# Ejercicio: Controlled Forms

En este ejercicio, aprenderás a usar componentes controlados para diseñar formularios controlados en React.

Realizaremos un formulario controlado en el componente Contact por lo cual es requisito que este sea un componente con estado. Este formulario será para recibir feedback de los clientes.

Convierta el componente funcional Contact a un componente contenedor o con estado.

Implemente la función render() que devuelva el return del componente funcional. Debe quedar asi:

**import** React, {Component} **from 'react'**;  
**import** {*Breadcrumb*, *BreadcrumbItem*} **from "reactstrap"**;  
**import** {Link} **from "react-router-dom"**;  
  
**class** Contact **extends** Component {  
 render() {  
 **return** (  
 <**div className="container"**>

Cree un constructor para almacenar variables del formulario en el state asi:

constructor(props, context) {  
 **super**(props, context);  
 **this**.**state**={  
 **firstname**: **''**,  
 **lastname**: **''**,  
 **telnum**: **''**,  
 **email**: **''**,  
 **agree**: **false**,  
 **contactType**: **'Tel.'**,  
 **message**: **''** }  
}

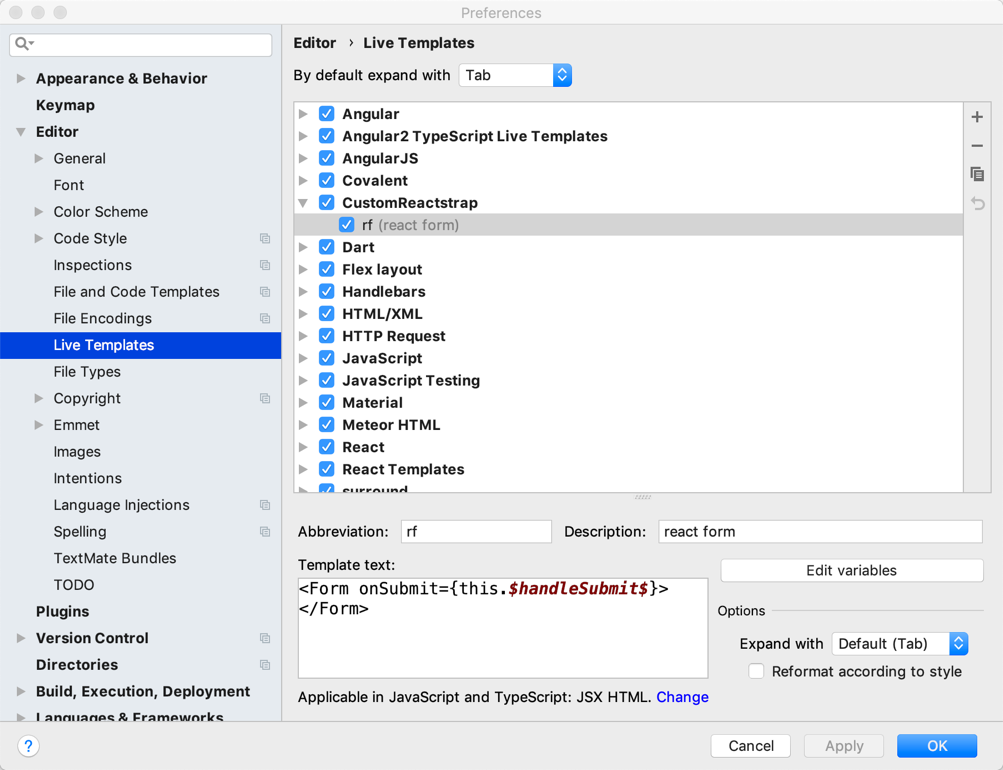
El Tag Form nos permite usar una grilla de layout en formularios, En cada FormGroup podemos poner la propiedad row y será equivalente a un div de con clase row de bootstrap.

En los elementos dentro de cada FormGroup es posible usar md={2} para especificar el tamaño de columas a ocupar en pantallas medias o superior lo cual es equivalente a col-md-2 por ejemplo. También es posible definir el numero de columnas y el offset asi md={{size: 6, offset: 2}}>

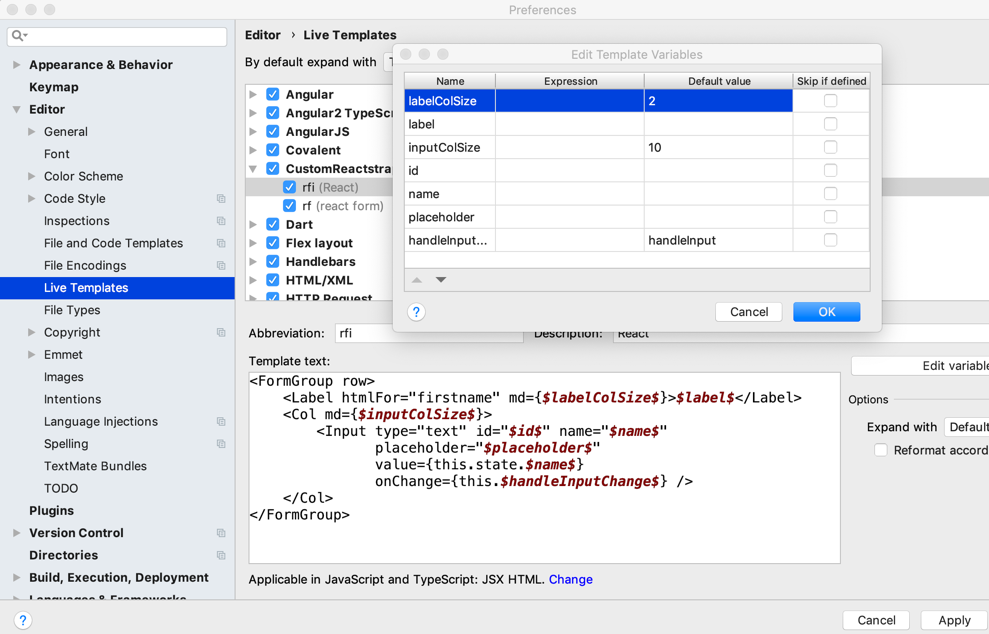
Para crear el formulario en Contact en el método render agregar un div nuevo dentro del div container asi:

<div className="row row-content">  
 <div className="col-12">  
 <h3>Send us your Feedback</h3>  
 </div>  
 <div className="col-12 col-md-9">  
 <Form onSubmit={this.handleSubmit}>  
 <FormGroup row>  
 <Label htmlFor="firstname" md={2}>First Name</Label>  
 <Col md={10}>  
 <Input type="text" id="firstname" name="firstname"  
 placeholder="First Name"  
 value={this.state.firstname}  
 onChange={this.handleInputChange} />  
 </Col>  
 </FormGroup>  
 <FormGroup row>  
 <Label htmlFor="lastname" md={2}>Last Name</Label>  
 <Col md={10}>  
 <Input type="text" id="lastname" name="lastname"  
 placeholder="Last Name"  
 value={this.state.lastname}  
 onChange={this.handleInputChange} />  
 </Col>  
 </FormGroup>  
 <FormGroup row>  
 <Label htmlFor="telnum" md={2}>Contact Tel.</Label>  
 <Col md={10}>  
 <Input type="tel" id="telnum" name="telnum"  
 placeholder="Tel. number"  
 value={this.state.telnum}  
 onChange={this.handleInputChange} />  
 </Col>  
 </FormGroup>  
 <FormGroup row>  
 <Label htmlFor="email" md={2}>Email</Label>  
 <Col md={10}>  
 <Input type="email" id="email" name="email"  
 placeholder="Email"  
 value={this.state.email}  
 onChange={this.handleInputChange} />  
 </Col>  
 </FormGroup>  
 <FormGroup row>  
 <Col md={{size: 6, offset: 2}}>  
 <FormGroup check>  
 <Label check>  
 <Input type="checkbox"  
 name="agree"  
 checked={this.state.agree}  
 onChange={this.handleInputChange} /> {' '}  
 <strong>May we contact you?</strong>  
 </Label>  
 </FormGroup>  
 </Col>  
 <Col md={{size: 3, offset: 1}}>  
 <Input type="select" name="contactType"  
 value={this.state.contactType}  
 onChange={this.handleInputChange}>  
 <option>Tel.</option>  
 <option>Email</option>  
 </Input>  
 </Col>  
 </FormGroup>  
 <FormGroup row>  
 <Label htmlFor="message" md={2}>Your Feedback</Label>  
 <Col md={10}>  
 <Input type="textarea" id="message" name="message"  
 rows="12"  
 value={this.state.message}  
 onChange={this.handleInputChange}></Input>  
 </Col>  
 </FormGroup>  
 <FormGroup row>  
 <Col md={{size: 10, offset: 2}}>  
 <Button type="submit" color="primary">  
 Send Feedback  
 </Button>  
 </Col>  
 </FormGroup>  
 </Form>  
 </div>  
</div>

Como los formularios es algo que usara frecuentemente es recomendable crear live templates en webstorm. Los live templates siguen la sintaxis de velocity:



Por ejemplo para un FormGroup:



A diferencia de frameworks como angular que tienen mecanismos provistos para sincronizar los formularios con el código JS como es el binding, react no los provee porque al no tener que comunicarse con un template html real por disponer en el código JS del dom virtual solo necesitamos reaccionar con llamadas a funciones que manejen el cambio que introduzca el usuario en los elementos de formulario.

