

# Tema 5

## React en el mundo real

# Routing

- Nuestro Ecommerce tiene un fallo claro...
- El usuario no puede navegar adelante/atrás por nuestras páginas
- Al construir una SPA, no queremos romper los hábitos del usuario
- Las URLs no pueden compartirse ni añadirse a marcadores

# Routing en React

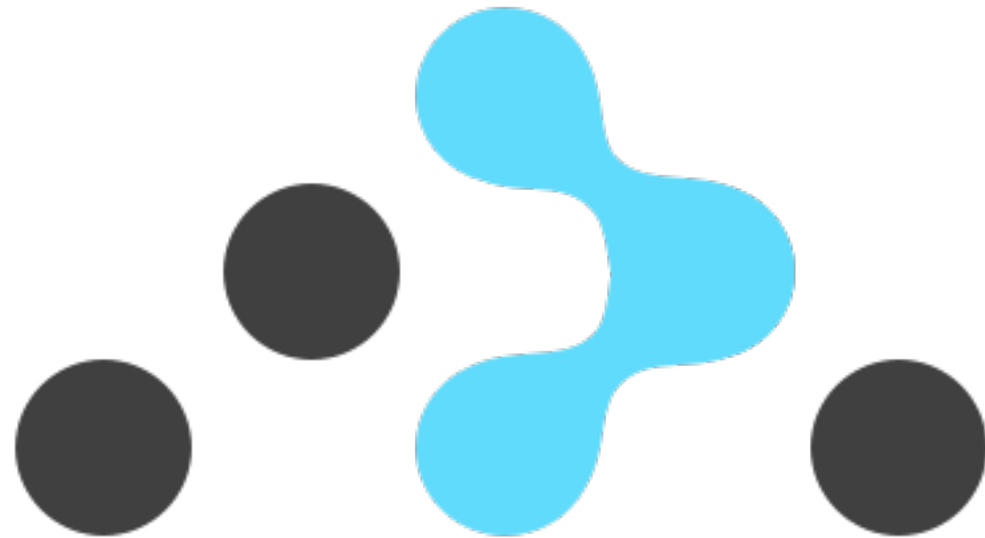
- React no incluye ninguna solución de routing
- Puedes emplear el que quieras...  
Backbone.Router, Page, Director
- Pero en realidad, ¿para qué usamos el router?

# Routing en React

- Pintar un componente/layout por ruta
- Ejecutar código cuando haya coincidencia de rutas

# React-router

<https://github.com/rackt/react-router>



**REACT/ROUTER**

# React-router

- Las rutas se definen como componentes React
- ¿ Rutas == UI ?
- Igual que JSX: “dale cinco minutos”
- **npm install --save react-router**

# React-router

```
const ExampleRouter = React.createClass({
  render() {
    return (
      <Router history={ history }>
        <Route path="/" component={Layout}>
          <IndexRoute component={Home} />
          <Route path="about" component={About} />
          <Route onEnter={ loadContact }
            path="contact/:id" component={Contact} />
        </Route>
      </Router>
    )
  }
});
```

# React-router

```
const ExampleRouter = React.createClass({  
  render() {  
    return (  
      ➡ <Router history={ history }>  
        <Route path="/" component={ Layout }>  
          <IndexRoute component={ Home } />  
          <Route path="about" component={ About } />  
          <Route onEnter={ loadContact }  
            path="contact/:id" component={ Contact } />  
        </Route>  
      </Router>  
    )  
  }  
});
```

El **Router** es nuestro componente de más alto nivel



# React-router

```
const ExampleRouter = React.createClass({  
  render() {  
    return (  
      <Router history={ history }>  
        ➔ <Route path="/" component={ Layout }>  
          <IndexRoute component={ Home } />  
          <Route path="about" component={ About } />  
          <Route onEnter={ loadContact }  
            path="contact/:id" component={ Contact } />  
        </Route>  
      </Router>  
    )  
  }  
});
```

Sus hijos son rutas individuales  
Definen para un **path** dado, qué  
componente debe mostrarse

# React-router

```
//routes
```

```
import ExampleRouter from './routes/example';
```

```
window.onload = function() {  
  ReactDOM.render(<ExampleRouter />, document.getElementById('app'));  
}
```

# Ventajas React-Router

- Rutas anidadas: un componente A que debe mostrarse en la ruta “/contactos” y otro, adicional, que debe ir **dentro** de A en la ruta “/contactos/1/edit” por ejemplo
- La configuración de rutas tiene la misma **forma** que la UI que queremos generar
- Posibilidad de ejecutarlo en el servidor (server-side rendering o SSR)

