

## DESARROLLO DE APLICACIONES WEB AVANZADO

### LABORATORIO N° 10

## Redux



<b>Alumno(s):</b>					<b>Nota</b>	
<b>Grupo:</b>			<b>Ciclo:V</b>			
<b>Criterio de Evaluación</b>	<b>Excelente (4pts)</b>	<b>Bueno (3pts)</b>	<b>Requiere mejora (2pts)</b>	<b>No accept. (0pts)</b>	<b>Puntaje Logrado</b>	
Implementar Redux						
Utiliza connect para manejo del estado						
Desarrolla aplicaciones web con stores						
Realiza con éxito lo propuesto en la tarea.						
Es puntual y redacta el informe adecuadamente						

## Laboratorio 10: Redux

### **Objetivos:**

Al finalizar el laboratorio el estudiante será capaz de:

- Entender el funcionamiento de Redux
- Desarrollar aplicaciones web con connect React-Redux
- Implementación correcta de stores

### **Seguridad:**

- Ubicar maletines y/o mochilas en el gabinete del aula de Laboratorio.
- No ingresar con líquidos, ni comida al aula de Laboratorio.
- Al culminar la sesión de laboratorio apagar correctamente la computadora y la pantalla, y ordenar las sillas utilizadas.

### **Equipos y Materiales:**

- Una computadora con:
  - Windows 7 o superior
  - VMware Workstation 10+ o VMware Player 7+
  - Conexión a la red del laboratorio
- Máquinas virtuales:
  - Windows 7 Pro 64bits Español - Plantilla
- Instalador de node.js

## Procedimiento:

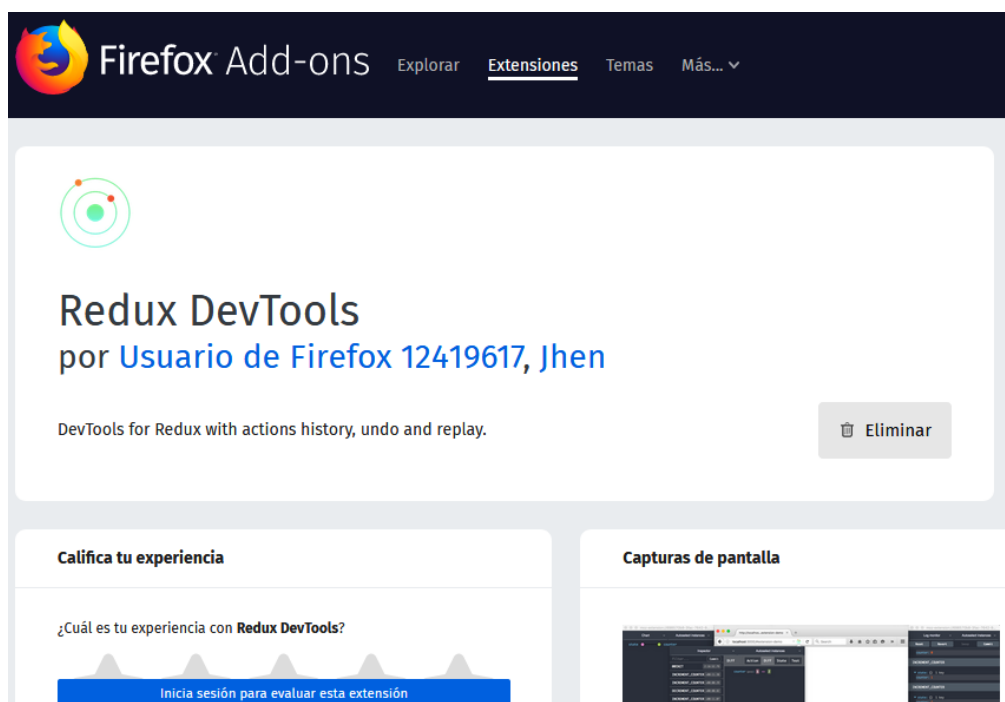
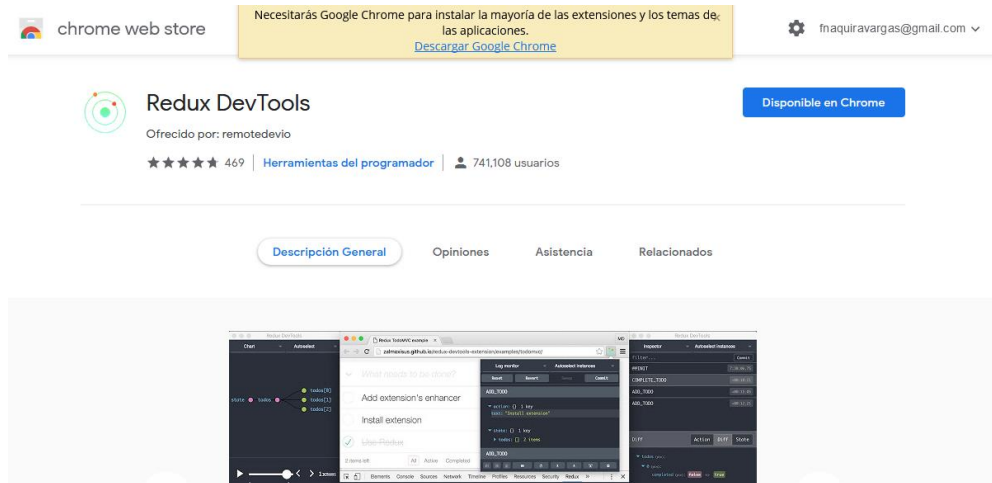
### Lab Setup