Los elementos definidos en el formulario reaccionan en el evento html onChange y llaman a la función handleInputChange la cual implementaremos para que actualice el valor de las variables de state de acuerdo al cambio de un elemento del formulario asi:

handleInputChange(event) {  
 **const** target = event.**target**;  
 **const** value = target.**type** === **'checkbox'** ? target.**checked** : target.**value**;  
 **const** name = target.**name**;  
  
 **this**.setState({  
 [name]: value  
 });  
}

Cuando hacemos submit del formulario el botón llama a la función handleSubmit que implementaremos asi:

handleSubmit(event) {  
 ***console***.log(**'Current State is: '** + ***JSON***.stringify(**this**.**state**));  
 *alert*(**'Current State is: '** + ***JSON***.stringify(**this**.**state**));  
 *// evita que el browser por defecto vaya a la pagina siguiente* event.preventDefault();  
}

como estamos llamando a las funciones desde el formulario sin un arrow function, necesitamos hacer bind de estas en el constructor asi:

**this**.handleInputChange = **this**.handleInputChange.bind(**this**);  
**this**.handleSubmit = **this**.handleSubmit.bind(**this**);

Hacemos esto para proveer el mismo contexto al hacer la lamada en el uso de this.

Tambien puede usar CTRL+J bind para escribir el bining del método que desee.

Si no deseamos escribir bindings debería alternativamente escribir la función como un arrow function asi:

*handleInputChange* = (event) => {  
 **const** target = event.**target**;  
 **const** value = target.**type** === **'checkbox'** ? target.**checked** : target.**value**;  
 **const** name = target.**name**;  
  
 **this**.setState({  
 [name]: value  
 });  
};

Verifique la aplicación ingresando datos al formulario de contacto y haciendo click en submit.

Haga commit con el mensaje “Controlled Forms”.

# Validaciones en formularios controlados

En state de Component agregar una variable touched que mantendrá estado de si un field del formulario se ha tocado o no, esto con el fin de no hacer validaciones si el usuario no ha tocado el field. Por lo cual asignamos false a cada variable de control asi:

message: '',  
touched: {  
 firstname: false,  
 lastname: false,  
 telnum: false,  
 email: false  
}

Cuando el usuario modifique cualquiera de estos input el valor debe cambiarse a true, para esto escribiremos un método llamado handleBlur asi:

*handleBlur* = (field) => (evt) => {  
 **this**.setState({  
 **touched**: { ...**this**.**state**.**touched**, [field]: **true** },  
 });  
};

Usted puede ver el equivalente del compilado js en <https://babeljs.io/>

Usaremos una función de validación de los inputs por ejemplo asi:

validate(firstname, lastname, telnum, email) {  
 **const** errors = {  
 **firstname**: **''**,  
 **lastname**: **''**,  
 **telnum**: **''**,  
 **email**: **''** };  
  
 **if** (**this**.**state**.**touched**.**firstname** && firstname.**length** < 3)  
 errors.**firstname** = **'First Name should be >= 3 characters'**;  
 **else if** (**this**.**state**.**touched**.**firstname** && firstname.**length** > 10)  
 errors.**firstname** = **'First Name should be <= 10 characters'**;  
  
 **if** (**this**.**state**.**touched**.**lastname** && lastname.**length** < 3)  
 errors.**lastname** = **'Last Name should be >= 3 characters'**;  
 **else if** (**this**.**state**.**touched**.**lastname** && lastname.**length** > 10)  
 errors.**lastname** = **'Last Name should be <= 10 characters'**;  
  
 **const** reg = /^\d+$/;  
 **if** (**this**.**state**.**touched**.**telnum** && !reg.test(telnum))  
 errors.**telnum** = **'Tel. Number should contain only numbers'**;  
  
 **if** (**this**.**state**.**touched**.**email** && email.split(**''**).filter(x => x === **'@'**).**length** !== 1)  
 errors.**email** = **'Email should contain a @'**;  
  
 **return** errors;  
}

Usted puede revisar expresiones regulares con ayuda de paginas online como <https://regexr.com/>

En la función render llamaremos a la función validate para capturar los errores de validación si existen:

render() {  
 const errors = this.validate(this.state.firstname, this.state.lastname, this.state.telnum, this.state.email);

En el formulario para cada input que se esta validando incluiremos un tag para mostrar errores si existen por ejemplo asi:

<Input type="text" id="firstname" name="firstname"  
 placeholder="First Name"  
 value={this.state.firstname}  
 onChange={this.handleInputChange}/>  
<FormFeedback>{errors.firstname}</FormFeedback>

Incluya el tag FormFeedback para lastname, telnum y email.

Para ver si un componente fue tocado tenemos la función handleBlur la cual invocaremos en el evento onBlur de cada input: firstname, lastname, telnum y email por ejemplo asi:

<Input type="text" id="firstname" name="firstname"  
 placeholder="First Name"  
 value={this.state.firstname}  
 onBlur={this.*handleBlur*('firstname')}  
 onChange={this.handleInputChange}/>

Debemos proveer el binding en el constructor para handleBlur presione CTRL+J bind

**this**.*handleBlur* = **this**.*handleBlur*.bind(**this**);

Note que handleBlur es una función flecha por lo cual el binding seria indiferente o innecesario.

Usaremos la propiedad valid e invalid de los input para especificar si el input es valido o no, es decir, si tiene errores o no asi:

<Input type="text" id="firstname" name="firstname"  
 placeholder="First Name"  
 value={this.state.firstname}  
 valid={errors.firstname === ''}  
 invalid={errors.firstname !== ''}  
 onBlur={this.*handleBlur*('firstname')}  
 onChange={this.handleInputChange}/>

Haga lo propio para lastname, telnum y email.

Pruebe los mensajes de validación en el formulario.

Haga commit con el mensaje “Controlled Form Validation”

# Uncontrolled forms

En el componente header prepararemos un modal que será un formulario no controlado para el login del usuario aparesca en primer plano.

Debajo del Jumbotron del header adicone el modal asi:

<Modal isOpen={this.state.isModalOpen} toggle={this.toggleModal}>  
 <ModalHeader toggle={this.toggleModal}>Login</ModalHeader>  
 <ModalBody>  
  
 </ModalBody>  
</Modal>

En el state adicionaremos la variable isModalOpen que almacenara si el modal esta abierto o no, inicialmente asignado a false.

Para hacer toogle del modal agregaremos esta función:

toggleModal() {  
 **this**.setState({  
 **isModalOpen**: !**this**.**state**.**isModalOpen** });  
}

Adicione el bind para esa función en el constructor.

Para invocar al modal adicionaremos un botón en el header otro Nav mas asi:

<Nav className="ml-auto" navbar>  
 <NavItem>  
 <Button outline onClick={this.toggleModal}><span className="fa fa-sign-in fa-lg"></span> Login</Button>  
 </NavItem>  
</Nav>

Observe que ml-auto dara el máximo margen posible, de forma que el botón quedaría situado a la derecha del Navbar.

Pruebe el botón y vea como se abre y cierra el modal.

