**BrowserRouter**

El primer componente que debemos ver a fondo y manipular será el BrowserRouter, este enrutador cuenta con diferentes propiedades para ser configurado:

* basename: configura la url base de todas las rutas.
* getUserConfirmation: recibe una función con la cual puedes validar si el usuario quiere dejar la pagina en la que se encuentra.
* forceRefresh: es un booleano, como su nombre lo indica en caso de ser verdadero se forzará a que el navegador recargue cuando se navegue.
* keyLength: un key es el id único que recibe cada movimiento en la navegación, keyLength se encarga de configurar la longitud de cada key y por defecto tiene una longitud de 6 caracteres.
* children: es lo que estará dentro de nuestro BrowserRouter.

Todo esto lo encontraras en su [documentación original de React Router](https://reacttraining.com/react-router/web/api/BrowserRouter).

Dentro de nuestro proyecto debemos envolver nuestros componentes principales dentro del BrowserRouter.

**Tipos de enrutadores**

React Router es una librería más que añadimos a nuestro stack, lo más básico que debemos aprender de React Router son sus distintos enrutadores:

1. **BrowserRouter**: Es el enrutador que quizá más tiempo utilices como frontend, usa el HTML5 history API lo que quiere decir que es el enrutador que nos da la posibilidad de cambiar las rutas en el navegador.
2. **HashRouter**: Funciona similar al BrowserRouter, pero usa un hash (**#**) al inicio de cada url.
3. **MemoryRouter**: Mantiene el historial de búsqueda en memoria y te sirve para realizar pruebas sin navegador. En este curso no haremos pruebas unitarias por lo tanto no veremos este enrutador.
4. **StaticRouter**: Nunca cambia de dirección, es perfecto para realizar Server Side Render.
5. **NativeRouter**: Es el router que nos servirá si queremos usar React Native, **NO** lo veremos en este curso. Es recomendable usar en su lugar [React Navigation](https://platzi.com/clases/react-navigation/" \t "_blank).

**Link y NavLink**

BrowserRouter no hará mucho si no esta acompañado de enlaces y rutas, empecemos hablando de los enlaces que se llaman Link y NavLink. Estos funcionan de manera similar a las anclas <a> </a> de HTML.

**Link** cuenta con las siguientes propiedades:

* to: similar al href de *<a>*, puede recibir un string indicando la ruta a donde va a mandar o bien recibir un objeto con: pathname, un string que representa la ruta a donde se dirige; search, un string que representa el query de una url; hash, un hash para poner en la url; y por último state, un objeto que representa un estado en la navegación.
* replace: similar a to, pero en lugar de añadir una nueva ruta al stack del historial de navegación, reemplaza la ultima ruta por la nueva ruta.
* innerRef: es una forma de obtener el elemento HTML del componente, funciona igual que el ref de React.

**NavLink** es una versión especial de Link, cuenta con varias características más poderosas como, por ejemplo:

* activeClassName: cuando se navegue a la ruta que dirija el NavLink, esta propiedad añadirá al className del componente el string que le pasemos.
* activeStyle: similar a activeClassName, pero con estilos en línea.
* isActive: es una función que se mandara cuando naveguemos a la ruta del NavLink.
* exact: recibe un booleano, sirve para marcar si dirige a una ruta exacta. Se vera a mayor profundidad cuando manejemos rutas.
* strict: recibe un booleano, sirve para marcar si dirige a una ruta estricta. Se vera a mayor profundidad cuando manejemos rutas.
* location: sirve para poder hacer la comparación de isActive con alguna otra ruta.

Vamos a implementar estos componentes dentro del componente Header, dicho componente lo encontraras en el sistema de archivos. Importamos y añadimos Header junto al componente de Home que se encuentra dentro del archivo app.js.

Recuerda importar **Fragment**, este es un componente que devuelve múltiples elementos. Fragment te permite agrupar multiples children sin agregar nodos adicionales al DOM.

