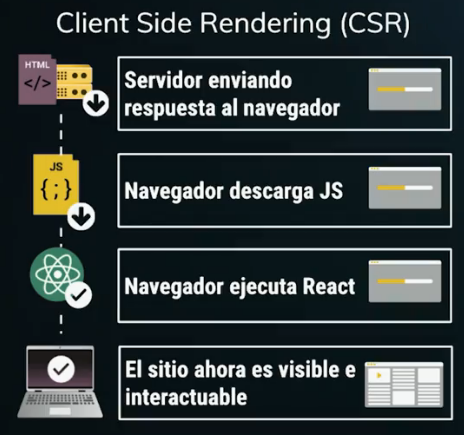
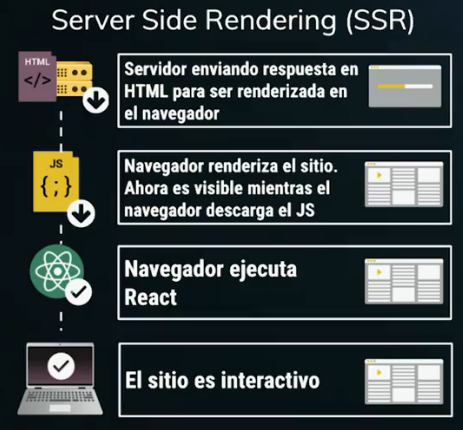
***Notas del Curso.***

**¿Qué es Server Side Rendering?**

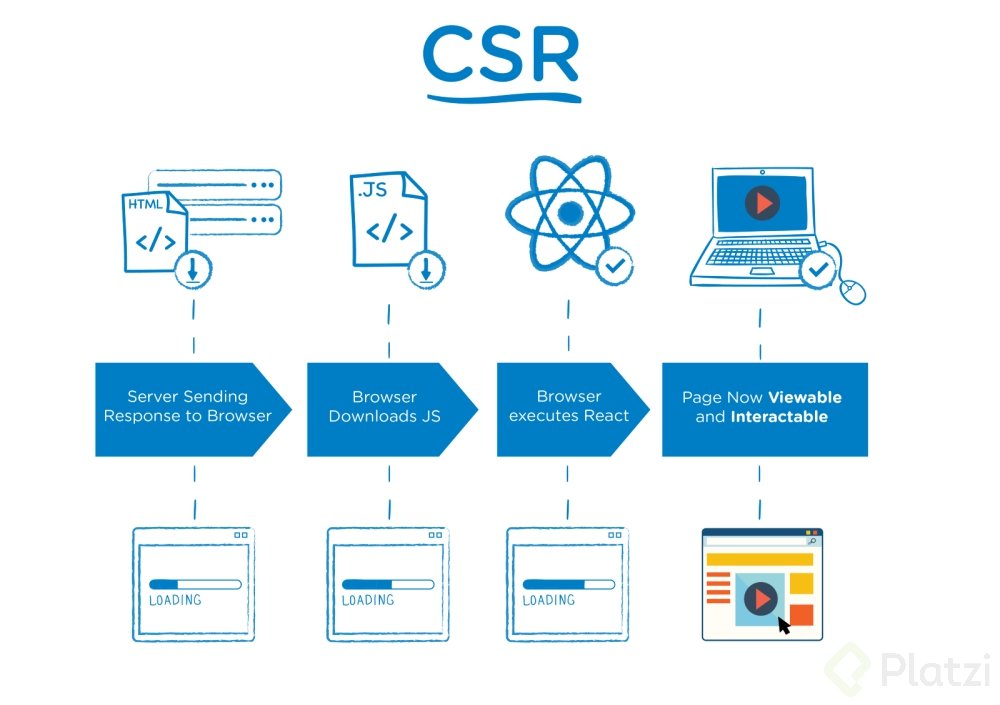
**Client Side Rendering (CSR)**  
Es lo habitual, el navegador descarga la aplicación y una vez procesada se presenta. Lo cual puede tomar algún tiempo en suceder y aburrir al usuario.