Dentro el body del modal cree un formulario Ctrl+J rf y pase como manejador de submit al método handleLogin que implementaremos luego, debe quedar asi:

<Form onSubmit={this.handleLogin}>  
</Form>

Ahora agregaremos los elementos del formulario no controlado asi:

<Form onSubmit={this.handleLogin}>  
 <FormGroup>  
 <Label htmlFor="username">Username</Label>  
 <Input type="text" id="username" name="username"  
 innerRef={(input) => this.username = input} />  
 </FormGroup>  
 <FormGroup>  
 <Label htmlFor="password">Password</Label>  
 <Input type="password" id="password" name="password"  
 innerRef={(input) => this.password = input} />  
 </FormGroup>  
 <FormGroup check>  
 <Label check>  
 <Input type="checkbox" name="remember"  
 innerRef={(input) => this.remember = input} />  
 Remember me  
 </Label>  
 </FormGroup>  
 <Button type="submit" value="submit" color="primary">Login</Button>  
</Form>

Agregaremos la function handleLogin mas su bind, que es invocada en el submit del formulario asi:

handleLogin(event) {  
 this.toggleModal();  
 *alert*("Username: " + this.username.value + " Password: " + this.password.value  
 + " Remember: " + this.remember.checked);  
 event.preventDefault();  
  
}

Haga commit con el mensaje “Uncontrolled Form“

# Redux

Para poder usar Redux y React-Redux ejecute los siguientes comandos:

yarn add redux@3.7.2

yarn add react-redux@5.0.7

En la carpeta src cree un nuevo folder llamado redux.

Crear un archivo llamado reducer.js en la carpeta redux.

Como ahora obtendremos los datos aplicando redux moveremos la información de ítems, employees y comments del componente Main al reducer para esto: corte el state y pegar en el reducer.

El documento que almacena state para nosotros debe ser un objeto constante llamado initialState el cual exportaremos asi:

**export const** initialState = {  
 **items**: ITEMS,

A continuación implemente una función llamada Reducer que devolverá el estado sin ningún cambio. Debemos recordar que el reducer debe recibir como parámetro el state y el action, debe quedar asi:

**export const** *Reducer* = (state, action) => {  
 **return** state;  
};

En el Store de redux cuando realicemos la primera llamada a esta función reducer, el store estará vacio, por lo cual le diremos que en tal situación cuando state no tenga valor asigne por defecto su valor a initialState asi:

**export const** *Reducer* = (state = initialState, action) => {  
 **return** state;  
};

Esta función Reducer es una función pura porque no modifica el state que recibe como input.

Ahora debemos configurar el store, para esto crearemos un archivo llamado configureStore.js en la carpeta redux.

Defina y exporte un arrow function llamada ConfigureStore. En ella llame a la función createStore de redux pasando como parámetros la función Reducer e initialState. Asigne el valor de la llamada a createStore a una constante llamada store asi:

**export const** *ConfigureStore* = () => {  
 **const** store = createStore(*Reducer*,  
 initialState);  
};

Finalmente devuelva store.

Ahora estamos listos para usar el store creado desde la aplicación.

En App.js justo antes de la definición de clase crearemos una constante llamada store y asignaremos a esta la llamada a ConfigureStore() de redux.

Envolveremos el tag BrowserRouter con el tag Provider de react-redux con parámetro store igual a store asi:

<Provider store={store}>  
 <BrowserRouter>

Ahora el store que creamos esta disponible para nuestra aplicación.

En MainComponent.js conectaremos y usaremos el store de Redux. Para esto crearemos fuera de la definición de clase una función flecha llamada mapStateToProps que recibe como parámetro state y devuelve un objeto con variables como ítems asignadas a store.items asi:

**const** *mapStateToProps* = (state) => {  
 **return** {  
 **items**: state.**items**,  
 **employees**: state.**employees**,  
 **comments**: state.**comments** }  
};

Esta función trae los datos del Store que definimos en redux a el objeto props de nuestro componente.

Para conectar el componente Main debemos en el export del final del archivo envolverlo en una función connect que recibe como parametro la función mapStateToProps y construye una función que a su vez recibe como parámetro a Main. Como estamos usando react router ademas debemos envolver todo eso con la llamada a withRouter, asi:

**export default** *withRouter*(connect(*mapStateToProps*)(Main));

En el componente Main los datos ahora están en props traídos desde el store, por lo cual cambiaremos las referencias a state por props cuando accedemos a ítems, employees, y comments.

Verifique todas las paginas de la aplicaion y vera que todo sigue funcionando igual que antes a diferencia que ahora los datos vienen del store de redux.

Haga commit con el mensaje “create Store, connect to redux and use store”.

24 enero

# React Redux Form

Instalaremos las dependencias con el comando:

yarn add react-redux-form@1.16.8

Ahora actualizaremos el formulario en ContactComponent.

Borraremos state del componente porque usaremos el Store de Redux.

Tampoco es necesario las funciónes handleInputChange ni handleBlur por lo cual las borraremos puesto que esto es manejado por react-redux-form.

No olvide borrar los bind de las funciones que quitamos.

Lo mismo para la función validate, por lo cual la borraremos también. En el método render ademas borraremos la llamada a validate.

Cambie el Tag Form por LocalForm. En este tag modificaremos la propiedad onSubmit asignando un función flecha que recibe values como parámetro y de cuerpo llama a handleSubmit pasándole values asi:

<**LocalForm onSubmit=**{(values) => **this**.handleSubmit(values)}>

Cambiaremos los Tag FormGroup por Row con className form-group asi:

<Row className="form-group">

Por ahora quitaremos los FormFeedback.

Cambiaremos los Tag Input por Control.text donde Control es de react-redux-form. Quitaremos la propiedad type= text y adicionaremos la propiedad model igual a .firstName. Tambien adicionaremos la propiedad className igual a form-control.

Tambien es necesario remover las propiedades value, valid, invalid, onBlur y onChange

Aplique lo mismo para los otros elementos input del formulario. Tenga cuidado para el text área usara Control.textarea. Para el input tipo select usara Control.select. Para el Input tipo checkbox usara Control.checkbox y use className igual a form-check-input ademas remueva checke y onChange finalmente cambie FormGroup por un div con className igua a form-check.

Debe quedar asi:

<LocalForm onSubmit={(values) => this.handleSubmit(values)}>  
 <Row className="form-group">  
 <Label htmlFor="firstname" md={2}>First Name</Label>  
 <Col md={10}>  
 <Control.text model=".firstname" id="firstname" name="firstname"  
 placeholder="First Name"  
 className="form-control"  
 />  
 </Col>  
 </Row>  
 <Row className="form-group">  
 <Label htmlFor="lastname" md={2}>Last Name</Label>  
 <Col md={10}>  
 <Control.text model=".lastname" id="lastname" name="lastname"  
 placeholder="Last Name"  
 className="form-control"  
 />  
 </Col>  
 </Row>  
 <Row className="form-group">  
 <Label htmlFor="telnum" md={2}>Contact Tel.</Label>  
 <Col md={10}>  
 <Control.text model=".telnum" id="telnum" name="telnum"  
 placeholder="Tel. Number"  
 className="form-control"  
 />  
 </Col>  
 </Row>  
 <Row className="form-group">  
 <Label htmlFor="email" md={2}>Email</Label>  
 <Col md={10}>  
 <Control.text model=".email" id="email" name="email"  
 placeholder="Email"  
 className="form-control" />  
 </Col>  
 </Row>  
 <Row className="form-group">  
 <Col md={{size: 6, offset: 2}}>  
 <div className="form-check">  
 <Label check>  
 <Control.checkbox model=".agree" name="agree"  
 className="form-check-input"  
 /> {' '}  
 <strong>May we contact you?</strong>  
 </Label>  
 </div>  
 </Col>  
 <Col md={{size: 3, offset: 1}}>  
 <Control.select model=".contactType" name="contactType"  
 className="form-control">  
 <option>Tel.</option>  
 <option>Email</option>  
 </Control.select>  
 </Col>  
 </Row>  
 <Row className="form-group">  
 <Label htmlFor="message" md={2}>Your Feedback</Label>  
 <Col md={10}>  
 <Control.textarea model=".message" id="message" name="message"  
 rows="12"  
 className="form-control" />  
 </Col>  
 </Row>  
 <Row className="form-group">  
 <Col md={{size: 10, offset: 2}}>  
 <Button type="submit" color="primary">  
 Send Feedback  
 </Button>  
 </Col>  
 </Row>  
</LocalForm>

Modifique la función handleSubmit para que reciba values como parámetro. Cambie el uso de this.state por values y quite event.preventDefault.

Verifique el funcionamiento del formulario de contacto.

Haga commit con el mensaje “React Redux Form”

# React Redux Form Validation

Veremos las validaciones de react-redux-form.