**Route**

Aun no estas cambiando nada dentro de la interfaz, solamente se esta cambiando la url. Para poder cambiar la interfaz acorde a la url usaremos **Route**, algunas propiedades son:

* component: que componente quieres renderizar.
* path: indica la ruta en la cual va a renderizar el componente que le pases.
* render: es una alternativa a componente, puedes hacer un renderizado en forma de función como en los componentes de React.
* children: son los hijos o componentes que tenga anidado.
* exact: recibe un booleano, si le indicas que es verdadero solo hará match si la ruta coincide exactamente con la ubicación, no hará caso a ninguna sub-ruta.
* strict: recibe un booleano, si le indicas que es verdadero solo hará match si la ruta a la que te diriges es idéntica a la ruta del Route.
* sensitive: recibe un booleano, si le indicas que es verdadero activara el **case sensitive** para la ruta.

Vamos a cambiar el nombre del componente Home por Videos y añadiremos un nuevo componente Home que encontraras en el sistema de archivos, por último, configuramos sus componentes Route.

**Reto**: crea la ruta y el componente para el **contacto**, añade tus redes sociales y un poco de información acerca de ti.

# Parámetros de navegación

En esta clase vamos a aprender a personalizar un poco más nuestra url y pasar parámetros a través de esta. Haremos uso del componente **Link** y su propiedad to dentro de cada componente Media de nuestro proyecto.

# Páginas de contenido no encontrado 404

Otra cosa común y necesaria para tu aplicación es poder responderle al usuario cuando no encuentre algún tipo de contenido, responder a ese error 404.

Podemos hacer esto de manera fácil gracias a que, si al componente Route no le indicamos una ruta, esto lo tomara como la ruta por defecto a renderizar y en caso de no hacer match con ninguna ruta marcada entonces va a renderizar el componente.

# Redirect - Switch

Habrás notado que nuestro componente NotFound se está renderizando a la vez que el componente Home. Esto sucede porque la Route de NotFound está haciendo match con la de Home, para resolverlo debemos implementar Switch como componente padre de nuestros componentes Route.

**Switch** se encarga de solo renderizar el primer componente que haga match con la ruta que estés designando.

El componente **Redirect** nos ayudara para realizar un redireccionamiento en el navegador, sus principales parámetros son from y to que sirven para indicar de que ruta van a redirigir hacía que ruta van a realizar el redireccionamiento.

Para no entrar en problemas del Server Side Render añadiremos un nuevo NavLink dentro de nuestro Header para poder realizar el redireccionamiento.

# Prompt, validación antes de dejar la página

Vamos a implementar una validación antes de dejar la página en la que se encuentra el usuario. Esto sucede comúnmente en páginas que incluyan un formulario para evitar que el usuario se vaya sin enviar el formulario o dejarlo a medias.

Dentro de nuestro proyecto esto tiene sentido cuando estamos realizando alguna búsqueda. Para implementarlo usaremos el componente **Prompt** cuyos parámetros que recibe son when que recibe un booleano para indicar si muestra el mensaje del navegador y message que recibe un string que será el mensaje que reciba el usuario.

**Manipulando el historial**

Hasta el momento has aprendido a manipular las rutas por medio de componentes, en esta clase vamos a aprender a navegar de forma más programática.

Dentro de los componentes que renderizamos a través de Route encontramos en sus props un objeto llamado **history**, este objeto cuenta con multiples propiedades y métodos como:

* go: es un método que te permite ir a cierto momento en el historial de navegación, recibe como parámetro un número, dependiendo de la cantidad es cuanto avanzara en el historial y si es positivo o negativo será la dirección que tome.
* goBack: es un método que te permite navegar una pagina hacia atrás, funciona de forma similar a que si llamáramos a go(-1).
* goForward: es un método que te permite navegar una pagina hacia adelante, funciona de forma similar que si llamáramos a go(1).
* push: te permite añadir una nueva ruta al stack de navegación.