#### 1. Configuración inicial del proyecto

- 1.1. Copie el contenido de la carpeta **lab09** (el laboratorio anterior) a excepción de la carpeta **node\_modules** dentro de una carpeta llamada **lab10**
- 1.2. Instalaremos una nueva librería en nuestra carpeta lab10:

```
$ npm install --save redux react-redux redux-thunk
```

- 1.3. Ahora procederemos a instalar las Redux DevTools en el navegador que estemos utilizando. Hay opción tanto para Chrome como para Firefox.



## 2. Configuración del store y el reducer

### 2.1. Iniciaremos realizando las siguientes adiciones en index.js

```
dawa ▸ lab10 ▸ src ▸ js index.js ▸ ...  
import React from 'react';  
import ReactDOM from 'react-dom';  
import { Provider } from 'react-redux';  
import { createStore, applyMiddleware, compose } from 'redux';  
import thunk from 'redux-thunk';  
  
import 'bootstrap/dist/css/bootstrap.css';  
import './index.css';  
  
import App from './App';  
import * as serviceWorker from './serviceWorker';  
  
import rootReducer from './store/reducers/reducer';  
  
const composeEnhancers =  
  process.env.NODE_ENV === 'development'  
    ? window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__  
    : null || compose;  
  
const store = createStore(  
  rootReducer,  
  composeEnhancers(applyMiddleware(thunk))  
);  
  
ReactDOM.render(  
  <Provider store={store}>  
    <App />  
  </Provider>,  
  document.getElementById('root')  
);  
  
serviceWorker.unregister();
```

El componente Provider será el manejador de nuestro estado principal, por eso es importante renderizarlo antes incluso que nuestra App. Esto es verdad tanto en la web como en React Native.

La línea correspondiente a composeEnhancers en realidad podría ser ignorada, pero la hemos agregado para que comunique nuestro estado al plugin de navegador antes instalado.

Pasamos a la creación del store, que consiste en pasarle nuestro reductor y cualquier middleware adicional, en este caso, utilizaremos thunk, que sirve para agregar soporte asíncrono a Redux (ideal para llamadas a APIs)

### 2.2. Comenzaremos con la creación de nuestro reductor. Para esto, primero crearemos el archivo actionTypes.js dentro de la carpeta actions, dentro de la carpeta store. Este archivo contendrá el listado de acciones que podrán modificar el estado.

```
dawa ▸ lab10 ▸ src ▸ store ▸ actions ▸ js actionTypes.js ▸ ...  
export const AUTH_START = 'AUTH_START';  
export const AUTH_SUCCESS = 'AUTH_SUCCESS';  
export const AUTH_FAIL = 'AUTH_FAIL';  
export const AUTH_LOGOUT = 'AUTH_LOGOUT';
```

- 2.3. Ahora crearemos el reductor. El archivo se llamará reducer.js y se encontrará dentro de la carpeta reducers que estará dentro de la carpeta store.

```
dawa ▸ lab10 ▸ src ▸ store ▸ reducers ▸ JS reducer.js ▸ authSuccess
import * as actionTypes from '../actions/actionTypes';
import { updateObject } from '../../utils/utility';

const initialState = {
  token: null,
  email: null,
  userId: null,
  userName: null,
  error: null,
  loading: false
};

const authStart = (state, action) => {
  return updateObject(state, { error: null, loading: true });
};

const authSuccess = (state, action) => {
  return updateObject(state, {
    token: action.token,
    email: action.email,
    userName: action.userName,
    userId: action.userId,
    error: null,
    loading: false
  });
};
```

Detalles importantes a destacar, el reductor debe declarar siempre un estado inicial (en este caso, la variable initialState) el cual será modificado a través de un switch (que escribiremos a continuación).