Antes de la definicion de la clase Contact defina las siguientes funciones:

**const** *required* = (val) => val && val.**length**;  
**const** *maxLength* = (len) => (val) => !(val) || (val.**length** <= len);  
**const** *minLength* = (len) => (val) => val && (val.**length** >= len);  
**const** *isNumber* = (val) => !*isNaN*(***Number***(val));  
**const** *validEmail* = (val) => /^[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/i.test(val);

En el tag Control.text de firstname adicione la propiedad validators asi:

**validators=**{{  
 *required*, **minLength**: *minLength*(3), **maxLength**: *maxLength*(15)  
}}

Para desplegar los errores de firstName use el tag Error despues del tag Control asi:

<Errors  
 className="text-danger"  
 model=".firstname"  
 show="touched"  
 messages={{  
 required: 'Required',  
 minLength: 'Must be greater than 2 characters',  
 maxLength: 'Must be 15 characters or less'  
 }}  
/>

Aplicaremos esto al resto de los campos del formulario y debe quedar asi:

<Row className="form-group">  
 <Label htmlFor="firstname" md={2}>First Name</Label>  
 <Col md={10}>  
 <Control.text model=".firstname" id="firstname" name="firstname"  
 placeholder="First Name"  
 className="form-control"  
 validators={{  
 *required*, minLength: *minLength*(3), maxLength: *maxLength*(15)  
 }}  
 />  
 <Errors  
 className="text-danger"  
 model=".firstname"  
 show="touched"  
 messages={{  
 required: 'Required',  
 minLength: 'Must be greater than 2 characters',  
 maxLength: 'Must be 15 characters or less'  
 }}  
 />  
 </Col>  
</Row>  
<Row className="form-group">  
 <Label htmlFor="lastname" md={2}>Last Name</Label>  
 <Col md={10}>  
 <Control.text model=".lastname" id="lastname" name="lastname"  
 placeholder="Last Name"  
 className="form-control"  
 validators={{  
 *required*, minLength: *minLength*(3), maxLength: *maxLength*(15)  
 }}  
 />  
 <Errors  
 className="text-danger"  
 model=".lastname"  
 show="touched"  
 messages={{  
 required: 'Required',  
 minLength: 'Must be greater than 2 characters',  
 maxLength: 'Must be 15 characters or less'  
 }}  
 />  
 </Col>  
</Row>  
<Row className="form-group">  
 <Label htmlFor="telnum" md={2}>Contact Tel.</Label>  
 <Col md={10}>  
 <Control.text model=".telnum" id="telnum" name="telnum"  
 placeholder="Tel. Number"  
 className="form-control"  
 validators={{  
 *required*,  
 minLength: *minLength*(3),  
 maxLength: *maxLength*(15),  
 *isNumber* }}  
 />  
 <Errors  
 className="text-danger"  
 model=".telnum"  
 show="touched"  
 messages={{  
 required: 'Required',  
 minLength: 'Must be greater than 2 numbers',  
 maxLength: 'Must be 15 numbers or less',  
 isNumber: 'Must be a number'  
 }}  
 />  
 </Col>  
</Row>  
<Row className="form-group">  
 <Label htmlFor="email" md={2}>Email</Label>  
 <Col md={10}>  
 <Control.text model=".email" id="email" name="email"  
 placeholder="Email"  
 className="form-control"  
 validators={{  
 *required*, *validEmail* }}  
 />  
 <Errors  
 className="text-danger"  
 model=".email"  
 show="touched"  
 messages={{  
 required: 'Required',  
 validEmail: 'Invalid Email Address'  
 }}  
 />  
 </Col>  
</Row>

Verifique los mensajes de error en los campos que tienen validaciones y haga submit y vera que todo sigue funcionando como antes mas las validaciones y mensajes de error.

Haga commit con el mensaje “react redux form validations”

# Laboratorio 3: React forms y Redux

En este laboratorio, usted actualizará ItemdetailComponent.js para incluir un formulario y realizar la validación del formulario en el código.

## Tarea 1

* Agregue un nuevo componente de clase llamado CommentForm que agregará un botón a la vista como se muestra en la imagen a continuación.
* Al hacer clic en el botón, se alterna (toogle) un modal que mostrará el formulario de comentarios.
* El componente CommentForm es utilizado por la función RenderComments para mostrar el botón toogle para alternar el modal.

## Tarea 2

En esta tarea, construirá el formulario para que los usuarios envíen sus comentarios como se muestra en la imagen a continuación.

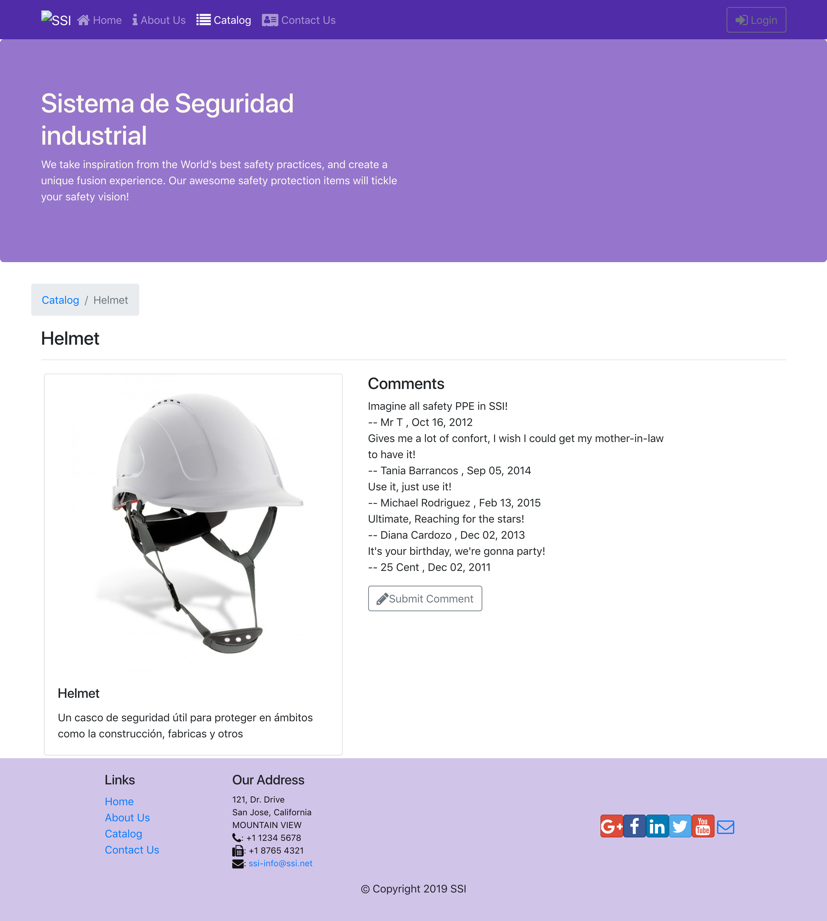
* Configure el formulario como un formulario local utilizando react-redux-form con los campos: author, rating y comment.
* El campo de rating en el formulario de comentarios se implementa usando un select, author se implementa utilizando un campo de texto y comment utiliza un área de texto con seis filas.

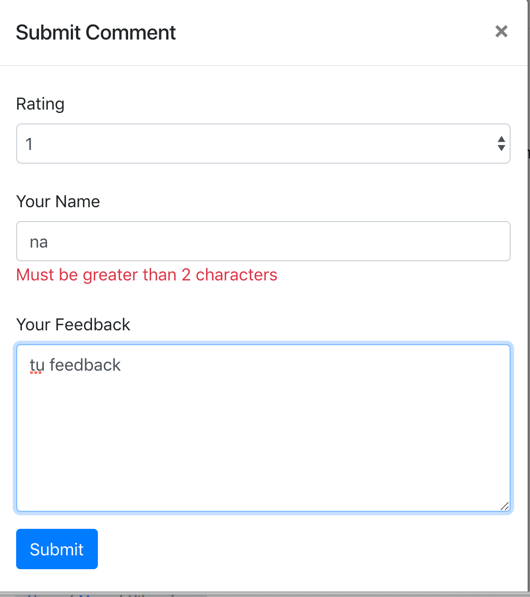
## Tarea 3

En esta tarea, habilitará la validación de formularios como se muestra en las imágenes a continuación.

* El campo author debe tener al menos tres caracteres.
* El campo author debe ser menor o igual a 15 caracteres.
* Se debe alertar al usuario mostrando el mensaje de inválido que se muestra en la parte inferior del campo.

Al final de las tres tareas su aplicación debe verse asi:





# Combinar Reducers

Aprenderemos cómo podemos tener reducers separados que administran partes del estado y cómo pueden combinarse para administrar todo el estado.