# Configuración de rutas

- Componente **Route** acepta como props obligatorios:
  - **path** - Ruta con parámetros al estilo habitual
  - **component** - Componente React que debe mostrarse para esa ruta
- Opcionales:
  - **onEnter** - función de transición de entrada
  - **onLeave** - función de transición de salida

# Configuración de rutas

- **onEnter** espera una función la siguiente firma

`function (nextState, replaceState)`

- Donde nextState contiene el “estado” del Router, por ejemplo los parámetros de ruta en **nextState.params** y replaceState es una función para reescribir la ruta

- O si queremos que nuestro código sea asíncrono:

`function(nextState, replaceState, done)`

- Cuando queramos que se complete la transición, llamaremos a **done**

# Configuración de rutas

- Los componentes **Route** se pueden anidar para crear rutas anidadas complejas y UIs complejas
- Si la ruta tiene parámetros, el componente los recibe como props, dentro de la clave **params**
- El componente debe incluir en su **render** `{ props.children }` para pintar los componentes anidados si la ruta los acepta

# Sintaxis para path

- **:parametro** - Concide con un segmento en la URL hasta el siguiente separador (/, #, ?). Ese valor lo tendremos en **props.params.parametro** en el Route Component
- **(x)** - Opcional
- **\*** - Coincide con cualquier carácter (non-greedy)
- **\*\*** - Coincide con cualquier carácter (greedy)

# Sintaxis para path

```
// matches /hello/michael and /hello/ryan  
<Route path="/hello/:name">
```

```
// matches /hello, /hello/michael, and /hello/ryan  
<Route path="/hello(/:name)">
```

```
// matches /files/hello.jpg and /files/hello.html  
<Route path="/files/*.*">
```

```
// matches /files/hello.jpg and /files/path/to/file.jpg  
<Route path="/**/*.jpg">
```



# Anidar rutas

- React Router nos permite declarar conjuntos de vistas anidadas que queremos mostrar al entrar en una URL
- Al buscar coincidencias, React Router busca en profundidad, de dentro a fuera

# Anidar rutas - ejemplo

- **/contacts/:id/edit**
- Para esa ruta queremos mostrar el layout de la aplicación completo, la página de contactos y dentro de ella el formulario para editar un contacto específico

# Anidar rutas - ejemplo

```
<Router history={ history }>
  <Route path="/" component={Layout}>
    <IndexRoute component={Home} />
    <Route path="contacts" component={ Contacts }>
      <Route onEnter={ loadContact } path=":id" component={Contact} />
    </Route>
  </Route>
</Router>
```

# Anidar rutas

```
<Router history={ history }>
  <Route path="/" component={Layout}>
    <IndexRoute component={Home} />
    <Route path="contacts" component={ Contacts }>
      <Route onEnter={ loadContact } path=":id" component={Contact} />
    </Route>
  </Route>
</Router>
```

Si la ruta es "/" se pintará Home dentro de Layout

# Confirmar navegación

- Si queremos “cancelar” la navegación desde un componente, podemos incluir el mixin **Lifecycle** de react-router
- Implementamos la función **routerWillLeave** en nuestro componente
- Si devolvemos un texto, el navegador lo mostrará como confirmación. “¿Seguro que quiere .... “?”
- Si devolvemos **false** se cancela directamente
- Útil para formularios a medio completar

# Integrar React Router con Flux

- Ejecutar **action creators** cuando coincida una ruta
- Modificar la ruta desde fuera del Router:
  - Navegar desde un Store
  - Navegar desde un componente

# Action creators en rutas

- Utilizamos **onEnter** en nuestro componente **<Route>** para lanzar las acciones que queramos
- O bien las llamamos desde **componentDidMount** / **componentWillReceiveProps** del componente de esa ruta

# Action creators en rutas

```
const ExampleRouter = React.createClass({
  loadContact(nextState) {
    //fetch contact from server
    myActions.loadContact(nextState.params.id);
  },
  render() {
    return (
      <Router history={ history }>
        <Route path="/" component={Layout}>
          <IndexRoute component={Home} />
          <Route path="contacts" component={ Contacts }>
            <Route path=":id" onEnter={ this.loadContact } component={Contact} />
          </Route>
        </Route>
      </Router>
    )
  }
});
```



# Navegar desde Stores

- ¿Y si queremos que al atender una acción en un Store, se modifique la ruta?
- Desde el Store no tenemos acceso al Router
- La solución: **history**