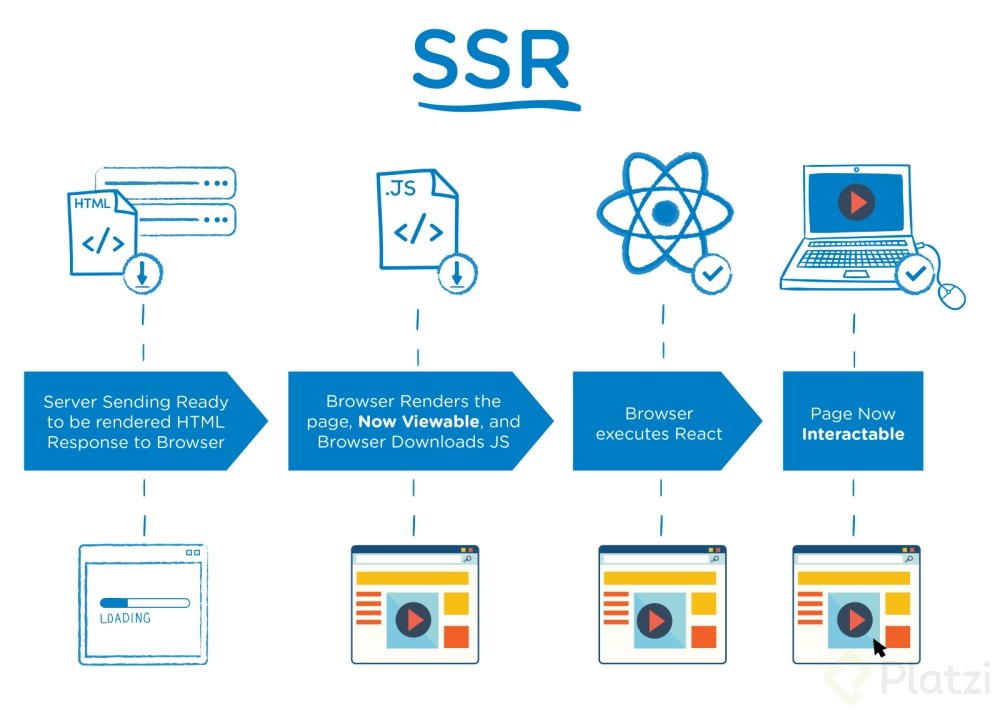


  
       
**Server Side Rendering (SSR)**  
Es un proceso similar, pero entre los datos que envía el servidor hay un pre renderizado de la aplicación en un String el cual se interpreta como HTML para ofrecerle una preview al usuario.

Las ventajas de SSR son:

* Hace que la primera carga sea más rápida.
* Mejora el SEO.
* Look and Feel --> No pasa de una pantalla en blanco a de repente toda la aplicación, sino que con un primer renderizado se va gradualmente haciendo interactiva.





# Integración de Webpack con Express.

**¿Qué es webpack-dev-middleware?**  
Es un middleware estilo express para ser usado con bundles de webpack que permite servir archivos emitidos de webpack. Solo debe ser usado para desarrollo

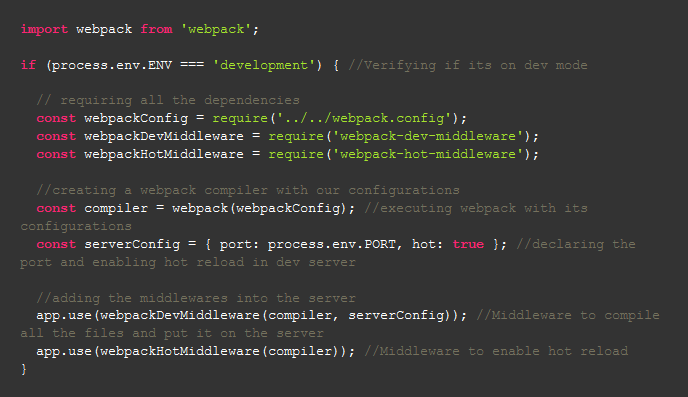
**¿Qué es Webpack Hot Middleware?**  
Webpack hot reloading usando solamente webpack-dev-middleware nos permite hacer hot reloading sin usar dev-server de webpack.

