vez contamos con el estado completo, debemos escribir un reducer para cada parte del mismo. En el fichero `reducers.js` creamos un reducer básico para cada una de ellas y combinamos todos los reducers para calcular el estado completo (`GlobalState`):

```
import { combineReducers } from 'redux';

function score(state = 0, action = {}) {
  switch(action.type) {
    default:
      return state;
  }
}

function finished(state = false, action = {}) {
  switch(action.type) {
    default:
      return state;
  }
}

function currentQuestion(state = 0, action = {}) {
  switch(action.type) {
    default:
      return state;
  }
}

function questions(state = [], action = {}) {
  switch(action.type) {
    default:
      return state;
  }
}

const GlobalState = (combineReducers({
    score,
    finished,
    currentQuestion,
    questions
}));

export default GlobalState;
```

Ahora podemos probar que la aplicación de React está recibiendo correctamente el estado de Redux como *props* (fíjate que es sólo *props* y no *this.props*, puesto que App ha sido definido como componente funcional y no como clase). Para ello puedes probar a hacer `console.log(props.questions)` en el componente `App` de `App.js` y ver el resultado en la consola del navegador. Primero, es necesario modificar la cabecera del componente para recibir las props como argumento. También podemos borrar ya todo el contenido por defecto del `return` de `App` para hacer sitio para los nuevos componentes.

```
function App(props) {
  console.log(props)
  return (
    <div>
    </div>
  );
}
```

Primer componente

Una vez que ya están disponibles los datos, vamos a crear nuestro primer componente personalizado. En el directorio `src` creamos un componente nuevo llamado `Game.js`. Escribimos el código básico para un componente (importación y declaración):

```
import React from 'react';
export default class Game extends React.Component {
  render() {
    return (
      <div>Mi primer componente</div>
    );
  }
}
```

¿Cómo hacemos uso de este componente? Cómo se ve en el diseño preliminar de nuestra aplicación, el "padre" de este componente es <App/>, por lo que vamos a `App.js` e importamos el componente nuevo:

```
import Game from "./Game";
```

También tenemos que modificar la función del componente `App` para renderizar `Game` donde estaban los componentes que venían por defecto:

```
function App(props) {
  return <Game/>
}
```

El componente `Game` debe renderizar la pregunta que está seleccionada. En nuestro estado, por defecto es la primera pregunta. Para poder tener acceso a ella desde `Game`, es necesario pasársela como `prop`.

```
function App(props) {
  return (
    <div className="App">
        <Game question={props.questions[props.currentQuestion]} />
    </div>
  );
}
```

En el método render de `Game` podemos mostrar, como prueba, el enunciado de la pregunta actual. Si observamos el fichero de datos de mock, comprobamos que en cada pregunta el enunciado viene definido en la clave `question` de cada objeto del array de preguntas. Para hacer uso de él desde `Game`, debemos hacerlo de la siguiente manera:

```
import React from 'react';
export default class Game extends React.Component {
  render() {
    return (
      <div>{this.props.question.question}</div>
    );
  }
}
```

En el navegador deberíamos ver el enunciado de la primera pregunta “Capital of Cyprus”.

Primera acción: Responder a una pregunta

Para que el usuario pueda responder a las preguntas, es necesario proveer un elemento en la página que le permita escribir. En `Game` añadimos un elemento HTML de tipo `input` y en cada pregunta añadiremos un nuevo campo llamado `userAnswer` donde almacenaremos la respuesta del usuario (ya hay un campo llamado `answer` que contiene la respuesta correcta). Por defecto el campo `userAnswer` estará vacío ya que no existe. Si insertamos el siguiente código en `Game`, se podrá escribir en el campo de texto que aparece en el navegador pero la respuesta del usuario no persistirá, ya que no hemos añadido ninguna función que maneje los cambios en el `input`.

```
import React from 'react';
export default class Game extends React.Component {
  render() {
    return (
      <div>
        {this.props.question.question}
        <input type="text" value={this.props.question.userAnswer || ''}/>
      </div>
    );
  }
}
```

Para poder guardar la respuesta del usuario tenemos que implementar la lógica necesaria en Redux. Para ello crearemos nuestra primera acción, del tipo `QUESTION_ANSWER`.

¿Qué argumentos (`payload`) necesita ? El objetivo de esta acción es modificar la parte del estado relativa a `questions` para que incluya en la pregunta actual la respuesta proporcionada por el usuario en el campo `userAnswer`. Por ello, esta acción necesita saber la pregunta actual y la respuesta introducida.