# history

- react-router depende de otro módulo npm: **history**, ya lo tenemos incluido al instalar react-router
- Podemos crear mecanismos de routing diferentes: basados en hash, HTML5, en memoria
- Una instancia de history incluye métodos para cambiar la URL: **push** para añadirla al histórico, **replace** para cambiar la actual por otra (reemplazándola)

# history

- Navegación basada en hash (/#ruta):  
**createHashHistory**

```
// src/lib/history.js
import createHistory from 'history/lib/createHashHistory'
export default createHistory({
  queryKey: false
});
```

# history

- Navegación basada en rutas absolutas (/ruta/a/b):  
**createBrowserHistory**

```
// src/lib/history.js  
import createHistory from 'history/lib/createBrowserHistory'  
export default createHistory({  
  queryKey: false  
});
```

# Configurar **history** en el Router

```
import history from '../..../lib/history';
```

```
const Shop = React.createClass({  
  render() {  
    return (  
      <div className='shopping-cart'>  
        <Router history={ history }>  
          ...  
        </Router>  
      </div>  
    );  
  }  
});
```

# Utilizar history en un Store


```
import history from '../lib/history';
```

```
...
```

```
var __page = 'catalog';
```

```
function changePage(newPage) {
```

```
  __page = newPage;
```

```
   history.push(newPage);  
}
```

Llamaremos a **changePage** desde  
**\_\_onDispatch** del Store

# Navegar desde componentes (3 opciones)

1. Usamos el componente **Link** de react-router que pinta una etiqueta de enlace: **<a>**

```
<Link to="ruta" className="xxx" onClick="xxx">Enlace</Link>
```

2. Importamos nuestro **history** como hemos hecho con los Stores (si no nos sirve un enlace)

3. O bien despachamos una acción de navegación, si ya tenemos **history** en nuestro Store

# Usar `<Link>` en lugar de `<a>`

- En **ecommerce/thankyou.js**

```
import { Link } from 'react-router';

const ThankYou = React.createClass({
  render() {
    const orderDetails = OrderStore.getDetails();

    return (
      <div className="thank-you">
        <Header text={'Gracias por tu compra ' + orderDetails.firstName} />
        <p>Te llegará en breve a { orderDetails.address }</p>
        <p><Link to="/" className="button">Volver a la tienda</Link></p>
      </div>
    )
  }
});
```



# Ejercicio - Rutas para nuestro Ecommerce

- Podemos usar nuestro componente **ecommerce/index.js**
- Rutas
  - “/” -> Catalog
  - “/cart” -> Cart
  - “/checkout” -> Checkout
  - “/thankyou” -> ThankYou
  - “\*” -> NotFound (nuestro 404 en el cliente)