En nuestro script, este switch invoca a funciones más pequeñas que modifican el estado. Esta es una buena práctica ya que vuelve al script más legible.

```
const authFail = (state, action) => {
  return updateObject(state, {
    error: action.error,
    loading: false
  });
};

const authLogout = (state, action) => {
  return updateObject(state, { token: null, email: null, name: null });
};

const reducer = (state = initialState, action) => {
  switch (action.type) {
    case actionTypes.AUTH_START:
      return authStart(state, action);
    case actionTypes.AUTH_SUCCESS:
      return authSuccess(state, action);
    case actionTypes.AUTH_FAIL:
      return authFail(state, action);
    case actionTypes.AUTH_LOGOUT:
      return authLogout(state, action);
    default:
      return state;
  }
};

export default reducer;
```

- 2.4. Hay que tomar en cuenta que los estados siempre deben ser inmutables, por lo que crearemos el archivo `utility.js` dentro de `utils`, con el siguiente contenido:

```
dawa ▸ lab10 ▸ src ▸ utils ▸ utility.js ▸ ...
export const updateObject = (oldObject, updatedProperties) => {
  return {
    ...oldObject,
    ...updatedProperties
  };
};
```

Aquí estamos aplicando una técnica llamada `spreading object`: consiste en traspasar cada propiedad del objeto que es invocado después de los tres puntos (`...miVariable`). De esta forma, creamos un nuevo objeto que contiene todas las propiedades del objeto antiguo y el objeto nuevo.

Con esto nuestra aplicación ya debería funcionar, pero no veremos ningún cambio aparente. Verifique el funcionamiento normal y luego empecemos a crear nuestros `actions`, que serán quienes invoquen al `reductor` para modificarlo.

### 3. Configuración del dispatcher

- 3.1. Crearemos el archivo `actions.js` dentro de la carpeta `actions`

```
dawa ▸ lab10 ▸ src ▸ store ▸ actions ▸ actions.js ▸ ...
import axios from '../utils/axios';

import * as actionTypes from './actionTypes';
import * as URLs from '../utils/urls';

export const authStart = () => {
  return {
    type: actionTypes.AUTH_START
  };
};

export const authSuccess = (token, email, name, id) => {
  return {
    type: actionTypes.AUTH_SUCCESS,
    token: token,
    email: email,
    userName: name,
    userId: id
  };
};

export const authFail = error => {
  return {
    type: actionTypes.AUTH_FAIL,
    error: error
  };
};
```

```
export const logout = () => {
  localStorage.removeItem('token');
  localStorage.removeItem('email');
  localStorage.removeItem('userId');
  localStorage.removeItem('userName');
  return {
    type: actionTypes.AUTH_LOGOUT
  };
};
```

Como se puede apreciar hasta ahora, el archivo actions.js solamente contiene funciones que devuelven parámetros y el argumento type, que a su vez contiene la palabra clave para que el reductor pueda modificar el estado. Ahora veremos cómo es el caso de un action asíncrono. Sigamos llenando nuestro archivo actions.js

```
export const auth = (username, password) => {
  return dispatch => {
    dispatch(authStart());
    axios({
      ...URLS.AUTH_LOGIN,
      data: {
        username: username,
        password: password
      }
    })
    .then(response => {
      localStorage.setItem('token', response.data.token);
      localStorage.setItem('userId', response.data.data._id);
      localStorage.setItem('userName', response.data.data.username);
      localStorage.setItem('email', response.data.data.email);
      dispatch(
        authSuccess(
          response.data.token,
          response.data.data.email,
          response.data.data.username,
          response.data.data._id
        )
      );
    })
    .catch(err => {
      dispatch(authFail(err.response.data));
    });
  });
};
```