A continuación dado que tenemos un solo reducer llamado Reducer en reducer.js entonces crearemos reducers para ítems, comments y employees que manejaran solo las partes de información correspondientes.

En la carpeta redux, cree un nuevo archivo llamado items.js y agregue lo siguiente:

**import** {ITEMS} **from "../shared/items"**;  
  
**export const** *Items* = (state = ITEMS, action) => {  
 **switch** (action.**type**) {  
 **default**:  
 **return** state;  
 }  
};

Luego, cree un nuevo archivo llamado comments.js y agregue lo siguiente:

**import** { COMMENTS } **from '../shared/comments'**;  
  
**export const** *Comments* = (state = COMMENTS, action) => {  
 **switch** (action.**type**) {  
 **default**:  
 **return** state;  
 }  
};

Luego, cree un nuevo archivo llamado employees.js y agregue lo siguiente:

**import** {EMPLOYEES} **from "../shared/employees"**;  
  
**export const** *Employees* = (state = EMPLOYEES, action) => {  
 **switch** (action.**type**) {  
 **default**:  
 **return** state;  
 }  
};

Como hemos dividido la administración del estado en diferentes reductores que administran el estado parcial, debemos combinarlos. Abra configureStore.js y actualícelo asi:

import {*combineReducers*, createStore} from "redux";  
import {*Items*} from "./items";  
import {*Comments*} from "./comments";  
import {*Employees*} from "./employees";  
  
export const *ConfigureStore* = () => {  
 const store = createStore(  
 *combineReducers*({  
 items: *Items*,  
 comments: *Comments*,  
 employees: *Employees* })  
 );  
  
 return store;  
};

Elimine reducer.js

Verifique que todo sigue funcional y haga commit con el mensaje: “Combinando Reducers”

# Acciones de Redux

Aprenderemos a definir varias acciones de Redux e implementaremos los creadores de acciones para enviar las acciones al store de Redux.

Primero crearemos acciones:

En la carpeta redux, crearemos un nuevo archivo llamado ActionTypes.js y agregaremos lo siguiente:

export const ADD\_COMMENT = 'ADD\_COMMENT';

Luego, crearemos un archivo llamado ActionCreators.js y agregaremos lo siguiente:

**import** \* **as** ActionTypes **from './ActionTypes'**;  
  
**export const** *addComment* = (itemId, rating, author, comment) => ({  
 **type**: ActionTypes.ADD\_COMMENT,  
 **payload**: {  
 **itemId**: itemId,  
 **rating**: rating,  
 **author**: author,  
 **comment**: comment  
 }  
});

A continuación, actualice comments.js para iniciar una acción cuando ActionCreator envíe la acción de la siguiente manera:

**import** {COMMENTS} **from '../shared/comments'**;  
**import** \* **as** ActionTypes **from './ActionTypes'**;  
  
**export const** *Comments* = (state = COMMENTS, action) => {  
 **switch** (action.**type**) {  
 **case** ActionTypes.ADD\_COMMENT:  
 **let** comment = action.**payload**;  
 comment.**id** = state.**length**;  
 comment.**date** = **new *Date***().toISOString();  
 ***console***.log(**"Comment: "**, comment);  
 **return** state.concat(comment);  
  
 **default**:  
 **return** state;  
 }  
};

Ahora actualizaremos MainComponent.js, primero agregaremos fuera de la definición de clase esta función:

**const** *mapDispatchToProps* = dispatch => ({  
 addComment: (itemId, rating, author, comment) => dispatch(*addComment*(itemId, rating, author, comment))  
});

En el tag ItemDetail pasaremos por props addComment asi:

<**ItemDetail item=**{**this**.**props**.**items**.filter((item) => item.**id** === *parseInt*(match.**params**.**itemId**, 10))[0]}  
 **comments=**{**this**.**props**.**comments**.filter((comment) => comment.**itemId** === *parseInt*(match.**params**.**itemId**, 10))}  
 **addComment=**{**this**.**props**.addComment}/>

En el export del final del archivo pasaremos mapDispatchToProps como segundo argumento de connect

Finalmente, actualizaremos ItemdetailComponent.js de la siguiente manera para iniciar la acción cuando el usuario envíe el formulario de comentarios. Primero adicionaremos que reciba por props ademas de comments, addComment y itemId

**function** *RenderComments*({comments,addComment, itemId}) {

En el tag CommentForm adiconaremos por propsel itemId y addComment asi:

<CommentForm itemId={itemId} addComment={addComment} />

Para esto enviaremos los props asi:

<RenderComments comments={props.comments} addComment={props.addComment} itemId={props.item.id}/>

En el comment form en lugar del alert use esto:

**this**.**props**.addComment(**this**.**props**.**itemId**, values.**rating**, values.**author**, values.**comment**);

Verifique que ahora es posible adicionar comentarios a un item mediante el formulario y haga commit con el mensaje “Redux Actions”

# Redux Thunk

Instale Redux Thunk y Logger asi:

yarn add redux-thunk@2.2.0

yarn add redux-logger@3.0.6

En configureStore.js pase como Segundo argumento el optimizador del store, para esto usaremos la function applymidleware pasando como parametros a thunk y logger de redux-thunk y redux-logger. Esta function creara el optimizador, asi:

**import** thunk **from "redux-thunk"**;  
**import** {**logger**} **from "redux-logger"**;  
  
**export const** *ConfigureStore* = () => {  
 **const** store = createStore(  
 *combineReducers*({  
 **items**: *Items*,  
 **comments**: *Comments*,  
 **employees**: *Employees* }),  
 *applyMiddleware*(thunk, **logger**)  
 );  
  
 **return** store;  
};

En ActionTypes adicionaremos mas action types asi:

export const ITEMS\_LOADING = 'ITEMS\_LOADING';  
export const ITEMS\_FAILED = 'ITEMS\_FAILED';  
export const ADD\_ITEMS = 'ADD\_ITEMS';

En action creators crearemos mas action creators, para eso declararemos una función flecha asignada a la constante fetchItems la función devolverá otra función que usara como parámetro a dispatch, esta función hara un dispatch con itemsLoading(true) y luego de un un delay de 2 segundos hara ptro dispatch para adicionar ítems asi:

**export const** *fetchItems* = () => (dispatch) => {  
  
 dispatch(itemsLoading(**true**));  
  
 *setTimeout*(() => {

//pondra los items al store  
 dispatch(addItems(ITEMS));  
 }, 2000);  
};

Ahora implementaremos las dos funciones del action creator mas una en caso de error asi:

**export const** *itemsLoading* = () => ({  
 **type**: ActionTypes.ITEMS\_LOADING  
});  
  
**export const** *itemsFailed* = (errmess) => ({  
 **type**: ActionTypes.ITEMS\_FAILED,  
 **payload**: errmess  
});  
  
**export const** *addItems* = (items) => ({  
 **type**: ActionTypes.ADD\_ITEMS,  
 **payload**: items  
});

Estas tres funciones son action creators: itemsLoading, itemsFailed, addItems.

fetchItems es un thunk que llama a dispatch de varias acciones.

En items en el switch haremos un case para las acciones que creamos, ademas cambiaremos el la forma state que ahora sera un documento con 3 fields isLoading:true, errMess:null y items=[], asi:

**export const** *Items* = (  
 state = {  
 **isLoading**: **true**,  
 **errMess**: **null**,  
 **items**: []  
 },  
 action) => {  
 **switch** (action.**type**) {  
 **case** ActionTypes.ADD\_ITEMS:  
  
 **case** ActionTypes.ITEMS\_LOADING:  
  
 **case** ActionTypes.ITEMS\_FAILED:  
   
 **default**:  
 **return** state;  
 }  
};

En es caso de ITEMS\_LOADING retornamos:

**case** ActionTypes.ITEMS\_LOADING:  
 **return** {...state, **isLoading**: **true**, **errMess**: **null**, **items**: []};

observe que estamos clonando state y modificando a isLoading true, mas limpiando los otros datos de acuerdo al caso.