Dentro de nuestro proyecto vamos a añadir algunas propiedades de history en nuestro componente NotFound para poder navegar hacia atrás, navegar hacia adelante o bien ver un video aleatorio.

# Obteniendo el historial desde cualquier componente

El history es una propiedad que le llega a componentes que son renderizados por el componente padre Route, pero ¿qué pasa con los componentes que no son paginas o qué simplemente no forman parte de ninguna ruta?.

Dentro de nuestro curso tenemos un caso de ese estilo, el Header no forma parte de ninguna ruta por lo tanto no recibe las propiedades de history, location y match.

Existe un High Order Component llamado **withRouter** que te permite añadir estas propiedades.

# Configurando Webpack para server render

Llego el momento de realizar nuestro Server Side Render, en este módulo iremos configurando nuestro proyecto para que sea una Single Page Application universal ya que el código va a correr tanto en el cliente como en el servidor.

Lo primero que debemos hacer es configurar Webpack ya que requerimos que algunos archivos se exporten de forma 100% orientada a que los lea Node.

Por defecto, Webpack te va a exportar los archivos para navegador, para cambiar esto debemos escribir la propiedad target con el valor que queramos en este caso Node.

Por último, le indicamos a Webpack que nuestro archivo lo guarde en una carpeta distinta y añadimos al package.json un nuevo script para que ejecute Webpack.

# Creando una aplicación de Express

Vamos a crear un servidor básico con Express, para ello primero debemos instalarlo con el siguiente comando:

npm install express --save

Vamos a realizar algo llamado Backend for Frontend, de momento solo va a responder ante cualquier dirección con un texto que será una plantilla base de nuestro proyecto.

# StaticRouter

Los métodos utilizados de HTML5 para el BrowserRouter no existen en Node, por eso se debe utilizar StaticRouter.

Dentro de nuestro proyecto en el archivo app.js encontraremos varios componentes que solo funcionan del lado del navegador, necesitamos separar las cosas en un archivo de compilación para el cliente y uno para el servidor.

Vamos a realizar el refactor del archivo app.js para tener dos archivos y configurar Webpack para separar la compilación de archivos.

Una vez separados y compilados los archivos, dentro de nuestro servidor vamos a importar StaticRouter y el archivo compilado que no tiene nada relacionado al navegador para envolverlo dentro de StaticRouter.

**ReactDOMServer**

En esta clase vamos a terminar el renderizado en el server y además vamos a implementar babel-node para poder, entre varias cosas, manejar los imports de ES6.

Para instalar babel-node debemos correr los siguientes comandos:

npm **install babel-cli** –save-dev

npm **install babel-watch** –save-dev

Al momento de renderizar nuestros componentes en React hemos estado utilizando el método render de reactDOM, pero este método solo funciona en el navegador. Para poder renderizar en el servidor haremos uso de reactDOMServer, cuenta con cuatro métodos de los cuales dos se utilizan dentro de un stream, los otros dos métodos son:

* renderToString: te sirve para hacer server render y re-renderizar en el navegador.
* renderToStaticMarkup: este método te sirve si quieres hacer un server render que **NO** utilice un renderizado en el navegador.

Para nuestro proyecto usaremos *renderToString*.

StaticRouter cuenta con un parámetro necesario llamado location, este parámetro le va a indicar que ubicación se va a renderizar. En nuestro servidor le pasaremos a StaticRouter el parámetro location a través de la url que venga en el request.

Babel por defecto no va a entender React, entonces al momento correr babel-node debemos de indicarle que presets utilizar, tal como en el siguiente comando que añadiremos al package.json:

babel-**node** **src**/server.js –presets react,es2015,stage-2

Para evitar problemas con nuestros archivos estáticos de CSS e imágenes debemos configurar nuestro servidor de express añadiendo las siguientes líneas de código:

app.**use**(express.**static**(‘dist’));

app.**use**(‘/images’, express.**static**(‘images’));