**Instalacion.**

Para poder integrar webpack con express debemos instalar las siguientes dependencias

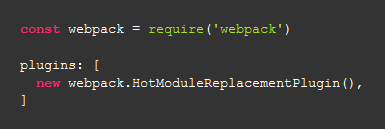


Luego debemos agregar el código siguiente a el archivo de configuración del servidor.

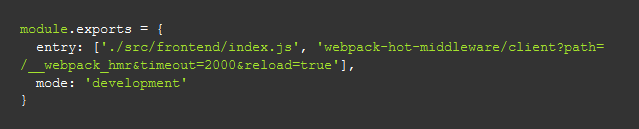


Después debemos agregar unas líneas extras en nuestra configuración de webpack

Primero agregamos un nuevo plugin:

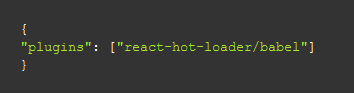


Luego, alteramos nuestro entry principal y agregamos en que modo se encontrara el servidor:



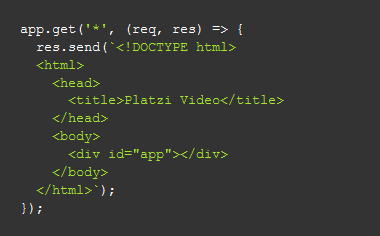
Con esto habilitamos el hot reload en webpack y le mencionamos explícitamente que nos encontramos en un entorno de desarrollo

Para finalizar, alteramos el archivo .babelrc agregando la siguiente linea

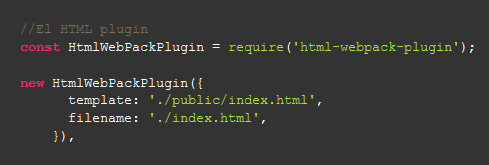


# Servir React con Express

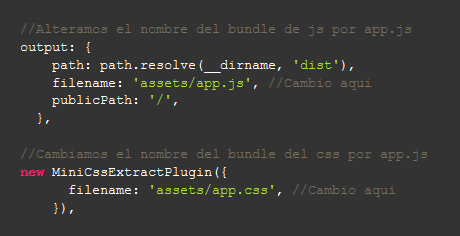
Para poder empezar a correr React sobre nuestro servidor de express debemos copiar el html que se encuentra en nuestra carpeta public y pegarlo en nuestra respuesta principal del servidor de Express. Esto se vera de la siguiente forma:



Luego en nuestra configuración de Webpack **eliminamos** las siguientes líneas de código:

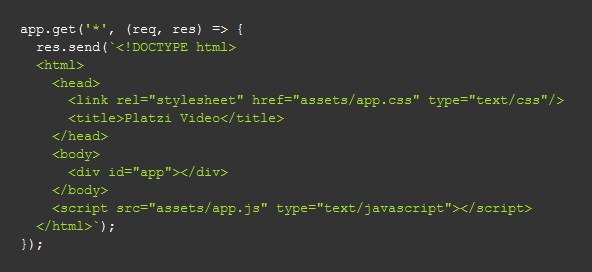


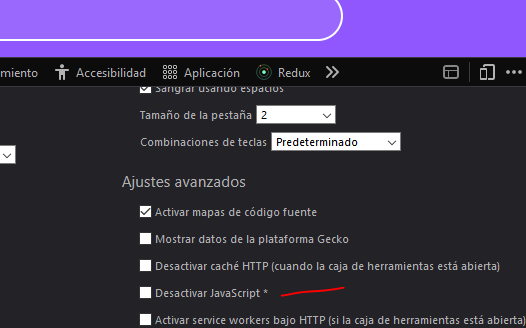
Posteriormente alteramos las siguientes líneas:



Por último, modificamos el HTML que estamos enviando desde el servidor, agregando las líneas que hacen el llamado al bundle del css y del js.

Quedando de la siguiente forma:



Ahora sí, podemos montar nuestro servidor nuevamente y comprobar el resultado

Nota: Para saber si realmente estamos haciendo SSR podemos desactivar Javascript desde las configuraciones de desarrollador del navegador.