De manera similar para ITEMS\_FAILED:

return {...state, isLoading: false, errMess: action.payload};

y apara ADD\_ITEMS:

return {...state, isLoading: false, errMess: null, items: action.payload};

Cuando los items se esten cargando mostraremos un spinner, para esto crearemos un nuevo componente en el archivo LoadingComponent con el siguiente contenido:

import React from 'react';  
  
export const *Loading* = () => {  
 return(  
 <div className="col-12">  
 <span className="fa fa-spinner fa-pulse fa-3x fa-fw text-primary"></span>  
 <p>Loading . . .</p>  
 </div>  
 );  
};

En MainComponent crearemos un nuevo props fetchItems asi:

**import** {*addComment*, *fetchItems*} **from "../redux/ActionCreators"**;  
  
  
**const** *mapDispatchToProps* = dispatch => ({  
 addComment: (itemId, rating, author, comment) => dispatch(*addComment*(itemId, rating, author, comment)),  
 fetchItems: () => { dispatch(*fetchItems*())}  
});

Cuando requerimos cargar datos usamos el método componentDidMount por ser este el lugar mas conveniente. Sobreeescriba ese método (dentro la definición de clase) y en el llame a fetchItems asi:

componentDidMount() {  
 **this**.**props**.fetchItems();  
}

Cuando el componente se monte llamara al thunk para cargar los ítems en redux y entonces los ítems también estarán disponibles para el uso en los componentes.

Recuerde que hemos cambiado la estructura del store y ahora items esta en ítems.items como se muestra del archivo ítems.js:

state = {  
 **isLoading**: **true**,  
 **errMess**: **null**,  
 **items**: []  
}

Por lo modificaremos el componente funcional Home para reflejar esto, además debemos pasar itemsLoading como props y también el errMess en caso que hubiese ocurrido una falla asi:

**const** *HomePage* = () => {  
 **return** (  
 <**Home  
 item=**{**this**.**props**.**items**.**items**.filter(item => item.**featured**)[0]}  
 **itemsLoading=**{ **this**.**props**.**items**.**isLoading**}

itemsErrMess={this.props.items.errMess}  
 **employee=**{**this**.**props**.**employees**.filter(employee => employee.**featured**)[0]}  
 />

Aplicaremos la misma idea para item detail en el tag de ItemWithId asi:

<ItemDetail item={this.props.items.items.filter((item) => item.id === *parseInt*(match.params.itemId, 10))[0]}  
 isLoading={ this.props.items.isLoading}  
 errMess={this.props.items.errMess}  
 comments={this.props.comments.filter((comment) => comment.itemId === *parseInt*(match.params.itemId, 10))}  
 addComment={this.props.addComment}/>

Ahora en la definición de ItemDetail en el props puede llegar el item nulo mientras se carga y deberíamos mostrar el spinner por lo cual adicionaremos una condicional para mostrar el componente loading asi:

**if** (props.**isLoading**) {  
 **return** (  
 <**div className="container"**>  
 <**div className="row"**>  
 <**Loading**/>  
 </**div**>  
 </**div**>  
 );  
} **else if** (props.**errMess**) {  
 **return** (  
 <**div className="container"**>  
 <**div className="row"**>  
 <**h4**>{props.**errMess**}</**h4**>  
 </**div**>  
 </**div**>  
 );  
} **else if** (props.**item** != **null**) {  
 **return** (

En HomeComponent aplicaremos la misma idea en la definición de RenderCard, pasando en props isLoading y errMess asi:

**function** *RenderCard*({item, isLoading, errMess}) {

En el uso del tag RenderCard pasaremos estos parámetros como props asi:

<**RenderCard item=**{props.**item**} **isLoading=**{props.itemsLoading} **errMess=**{props.itemsErrMess}/>

En la definición de RenderCard aplicamos la misma idea que en ItemDetail asi:

**if** (isLoading) {  
 **return** (  
 <**Loading**/>  
 );  
} **else if** (errMess) {  
 **return** (  
 <**h4**>{errMess}</**h4**>  
 );  
} **else  
 return** (

En CatalogComponent en la definición de Catalog debemos ajustar la ruta para acceder a los ítems asi:

**var** catalog = props.**items**.**items**.map(item => {

en la parte de return de Catalog aplicamos la misma idea para mostrar el spinner y errores asi:

**if** (props.**items**.**isLoading**) {  
 **return** (  
 <**div className="container"**>  
 <**div className="row"**>  
 <**Loading**/>  
 </**div**>  
 </**div**>  
 );  
} **else if** (props.**items**.**errMess**) {  
 **return** (  
 <**div className="container"**>  
 <**div className="row"**>  
 <**div className="col-12"**>  
 <**h4**>{props.**items**.**errMess**}</**h4**>  
 </**div**>  
 </**div**>  
 </**div**>  
 );  
} **else  
 return** (

En el handleSubmit de commentForm corrija, si fuese necesario:

handleSubmit(values) {  
 ***console***.log(**'Current State is: '** + ***JSON***.stringify(values));  
 **this**.**props**.addComment(**this**.**props**.**itemId**, values.**rating**, values.**author**, values.**comment**);  
}

Pruebe la aplicación y observe como en Home aparece el spinner al recargar la pagina, lo propio en catalog y itemdetail.

Haga commit con el mensaje “Redux Thunk”

# React-Redux-Form

Veremos la interaccion entre react-redux-form y el store de Redux. Veremos cómo asignar el formulario en el store para que el estado del formulario se mantenga en el store.

En la carpeta redux cree un archivo forms.js con el siguiente contenido:

export const InitialFeedback = {  
 firstname: '',  
 lastname: '',  
 telnum: '',  
 email: '',  
 agree: false,  
 contactType: 'Tel.',  
 message: ''  
};

en ConfigureStore en el método combineReducers agregue como ultimo parametro después de employees …createForms de react-redux-forms con paramtro un documento json con un campo feedback igual a initialFeedback asi:

*combineReducers*({  
 **items**: *Items*,  
 **comments**: *Comments*,  
 **employees**: *Employees*,  
 ...*createForms*({  
 **feedback**: InitialFeedback  
 })  
}),

En el MainComponent agregaremos un nuevo dispatch mas en mapDispatchToProps asi:

resetFeedbackForm: () => { dispatch(actions.**reset**(**'feedback'**))}

Tenga cuidado en el import de actions debería quedar asi:

import {actions} from "react-redux-form";

En la ruta de contactus pasaremos como porps a resetFeedback asi:

<Route exact path='/contactus' component={() => <Contact resetFeedbackForm={this.props.resetFeedbackForm} />} />

En contact component en el método handleSubmit luego del alert llamaremos a reset asi:

this.props.resetFeedbackForm();

En lugar de LocalForm usaremos Forma asi:

<**Form model="feedback" onSubmit=**{(values) => **this**.handleSubmit(values)}>

Verifique el form de contact y a pesar de navegar a otras paginas en el sistema podrá ver que la información no desaparece hasta que hace submit y se limpian los campos.

Puede ademas mirar los logs de console para ver los logs de react redux form rrf y ver como se van guardando los valores en el formulario.

Haga commit con el mensaje “React Redux Forms parte 2”

# Configurar Json-server

Json-server es un módulo de node y, por lo tanto, se puede instalar globalmente ejecutando:

npm install json-server -g

Si usas Linux o osx antepon sudo para que se ejecute desde cualquier lugar.

Cree una carpeta llamada json-server en src.

Descargue el archivo db.json del drive a esta carpeta.

Cree una carpeta llamada public en json-server

Copie la carpeta images a la carpeta public

Inicie el json-server ejecutando en la ruta de la carpeta json-server el comando:

json-server --watch db.json -p 3001 -d 2000

Pruebe las rutas sugeridas en consola y también las imágenes como:

<http://localhost:3001/images/amy.jpg>

# Comunicación Cliente Servidor: Fetch

Asegúrese que el json-server este corriendo.