En el fichero `redux/actions` creamos dicha acción incluyendo como argumentos el índice de la pregunta en la que se encuentra el usuario (`index`) y la respuesta introducida (`answer`).

```
export const QUESTION_ANSWER = 'QUESTION_ANSWER';

export function questionAnswer(index, answer) {
     return { type: QUESTION_ANSWER, payload: { index, answer } };
}
```

En los reducers debemos implementar la lógica necesaria para recoger dicha acción. La única parte del estado afectada por esta acción es `questions`. En el reducer de `questions` añadimos esta acción al switch:

```
import { QUESTION_ANSWER } from './actions'

// … Resto de código

function questions(state = [], action = {}) {
  switch(action.type) {
    case QUESTION_ANSWER:
        return state.map((question,i) =>  {
          return { ...question,
                     userAnswer: action.payload.index === i ?
                                 action.payload.answer : question.userAnswer}
        })
    default:
      return state;
  }
}

// ...
```

Con este código lo que conseguimos es que si la acción es del tipo `QUESTION_ANSWER`, la parte del estado donde se almacena el array de preguntas actualizará la pregunta en la posición indicada (`action.payload.index`) para que contenga la respuesta introducida por el usuario (`action.payload.answer`). El resto de preguntas (`action.payload.index !== i`) se mantienen igual.

Ahora tenemos que conectar nuestra acción con el componente de React donde está el `input` (`Game`) para poder lanzarla a medida que el usuario escriba en él. El único componente que está conectado con Redux es `App`, por lo que tendremos que importar la acción desde ese componente y pasarle como *prop* una función a `Game` para que pueda llamarla cada vez que se modifique el `input`.

Para lanzar la acción desde `App`, la importamos primero en la parte superior del fichero `App.js`:

```
import {questionAnswer} from './redux/actions'
```

A continuación le pasamos la prop a `Game` para que pueda acceder a `questionAnswer`:

```
function App(props) {
    return (
      <div>
        <Game question={props.questions[props.currentQuestion]}
              onQuestionAnswer={(answer)=>{
                props.dispatch(questionAnswer(props.currentQuestion, answer))
              }} />
      </div>
    );
}
```

Con el método `dispatch` que provee Redux conseguimos lanzar la acción, sin olvidar pasarle como parámetros el índice de la pregunta y la respuesta que obtendremos de `Game`.

Ahora en Game ya podemos hacer uso de la función. Como la prop que le hemos pasado a Game se llama `onQuestionAnswer`, para hacer uso de ella tendremos que llamar a `this.props.onQuestionAnswer` y asegurarnos de pasarle el valor del input (`e.target.value`):

```
function App(props) {
    return (
      <div>
        <Game question={this.props.questions[this.props.currentQuestion]}
              onQuestionAnswer={(answer)=>{
                this.props.dispatch(questionAnswer(this.props.currentQuestion, answer))
              }} />
      </div>
    );
}
```

El resto de acciones que se precisan para conseguir la funcionalidad requerida son:

- **`CHANGE_QUESTION`** : `changeQuestion(index)`. Cambia la pregunta que se muestra
- **`SUBMIT`**: `submit(questions)`. Finaliza el juego y evalúa las respuestas comparando **`userAnswer`** con **`answer`** y mostrando la puntuación obtenida. También este cambia **`finished`** a `true`.
- **`INIT_QUESTIONS`**: `initQuestions(questions)`. Sobreescribe las preguntas del estado por las que se indica en el argumento **`questions`**. Esta acción se utilizará al pedir las preguntas del servidor.

Siguientes pasos

- Mostrar para cada pregunta su imagen correspondiente (`attachment`) si la tiene y sus `tips` asociados (pista: usar función *map* de JavaScript) siguiendo la jerarquía de componentes presentada al inicio del enunciado.
- Mostrar para cada pregunta el nombre del autor y su avatar (si lo tiene)
- Crear botones de “Anterior”, “Siguiente” y “Submit”.
- Lograr que los botones de “Anterior” y “Siguiente” cambien la pregunta que se muestra al hacer click en ellos. Para ello es necesario implementar la acción de `CHANGE_QUESTION` y modificar la parte del estado asociada a `currentQuestion`. Es importante prestar especial atención a los extremos: es necesario deshabilitar (usar el atributo `disabled`) el botón de “Siguiente” al llegar a la última pregunta y el de “Anterior” cuando se está en la primera. También hay que contemplar el caso de que el array de preguntas esté vacío.
- Lograr que el botón de “Submit” evalúe si las respuestas son correctas lanzando la acción `SUBMIT`. Ésta debe actualizar la parte del estado asociada a `score` (con la puntuación obtenida) y a `finished` (`true`).
- Mostrar la puntuación obtenida una vez el juego haya finalizado (en vez de las preguntas). Se recomienda usar la función `reduce` para calcular la puntuación.