Hemos creado nuestra llamada de inicio de sesión. Sin embargo, también necesitamos una que nos permita verificar la sesión existente en caso se refresque el navegador.

```
export const authCheckState = () => {
  return dispatch => {
    const token = localStorage.getItem('token');
    if (!token) {
      dispatch(logout());
    } else {
      const email = localStorage.getItem('email');
      const userName = localStorage.getItem('userName');
      const userId = localStorage.getItem('userId');
      dispatch(authSuccess(token, email, userName, userId));
    }
  });
};
```

3.2. Agregaremos la siguiente URL dentro del archivo urls.js

```
dawa ▸ lab10 ▸ src ▸ utils ▸ urls.js ▸ ...
export const PROFILE_EDIT = {
  method: 'put',
  url: 'user/' + localStorage.getItem('userId')
}
export const AUTH_LOGIN = {
  method: 'post',
  url: 'user/signin'
};
```

Con esto, nuestro actions.js estará listo para funcionar, procederemos a conectar nuestros componentes al estado del redux.

#### 4. Conectar componentes de React a Redux

4.1. Modificaremos el archivo Login.js para que pueda importar los siguientes archivos.

```
dawa ▸ lab10 ▸ src ▸ views ▸ JS Login.js ▸ ...  
1 import React, { Component, Fragment } from 'react';  
2 import { Form, Button } from 'react-bootstrap';  
3 import Helmet from 'react-helmet';  
4 import { connect } from 'react-redux';  
5  
6 import * as actions from '../store/actions/actions';  
7  
8 import Spinner from '../components/Spinner/Spinner';  
9  
10 class Login extends Component {  
11   state = {  
12     usuario: '',  
13     password: ''  
14   };  
15 }
```

4.2. La función connect que acabamos de importar sirve justamente para conectar tanto estados como dispatchers (invocadores de reductor) a nuestro componente. La función de realizar esta invocación es pasándole dos parámetros a dicha función: el primer parámetro es las propiedades del estado a las que quiero tener acceso, y el segundo parámetro es el dispatcher en sí. En este caso, queremos saber si el redux está cargando, si da error, o si inició sesión con éxito. Así mismo, solamente dispararemos la acción de iniciar sesión. Modificaremos la última línea del archivo y la reemplazaremos por lo siguiente:

```
const mapStateToProps = state => {  
  return {  
    loading: state.loading,  
    error: state.error,  
    isAuthenticated: state.token !== null  
  };  
};  
  
const mapDispatchToProps = dispatch => {  
  return {  
    onAuth: (email, password) => dispatch(actions.auth(email, password))  
  };  
};  
  
export default connect(  
  mapStateToProps,  
  mapDispatchToProps  
) (Login);
```

4.3. Así mismo, ya no necesitamos realizar la llamada a axios desde nuestro componente, reemplacemos el contenido de submitHandler por lo siguiente.

```
submitHandler = e => {  
  e.preventDefault();  
  this.props.onAuth(this.state.usuario, this.state.password);  
};
```



- 4.4. Utilizaremos el mismo truco del laboratorio anterior para simular una carga de componente. Copiaremos todo el contenido de nuestro form y lo pegaremos dentro de una variable content. Esta variable mostrará un spinner en caso de que el estado general sea cargando.

```
render() {
  let content = (
    <Form onSubmit={this.submitHandler}>
      <Form.Group>
        <Form.Label>Usuario</Form.Label>
        <Form.Control
          type="text"
          placeholder="Usuario para sesión"
          value={this.state.usuario}
          onChange={this.usuarioHandler}
        />
      </Form.Group>
      <Form.Group>
        <Form.Label>Contraseña</Form.Label>
        <Form.Control
          type="password"
          placeholder="123456"
          value={this.state.password}
          onChange={this.passwordHandler}
        />
      </Form.Group>
      <Button variant="primary" type="submit">
        Iniciar sesión
      </Button>
    </Form>
  );
  if (this.props.loading) content = <Spinner />;
  return (
    <Fragment>
      <Helmet>
        <style type="text/css">{`
```

- 4.5. No nos olvidemos de invocar esta nueva variable content dentro de nuestro return en donde solía estar nuestro formulario.

```
<div className="container">
  <div className="row justify-content-around">
    <div className="col-6 mt-2 mb-2">
      <div className="card">
        <div className="card-body">{content}</div>
      </div>
    </div>
  </div>
</div>
```