Ahora instalaremos fetch con el siguiente comando:

yarn add [cross-fetch@2.1.0](mailto:cross-fetch@2.1.0)

Configuraremos la aplicación para conectarnos al servidor. Para esto, crearemos un archivo llamado baseUrl.js en la carpeta shared y agregue lo siguiente:

export const baseUrl = 'http://localhost:3001/';

En ActionTypes agregue:

export const ADD\_COMMENTS = 'ADD\_COMMENTS';  
export const COMMENTS\_FAILED = 'COMMENTS\_FAILED';

En ActionCreators haremos la comunicación con el servidor en fetchItems, para esto quitaremos el delay artificial y remplazaremos por una llamada al server mediante fetch asi:

**export const** *fetchItems* = () => (dispatch) => {  
 dispatch(*itemsLoading*(**true**));  
 **return** *fetch*(baseUrl + **'items'**)  
 .then(response => response.json())  
 .then(items => dispatch(*addItems*(items)));  
};

De manera similar adicionaremos código para fetch comments asi:

**export const** *fetchComments* = () => (dispatch) => {  
 **return** *fetch*(baseUrl + **'comments'**)  
 .then(response => response.json())  
 .then(comments => dispatch(*addComments*(comments)));  
};  
  
**export const** *commentsFailed* = (errmess) => ({  
 **type**: ActionTypes.COMMENTS\_FAILED,  
 **payload**: errmess  
});  
  
**export const** *addComments* = (comments) => ({  
 **type**: ActionTypes.ADD\_COMMENTS,  
 **payload**: comments  
});

Ahora necesitamos modificar los reducers, para esto en comments.js actualice asi:

**export const** *Comments* = (state = {**errMess**: **null**, **comments**: []}, action) => {  
 **switch** (action.**type**) {  
 **case** ActionTypes.ADD\_COMMENTS:  
 **return** {...state, **errMess**: **null**, **comments**: action.**payload**};  
  
 **case** ActionTypes.COMMENTS\_FAILED:  
 **return** {...state, **errMess**: action.**payload**};  
  
 **case** ActionTypes.ADD\_COMMENT:  
 **let** comment = action.**payload**;  
 comment.**id** = state.**comments**.**length**;  
 comment.**date** = **new *Date***().toISOString();  
 **return** {...state, **comments**: state.**comments**.concat(comment)};  
  
 **default**:  
 **return** state;  
 }  
};

Como las acciones de Redux han sido actualizadas, debemos actualizar los componentes.

Abra MainComponent.js y adicionaremos en mapDispatchToProps:

fetchComments: () => dispatch(*fetchComments*()),

En componentDidMount adicionaremos:

**this**.**props**.fetchComments();

En la definición de ItemWithId actualizaremos la ruta a this.props.comments.comments y de forma similar a los errores del item agreagemos otro tag de errores asi:

<ItemDetail  
 item={this.props.items.items.filter((item) => item.id === *parseInt*(match.params.itemId, 10))[0]}  
 isLoading={this.props.items.isLoading}  
 errMess={this.props.items.errMess}  
 comments={this.props.comments.comments.filter((comment) => comment.itemId === *parseInt*(match.params.itemId, 10))}  
 commentsErrMess={this.props.comments.errMess}  
 addComment={this.props.addComment}/>

Tambien necesitamos actualizar catalog component, en la definición de RenderCatalogItem, en el Card actualizaremos la imagen para que sea provista desde el server, asi:

<CardImg width="100%" src={baseUrl + item.image} alt={item.name}/>

En HomeComponent la misma idea para proveer la imagen, asi:

<**CardImg src=**{baseUrl + item.**image**} **alt=**{item.**name**}/>

También para ItemdetailComponent, asi:

<CardImg width="100%" src={baseUrl + item.image} alt={item.name}/>

Pruebe la aplicación y vea como ahora los ítems y comentarios son traídos desde el json-server, además note que en home la imagen del empleado no se muestra lo cual trabajaremos después.

Haga commit con el mensaje “Fetch del server”

# Manejo de errores

En ActionCreators modificaremos la función fetchItems asi:

**export const** *fetchItems* = () => (dispatch) => {  
 dispatch(*itemsLoading*(**true**));  
 **return** *fetch*(baseUrl + **'items'**)  
 .then(response => {  
 **if** (response.**ok**) {  
 **return** response;  
 } **else** {  
 **let** error = **new *Error***(**'Error '** + response.**status** + **': '** + response.**statusText**);  
 error.**response** = response;  
 **throw** error;  
 }  
 },  
 error => {  
 **var** errmess = **new *Error***(error.**message**);  
 **throw** errmess;  
 })  
 .then(response => response.json())  
 .then(items => dispatch(*addItems*(items)))  
 .catch(error => dispatch(*itemsFailed*(error.**message**)));  
};

Para comments aplica exactamente el mismo manejo de errores asi:

**export const** *fetchComments* = () => (dispatch) => {  
 **return** *fetch*(baseUrl + **'comments'**)  
 .then(response => {  
 **if** (response.**ok**) {  
 **return** response;  
 } **else** {  
 **var** error = **new *Error***(**'Error '** + response.**status** + **': '** + response.**statusText**);  
 error.**response** = response;  
 **throw** error;  
 }  
 },  
 error => {  
 **var** errmess = **new *Error***(error.**message**);  
 **throw** errmess;  
 })  
 .then(response => response.json())  
 .then(comments => dispatch(*addComments*(comments)))  
 .catch(error => dispatch(*commentsFailed*(error.**message**)));  
};

Verificamos que la aplicación sigue funcionando igual.

Detendremos en json-server y observaremos los mensajes de error en la aplicación.

Cambie temporalmente la ruta de ítems y vea como ahora tiene un error 404.

Haga commit con el mensaje “Fetch Manejo de Errors”.

# Post

Aplicaremos fetch para hacer post de los comments.

En ActionCreator.js crearemos una función postComments asi:

**export const** *postComment* = (itemId, rating, author, comment) => (dispatch) => {  
  
 **const** newComment = {  
 **itemId**: itemId,  
 **rating**: rating,  
 **author**: author,  
 **comment**: comment  
 };  
 newComment.**date** = **new *Date***().toISOString();  
  
 **return** *fetch*(baseUrl + **'comments'**, {  
 **method**: **"POST"**,  
 **body**: ***JSON***.stringify(newComment),  
 **headers**: {  
 **"Content-Type"**: **"application/json"** },  
 **credentials**: **"same-origin"** })  
 .then(response => {  
 **if** (response.**ok**) {  
 **return** response;  
 } **else** {  
 **var** error = **new *Error***(**'Error '** + response.**status** + **': '** + response.**statusText**);  
 error.**response** = response;  
 **throw** error;  
 }  
 },  
 error => {  
 **throw** error;  
 })  
 .then(response => response.json())  
 .then(response => dispatch(*addComment*(response)))  
 .catch(error => {  
 ***console***.log(**'post comments'**, error.**message**);  
 *alert*(**'Your comment could not be posted\nError: '** + error.**message**);  
 });  
};

Ahora addComment será mas simple y recibirá directamente el comment, y el payload será igual a comment asi:

**export const** *addComment* = (comment) => ({  
 **type**: ActionTypes.ADD\_COMMENT,  
 **payload**: comment  
});

En comment.js elimine estas líneas:

comment.**id** = state.**comments**.**length**;  
comment.**date** = **new *Date***().toISOString();

En MainComponent haremos que postComment este disponible en lugar de addComment editando el nombre asi:

postComment: (itemId, rating, author, comment) => dispatch(*postComment*(itemId, rating, author, comment))

Aplicamos postComment en lugar de addComment en el tag ItemDetail. Tambien aplicamos el cambio en la definición de ItemDetail en las partes donde se usa addComment. Use la función reemplazar del ide.

Tambien aplica para CommentForm

En la aplicación haga submit de un comentario y vea como se persiste a pesar de reinicar la aplicación.

Observe comments del json-server por el endpoint.

Observe la consola y vea el nextstate en add comment.

Haga commit con el mensaje “Fetch Post Comment”

# Despliegue de la aplicación con Webpack

Ejecute

npm run build

Esto creara una carpeta build.

Para desplegar necesitamos un servidor, en nuestro caso usaremos el json-server

Copie el contenido de la carpeta build en la carpeta public de json-server

Ahora puede acceder a la aplicación React en el enlace http://localhost:3001

# Docker

Instalar Docker community edition edge. Usuarios Windows necesitan virtual box instalado.

Ejecutar el comando

docker version

docker run hello-world

# Dockerizacion del cliente

En el archive README.md debe mencionar las instrucciones:

Crearemos un archivo llamado DockerFile con el contenido:

# Stage 0, "build-stage", based on Node.js, to build and compile the frontend  
FROM tiangolo/node-frontend:10 as build-stage

WORKDIR /app

COPY package\*.json /app/

RUN npm install

COPY ./ /app/

RUN npm run build

# Stage 1, based on Nginx, to have only the compiled app, ready for production with Nginx  
FROM nginx:1.15

COPY --from=build-stage /app/build/ /usr/share/nginx/html

# Copy the default nginx.conf provided by tiangolo/node-frontend  
COPY --from=build-stage /nginx.conf /etc/nginx/conf.d/default.conf

Crearemos un archivo .dockerignore con el contenido

node\_modules

Construiremos la imagen docker con el comando:

docker build -t ssi-react-app .