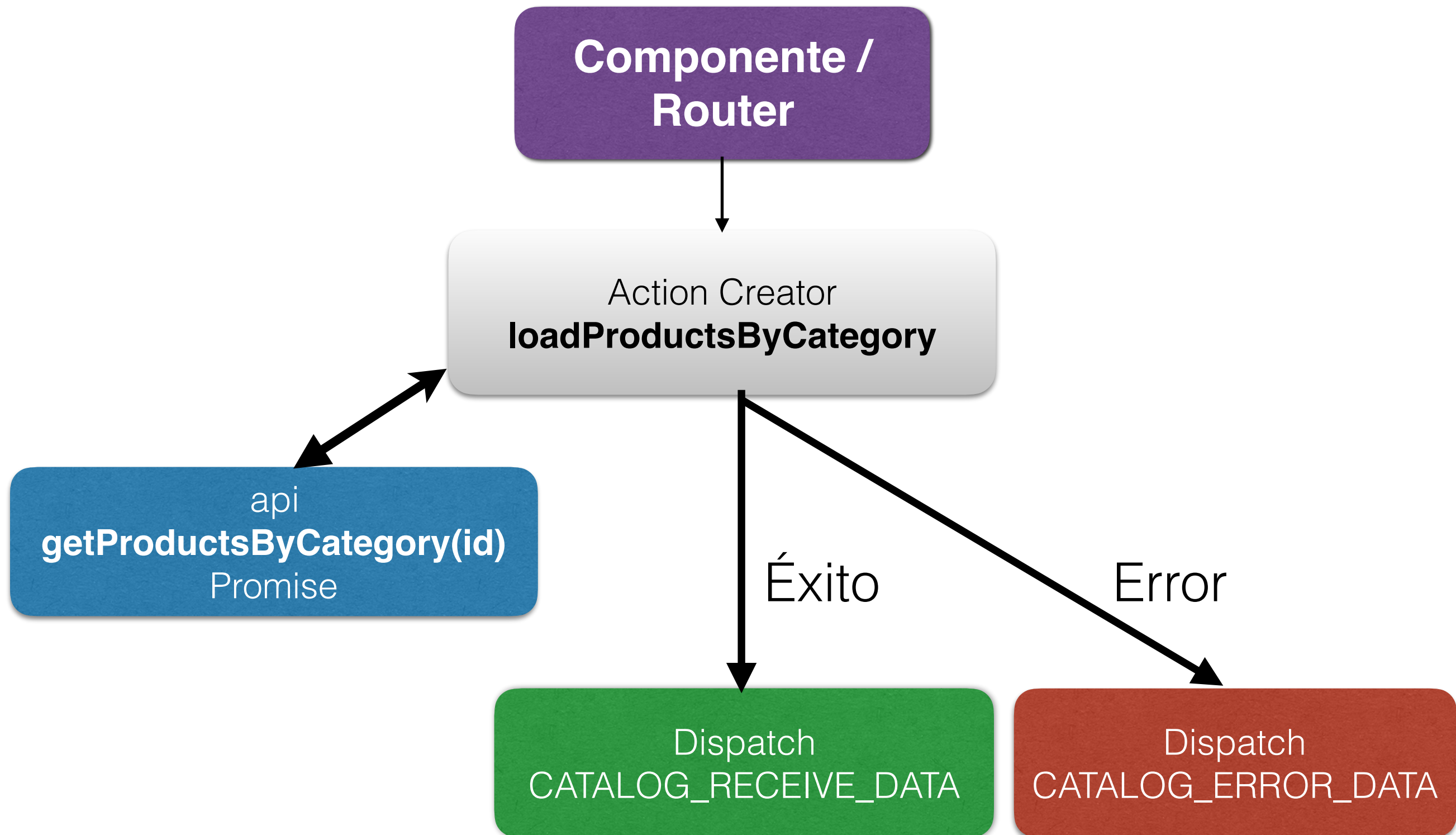
# Ejercicio - terminar Ecommerce con React Router

- Crear nuestro **history** para usarlo en Stores
- Incluirllo en **Route Store**
- Utilizar componentes **Link** donde queramos

# Integración con APIs REST

- Creamos un módulo/librería para peticiones HTTP que nos devuelva los datos o un error (Promesas)
- Llamamos a este módulo desde nuestros **action creators**
- Dependiendo del resultado de la operación, despacharemos una acción u otra

# Integración con APIs REST



# Integración con APIs REST

## Action Creator

```
export function loadProductsByCategory(categoryId) {  
  api.getProducts(categoryId)  
    .then(products => {  
    Dispatcher.dispatch({  
      type: CATALOG_RECEIVE,  
      categoryId,  
      products  
    });  
  })  
  .catch(err => {  
    // o dispatch de una acción de error  
    setPage('notfound');  
  });  
}
```

# Integración con APIs REST

## Módulo API

```
import request from 'superagent';

const API = {
  getJSON(url) {
    return new Promise((resolve, reject) => {
      //superagent returns JSON by default
      request
        .get(url)
        .end((err, response) => {
          return err ? reject(err) : resolve(response.body)
        });
    });
  },

  getProducts(categoryId) {
    return this.getJSON('/api/products/' + categoryId);
  }
};

export default API;
```

# Integración con APIs REST

## Store

```
__onDispatch(action) {  
  switch(action.type) {  
    case CATALOG_RECEIVE:  
      saveProducts(action.products, action.categoryId);  
      this.__emitChange();  
      break;  
    default:  
      return;  
  }  
}
```

# Ejercicio - cargar catálogo vía HTTP

- Usaremos **superagent**
- Crearemos nuestra librería en **src/lib/api**
- Crear **actionCreator**
- Llamar a **actionCreator** al entrar en la ruta del Catálogo
- No tenemos un back, pero podemos cargar archivos JSON desde el servidor (en **/dist**) para probar



# Ejercicio - catálogo con categorías

- Vamos a hacer la tienda ahora con dos categorías de productos: teléfonos y tablets
- Tendremos en `/` la lista de categorías, en **`/phones`** los teléfonos y en **`/tablets`** las tablets.
- Nuestra pseudo-API JSON nos ofrece:
  - **`/api/categories.json`** - devuelve la lista de categorías (1, phones; 2, tablets)
  - **`/api/products/1.json`** - devuelve los teléfonos
  - **`/api/products/2.json`** - devuelve las tablets