## 5. Rutas con permisos

- 5.1. Nuestro inicio de sesión funciona, pero debemos extender esto a toda la aplicación. Modificaremos App.js, empezando por la cabecera.

```
import Profile from './views/Profile/Profile';
import NotFound from './views/NotFound';

import { connect } from 'react-redux';
import * as actions from './store/actions/actions';

class App extends Component {
  componentDidMount() {
    this.props.onTryAutoSignup();
  }

  render() {
    return (
      <BrowserRouter>
```

- 5.2. Note que estamos invocando a una función que (por su nombre) intentará realizar una comprobación de alguna sesión existente (recuerde que esta función ya existe en nuestro actions.js) Modificaremos la última línea del mismo archivo App.js para que luzca de la siguiente manera.

```
const mapStateToProps = state => {
  return {
    isAuthenticated: state.token !== null,
    site: state.site
  };
};

const mapDispatchToProps = dispatch => {
  return {
    onTryAutoSignup: () => dispatch(actions.authCheckState()),
    onLogout: () => dispatch(actions.logout())
  };
};

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

- 5.3. Toda esta configuración realizada nos permitirá crear rutas con restricciones dependiendo de si se ha logueado el usuario o no. Modificaremos el render de nuestro archivo App.js, solamente la parte de las rutas, para que luzca de la siguiente manera.

```
<Layout>
  {this.props.isAuthenticated ? (
    <Switch>
      <Route path="/" exact component={Home} />
      <Route path="/details" component={Details} />
      <Route path="/welcome" component={Welcome} />
      <Route path="/chat" component={Chat} />
      <Route path="/profile" component={Profile} />
      <Route
        path="/logout"
        render={() => {
          this.props.onLogout();
          return <Redirect to="/login" />;
        }}
      />
      <Route component={NotFound} />
    </Switch>
  ) : (
    <Switch>
      <Route path="/" exact component={Home} />
      <Route path="/login" component={Login} />
      <Route component={NotFound} />
    </Switch>
  )}
</Layout>
```

- 5.4. Hagamos un experimento. Ahora después de estos cambios realizados, intente entrar a la vista de chat y compruebe si carga. En caso no cargue, inicie sesión e intente de nuevo. Deje sus comentarios en el archivo informe de este laboratorio.

## 6. Modificando el menú de cabecera

- 6.1. Modificaremos Header.js, agregando primero connect y luego modificando la última línea del archivo.

```
dawa ▸ lab10 ▸ src ▸ components ▸ Header ▸ JS Header.js ▸ ...

import React from 'react';
import { Link, NavLink } from 'react-router-dom';
import { Nav, Navbar, NavDropdown } from 'react-bootstrap';
import { connect } from 'react-redux';

import imgLogo from '../../assets/img/logo-tecsup.png';

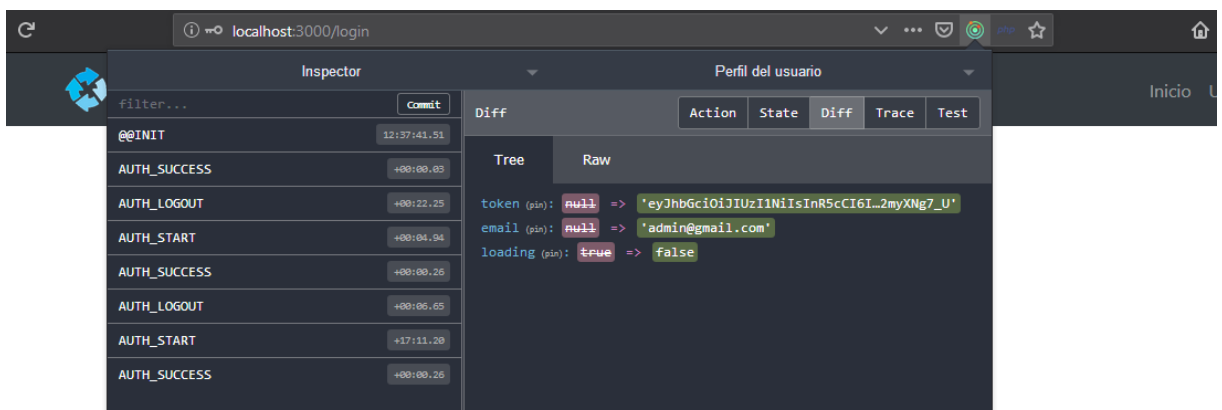
const mapStateToProps = state => {
  return {
    isAuthenticated: state.token !== null
  };
};