Usted puede hacer push de su imagen a docker hub opcionalmente

docker push edsonterceros/ssi-react-app:v0.0.0-rc.1

Usted puede hacer pull de su imagen a docker hub opcionalmente

docker pull tucuentaDocker/ssi-react-app:v0.0.0-rc.1

Ejecutar la imagen con

docker run -p 80:80 ssi-react-app

Verificar la aplicación corriendo en localhost

Otros comandos docker

docker kill $(docker ps -q) elimina procesos de contenedores

docker rm $(docker ps -a -q) elimina contenedores

# Recursos

[Git cheat sheet](https://www.atlassian.com/dam/jcr:8132028b-024f-4b6b-953e-e68fcce0c5fa/atlassian-git-cheatsheet.pdf)

[NPM Documentation](https://docs.npmjs.com/)

[lite-server](https://github.com/johnpapa/lite-server)

[Reactjs.org](https://reactjs.org/)

[create-react-app](https://github.com/facebook/create-react-app)

[reactstrap](https://reactstrap.github.io/)

[reactstrap Navbar](https://reactstrap.github.io/components/navbar/)

[Introducing JSX](https://reactjs.org/docs/introducing-jsx.html)

[Convert JSX using Online Babel Compiler](https://babeljs.io/repl/#?presets=react&code_lz=GYVwdgxgLglg9mABACwKYBt1wBQEpEDeAUIogE6pQhlIA8AJjAG4B8AEhlogO5xnr0AhLQD0jVgG4iAXyJA)

[React Components](https://reactjs.org/docs/components-and-props.html)

[React Component State](https://reactjs.org/docs/state-and-lifecycle.html)

[React Component Props](https://reactjs.org/docs/components-and-props.html)

[reactstrap Media Object](https://reactstrap.github.io/components/media/)

[React.Component and Lifecycle Methods](https://reactjs.org/docs/react-component.html)

[reactstrap Card](https://reactstrap.github.io/components/card/)

[Bootstrap unstyled list](http://getbootstrap.com/docs/4.0/content/typography/#unstyled)

[Bootstrap Grid](http://getbootstrap.com/docs/4.0/layout/grid/)

[React Dev Tools](https://github.com/facebook/react-devtools)

[React Dev Tools Chrome Extension](https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi)

[Presentational and Container Components](https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0)

[Presentational and Container Components (Redux Perspective)](https://redux.js.org/basics/usage-with-react)

[React Component Patterns](https://levelup.gitconnected.com/react-component-patterns-ab1f09be2c82)

[Functional Stateless Components in React](https://javascriptplayground.com/functional-stateless-components-react/)

[react-router](https://github.com/ReactTraining/react-router)

[react-router-dom](https://github.com/ReactTraining/react-router/tree/master/packages/react-router-dom)

[React Router Documentation](https://reacttraining.com/react-router/)

[React Router Dom Documentation](https://reacttraining.com/react-router/web/guides/philosophy)

[React Router DOM: set-up, essential components, & parameterized routes](https://blog.logrocket.com/react-router-dom-set-up-essential-components-parameterized-routes-505dc93642f1)

[Basic intro to React Router v4](https://medium.com/@thejasonfile/basic-intro-to-react-router-v4-a08ae1ba5c42)

[A Simple React Router v4 Tutorial](https://medium.com/@pshrmn/a-simple-react-router-v4-tutorial-7f23ff27adf)

[Arrow Functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

[Single Page Applications (Wikipedia)](https://en.wikipedia.org/wiki/Single-page_application)

[Deep linking](https://en.wikipedia.org/wiki/Deep_linking)

[Single Page Apps in depth](http://singlepageappbook.com/)

[SPA and the Single Page Myth](http://www.johnpapa.net/pageinspa/)

[Controlled Components](https://reactjs.org/docs/forms.html)

[reactstrap Form](https://reactstrap.github.io/components/form/)

[Controlled / Uncontrolled React Components](https://www.viget.com/articles/controlling-components-react/)

[Controlled and Uncontrolled Input Values in React](https://medium.com/@peter.yun.kim/controlled-and-uncontrolled-input-values-in-react-907119cc98d4)

[Controlled and uncontrolled form inputs in React don't have to be complicated](https://goshakkk.name/controlled-vs-uncontrolled-inputs-react/)

[How to Work with Forms, Inputs and Events in React](https://medium.com/capital-one-developers/how-to-work-with-forms-inputs-and-events-in-react-c337171b923b)

[Some Thoughts On Forms in React](https://medium.com/javascript-inside/some-thoughts-on-forms-in-react-9ca2d9078c20)

[Instant form field validation with React's controlled inputs](https://goshakkk.name/instant-form-fields-validation-react/)

[Uncontrolled Components](https://reactjs.org/docs/uncontrolled-components.html)

[reactstrap Form](https://reactstrap.github.io/components/form/)

[reactstrap Modal Component](https://reactstrap.github.io/components/modals/)

[Controlled / Uncontrolled React Components](https://www.viget.com/articles/controlling-components-react/)

[Controlled and Uncontrolled Input Values in React](https://medium.com/@peter.yun.kim/controlled-and-uncontrolled-input-values-in-react-907119cc98d4)

[Controlled and uncontrolled form inputs in React don't have to be complicated](https://goshakkk.name/controlled-vs-uncontrolled-inputs-react/)

[How to Work with Forms, Inputs and Events in React](https://medium.com/capital-one-developers/how-to-work-with-forms-inputs-and-events-in-react-c337171b923b)

[Some Thoughts On Forms in React](https://medium.com/javascript-inside/some-thoughts-on-forms-in-react-9ca2d9078c20)

[Redux](https://redux.js.org/)

[Redux on Github](https://github.com/reactjs/redux)

[React and Redux](https://redux.js.org/basics/usage-with-react)

[Redux Basics Documentation](https://redux.js.org/basics)

[The Flux Architecture](https://facebook.github.io/flux/)

[Redux Tutorials](https://github.com/markerikson/react-redux-links/blob/master/redux-tutorials.md)

[Flux Architecture In Depth Overview](https://facebook.github.io/flux/docs/in-depth-overview.html#content)

[React Redux Tutorial Valentinog](https://www.valentinog.com/blog/redux/)

[React Redux Form Documentation](https://davidkpiano.github.io/react-redux-form/docs.html)

[The boring React Redux forms](https://medium.com/@steida/the-boring-react-redux-forms-a15ee8a6b52b)

[How to populate react-redux-form with dynamic default values](https://swizec.com/blog/populate-react-redux-form-dynamic-default-values/swizec/8158)

[Should you store your form state in Redux?](https://goshakkk.name/should-i-put-form-state-into-redux/)

[Redux Actions](https://redux.js.org/basics/actions)

[Redux Reducers](https://redux.js.org/basics/reducers)

[Redux Usage with React](https://redux.js.org/basics/usage-with-react)

[Redux Middleware](https://redux.js.org/advanced/middleware)

[Redux Thunk](https://github.com/gaearon/redux-thunk)

[Redux Logger](https://github.com/evgenyrodionov/redux-logger)

[React-redux-form](https://davidkpiano.github.io/react-redux-form/docs.html)

[json-server](https://github.com/typicode/json-server)

[Creating Demo APIs with json-server](https://egghead.io/lessons/nodejs-creating-demo-apis-with-json-server)

[JSON](http://www.json.org/)

[Cross-Fetch](https://github.com/lquixada/cross-fetch)

Promise Resources

[JavaScript Promise](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise)

[JS Promise (Part 1 - Basics)](https://medium.com/@ramsunvtech/promises-of-promise-part-1-53f769245a53)

[JavaScript Promises for Dummies](https://scotch.io/tutorials/javascript-promises-for-dummies)

[JavaScript Promises: an Introduction](https://developers.google.com/web/fundamentals/getting-started/primers/promises)

[Introduction to fetch()](https://developers.google.com/web/updates/2015/03/introduction-to-fetch)

[Using Fetch](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch)

[Fetch vs. Axios.js for making http requests](https://medium.com/@thejasonfile/fetch-vs-axios-js-for-making-http-requests-2b261cdd3af5)

# Definiciones

[Framework](https://en.wikipedia.org/wiki/Software_framework)

[Hollywood Principle](https://en.wikipedia.org/wiki/Hollywood_principle)

[Inversion of Control](https://en.wikipedia.org/wiki/Inversion_of_control)

[Imperative vs Declarative Programming](https://netguru.co/blog/imperative-vs-declarative)

[Imperative vs Declarative](http://latentflip.com/imperative-vs-declarative)