# Ejercicio - catálogo con categorías

- Necesitamos una nueva acción **loadProductsByCategory**, y un nuevo Store para guardar nuestras categorías: CategoryStore
- El componente **Home** ya incluye un menú de navegación entre categorías y estará conectado a CategoryStore
- En nuestro Router, tendremos que hacer que **Home** sea el componente “padre” en la ruta /
- Decidir cómo mapeamos el id de la categoría a una ruta y usar el componente **Catalog** en ella

# Ejercicio - catálogo con categorías

- Usamos el mismo componente Catalog para las dos categorías
- ¿Y cómo cargamos los productos apropiados a la categoría?
- Dos opciones:
  - **onEnter** en la ruta
  - **componentWillReceiveProps(nextProps)** en el componente

# React en el servidor

- Aplicaciones **isomórficas** o **universales**
- Significa compartir el código, UI, etc. entre cliente y servidor
- ¿Por qué?

# Problemas SPA en general

- **SEO**

Indexación con rutas hash es muy baja

- **Tiempo de respuesta efectivo**

Tiempo hasta que el usuario puede interactuar con la aplicación: cargar HTML, cargar bundle, instanciar aplicación, procesar ruta, pintar UI...

- ¿Solución? Devolver el HTML correcto **desde el servidor** de modo que antes de ejecutar la app, el usuario ya esté viendo el contenido que espera

# React en el servidor

- import ReactDOM from “**react-dom/server**”
- **renderToString**(<Component />  
Devuelve el HTML correspondiente a un elemento React. Podemos usarlo en el servidor para devolver el HTML de la ya listo.

Si después en el cliente llamamos a ReactDOM.render en un nodo que ya tiene este HTML, React **preservará el HTML y sólo añadirá eventos**

# React en el servidor

- Si podemos hacer un render de la app completa en el servidor y enviarlo como HTML inicial (junto con el bundle)
- React **no modificará el DOM** al lanzar la app, si coincide con lo que ya habíamos devuelto desde el servidor
- Así conseguimos: **SEO** y experiencia de usuario óptimos

# React en el servidor - retos

- Transpilar JSX en el servidor
- Reutilizar la configuración de rutas en cliente y servidor
- Hacer el **matching** de rutas en el servidor
- Si usamos Flux: cargar datos en el servidor y restaurar el estado en el cliente
- Vamos a intentarlo con **node.js** y Express



# JSX en el servidor

- **require(“babel-core/register”)(opciones)**  
Para transpilar con Babel bajo demanda, cuando hagamos require(xxxx)
- **React.createFactory(Componente)**  
Nos devuelve una función a la que llamar para obtener elementos de tipo *Componente* (en lugar de `<Componente />`)

De esta forma podemos hacer un **require** de un componente React, y pasarlo a ReactDOM.renderToString sin usar JSX en el archivo

# Rutas comunes

- **React router** permite definir las rutas en un archivo externo y luego pasarlas al componente **Router** como props (*routes*)
- Además, podemos configurar las rutas **sin** JSX, con un array de objetos JSX (las props en JSX son propiedades estándar):

# Rutas sin JSX

```
export const Routes = [  
  {  
    path: '/',  
    component: Layout,  
    indexRoute: { component: Home },  
    childRoutes: [  
      { path: 'page', component: Page }  
    ]  
  }  
]
```

# Route matching universal

- Usamos **match** de React Router para evaluar una ruta de forma “manual”

# Estado inicial Flux

- ¿Action creators en el servidor?
- Guardar el estado en window.\_\_\_\_\_xxxx
- Cargarlo mediante el Dispatch de una acción especial, por ejemplo “@@@INIT” y que todos los Stores lean el estado inicial

# Más allá de React y Flux “estándar”

- Datos inmutables
- Implementaciones Flux
  - Redux
- Relay, GraphQL

# Datos inmutables

- Ventajas
- immutable.js

# Ventajas inmutabilidad y React

- Comparación de valores, no de objetos
- **shouldComponentUpdate()** “gratis” y con objetos complejos
- Rendimiento



# Implementaciones de Flux

- La mayoría intentan reducir el “boilerplate” (Stores, Action Creators, Action constants, Dispatcher, conexión Componente <-> Store)
- Pero en el fondo son básicamente lo mismo
- Algunas plantean una idea diferente, en cambio...

# Redux

- <http://redux.js.org/>
- Un sólo “store”
- El Dispatcher envía el estado actual junto con la acción
- `onDispatch(state, action) -> state`
- Lo que escribimos son “reducers”