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

- 6.2. Como en este caso solamente queremos saber si el usuario inició sesión, no necesitaremos más que eso para luego modificar nuestro render.

```
<NavLink to="/" exact className="nav-link">
  Inicio
</NavLink>
{props.isAuthenticated ? (
  <NavDropdown title="Usuario" id="basic-nav-dropdown">
    <NavLink to="/profile" className="dropdown-item">
      Mi Perfil
    </NavLink>
    <NavLink to="/chat" className="dropdown-item">
      Chat
    </NavLink>
    <NavDropdown.Divider />
    <NavLink to="/logout" className="dropdown-item">
      Cerrar sesión
    </NavLink>
  </NavDropdown>
) : (
  <NavLink to="/login" className="nav-link">
    Iniciar sesión
  </NavLink>
)}
</Nav>
</Navbar.Collapse>
```

- 6.3. Después de este cambio, inicie y cierre sesión. Los menús se mostrarán de acuerdo a si uno está logueado o no. A todo esto, haga click en el plugin de Redux Devtools y deberá observar la historia de nuestro estado.



The screenshot shows the Redux Devtools interface. On the left, the 'Inspector' panel displays a list of actions: @@INIT, AUTH\_SUCCESS, AUTH\_LOGOUT, AUTH\_START, AUTH\_SUCCESS, AUTH\_LOGOUT, AUTH\_START, and AUTH\_SUCCESS. The 'Diff' panel on the right shows the state changes for 'token', 'email', and 'loading'. The 'Tree' panel shows the current state: token is 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpzZW50L3M4IiwiaWF0IjoxNjU0MjU0MjU0', email is 'admin@gmail.com', and loading is false.

## 7. Retoques finales

7.1. Modificaremos ProfileDetails.js, primero la importación y luego la última línea.

```

import { connect } from 'react-redux';
import getAvatar from '../utils/avatar';

const mapStateToProps = state => {
  return {
    userName: state.userName,
    email: state.email
  };
};

export default connect(mapStateToProps)(ProfileDetails);

```

7.2. Ya no necesitaremos de un estado o un componentDidMount, por lo que los eliminaremos y nuestro componente lucirá así.

```

class ProfileDetails extends Component {
  passwordHandler = e => {
    alert('Funcion por implementar!');
  };
  pictureHandler = e => {
    alert('Funcion por implementar!');
  };
  render() {
    const urlLorem = 'https://www.lipsum.com/feed/html';
    return (

```

7.3. Finalmente, modificaremos el render, reemplazando las variables que antes referenciaban al estado, ahora llamarán a las propiedades.

```

<div className="col-md-3">
  <div className="profile-img">
    <img
      src={getAvatar(this.props.email)}
      className="rounded-circle user_img"
      alt=""
    />
  </div>
</div>
<div className="col-md-7">
  <div className="profile-head">
    <h5>
      <FontAwesomeIcon icon={faUser} /> {this.props.userName}
    </h5>
    <h6>Web Developer y Diseñador</h6>

```

7.4. Así mismo, modificaremos el archivo ProfileEdit.js, primero la cabecera, luego la última línea

```

dawa ▸ lab10 ▸ src ▸ views ▸ Profile ▸ ProfileEdit ▸ ProfileEdit.js ▸ ...
import React, { Component } from 'react';
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome';
import { faSave, faWindowClose } from '@fortawesome/free-solid-svg-icons';
import { connect } from 'react-redux';

```

```
const mapStateToProps = state => {  
  return {  
    token: state.token,  
    userName: state.userName,  
    email: state.email  
  };  
};  
  
export default connect(mapStateToProps)(ProfileEdit);
```

- 7.5. Eliminaremos el componentDidMount y luego modificaremos el estado para que reciba las propiedades iniciales.

```
class ProfileEdit extends Component {  
  state = {  
    username: this.props.userName,  
    email: this.props.email,  
    loading: false  
  };  
  inputHandler = (event, field) => {  
    this.setState({ [field]: event.target.value });  
  };  
}
```

- 7.6. Finalmente, modificaremos el submitHandler para que el token sea recibido a través de redux.

```
    axios({  
      ...PROFILE_EDIT,  
      headers: {  
        Authorization: this.props.token  
      },  
      data: data  
    })
```

**6. Finalizar la sesión**

- 6.1. Apagar el equipo virtual
- 6.2. Apagar el equipo

**Conclusiones:**

Indicar las conclusiones que llegó después de los temas tratados de manera práctica en este laboratorio.