- Conseguir descargar los datos desde el servidor. Para ello se debe realizar una petición asíncrona a la URL que se menciona en la introducción y asignar el resultado de la petición al estado de Redux. Será necesario crear una nueva acción llamada `INIT_QUESTIONS` que tome como argumento las preguntas devueltas por el servidor y las asigne a `questions`. La petición asíncrona se puede hacer en el componentDidMount del componente `App`. Para hacer esta petición se puede emplear fetch, AJAX (incluido en jQuery), axios, etc. Otra forma de hacerlo es mediante una acción asíncrona de Redux.

Para terminar

Una vez terminado el desarrollo, es necesario empaquetar el código en ficheros estáticos. Primero, añadimos la siguiente línea a nuestro package.json:

```
"homepage": "./",
```

Esta línea consigue que los paths dentro de nuestra aplicación final sean relativos al directorio donde se encuentre. A continuación, en el terminal, nos situamos en el directorio del proyecto y hacemos `yarn build`. A este proceso se le llama "**preparar para producción**" (*build for production*).

En el directorio `build` se crearán todos los archivos necesarios para ejecutar la webapp. Puedes comprobarlo abriendo el `index.html` de la carpeta `build` y viendo que todo funciona sin necesidad del servidor de desarrollo. Incluso si quisiéramos podríamos subirlo a un servidor de ficheros estáticos, como *neocities* y nuestra webapp seguiría funcionando.

Ficheros a entregar

El alumno debe entregar todo el código realizado en un archivo comprimido, incluida la aplicación lista para producción. **NO** incluir el directorio `node_modules`. Todas las dependencias necesarias deben estar reflejadas en el `package.json`.

Se debe incluir un fichero llamado `00datos.txt` en la raíz del fichero comprimido en el que figure el nombre de los integrantes de la pareja y el listado de las mejoras realizadas sobre la práctica básica (se explican a continuación).

> ***IMPORTANTE: **No se corregirán prácticas que no incluyan la aplicación lista para producción en el directorio `build`, tal y como se explica en el apartado anterior.**

Evaluación

La práctica se valorará sobre un máximo de 10 puntos, los cuales se podrán obtener realizando las siguientes tareas:

- **6 puntos (obligatorio)**: Correcto funcionamiento de la práctica con React y Redux con los siguientes requisitos:

    - Bajarse las preguntas del servidor.
    - Contemplar el caso de que el array de preguntas esté vacío.
    - Mostrar las preguntas en la interfaz web de una en una.
    - Mostrar la imagen asociada a la pregunta si la tiene.
    - Mostrar en una lista los tips disponibles para cada pregunta si los tiene.
    - Mostrar el nombre del usuario que ha creado la pregunta y su foto (si la tiene)
    - Cambiar de pregunta al hacer click en los botones de siguiente/atrás.
    - Permitir al usuario responder a las preguntas.
    - Permitir al usuario evaluar sus respuestas y calcular su puntuación.
    - Desarrollar la lógica de cambiar de pregunta y evaluar las respuestas utilizando Redux.
    - Empaquetar la aplicación resultante para producción utilizando las instrucciones provistas.
- **1 punto**: Personalización de los estilos (CSS, animaciones, diseño original, spinner, etc.). Sólo se valorará si se estima que el alumno ha invertido una cantidad de esfuerzo significativa.
- **1 punto**: Mostrar un índice navegable de las preguntas disponibles para poder acceder a cada pregunta directamente y no a través de los botones.
- **1 punto:** Botón de reset que pida un conjunto nuevo de preguntas al servidor y reinicie el juego.
- **1 punto:** Cuenta atrás que limite el tiempo que tiene el usuario para responder las preguntas y termine el juego si se acaba el tiempo.
- **1 punto:** Incluir react-router para añadir una página de inicio a la aplicación además de la del juego y un menú para navegar entre las distintas secciones.
- **1 punto:** Lograr que la aplicación cumpla con las recomendaciones de accesibilidad del W3C usando el plugin eslint-plugin-jsx-a11y (ya instalado con create-react-app) y la extensión SiteImprove (nivel de conformance A).
- **1 punto:** Crear una batería de tests unitarios para cada reducer con Jest (ya incluído en create-react-app).

Dudas y tutorías

Sonsoles López Pernas -sonsoles.lopez.pernas@*upm.es* - B-323
Enrique Barra Arias - *enrique.barra@upm.es* - B-323