# Abstrayendo React Router, creando history y haciendo initialState más accesible.

Primero debes abstraer React Router. Para ello, instalamos las siguientes dependencias

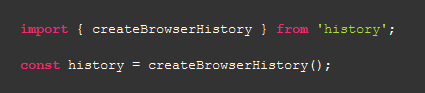


history permite mantener un historial mediante sesiones

react-router permite manejar rutas dentro de react

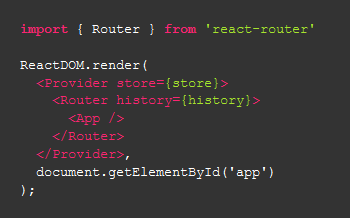
react-router-config permite añadir configuración extra al enrutador, en esta caso React Router

Después de instalar nuestras dependencias, debemos proceder a agregar las siguientes líneas al archivo index.js de nuestro frontend (src/frontend/index.js)



Con esto, creamos una historia en nuestro frontend logrando que la SPA sea la misma al momento de ir moviéndonos entre rutas.

Luego debemos agregar las siguientes líneas y modificar el ReactDOM.render justo como se muestra



Con esto, agregamos un router a toda nuestra app de React que cuenta con una history. Con esto podemos persistir datos a lo largo de las diferentes rutas

Ahora debemos crear el archivo de configuración de rutas que ocupará Express, este archivo lo creamos en la ruta src/frontend/routes y lo llamaremos serverRoutes.js

Dentro de este archivo, colocaremos lo siguiente:



Básicamente estamos reescribiendo el sistema de rutas pero en este caso, para que Express pueda interpretarlas correctamente.

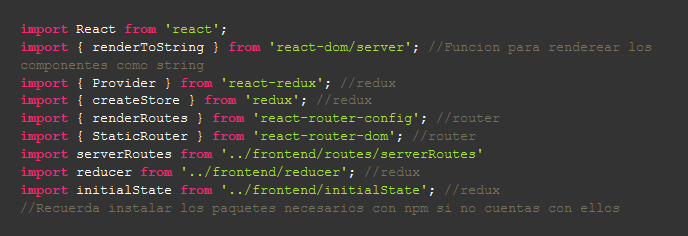
Por último, pero no menos importante, vamos a colocar el initialState en un archivo individual pues mas adelante accederemos a el en varias partes de nuestro código.

Para esto simplemente movemos el initialState que se encuentra en src/frontend/index.js a un nuevo archivo en esta misma ruta, lo llamaremos initialState.js Ya para finalizar, exportamos el archivo y lo importamos en el index.js ya que es requerido aquí:

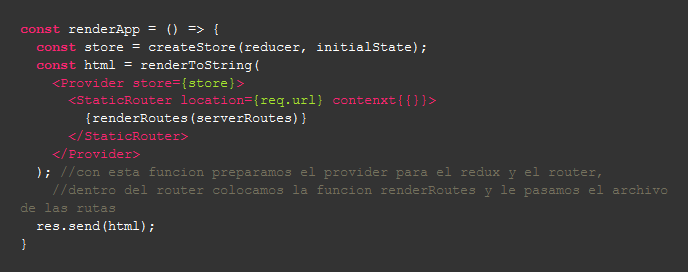
# Definición de la función principal para realizar el renderizado desde el servidor.

Para poder completar el SSR es necesario generar una función el cual va a enviar el html como respuesta para que el cliente sea capaz de renderizarlo.

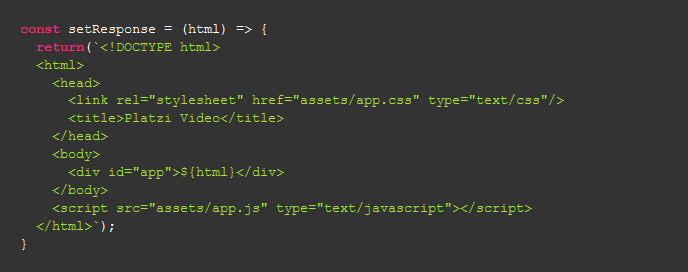
Para comenzar debemos importar las siguientes dependencias en el archivo server.js



Luego, vamos a definir una función dentro del mismo archivo que se encargara de preparar el las rutas y el redux



Después, creamos otra función la cual prepara la respuesta en HTML como string



Con esta función, enviamos un template con nuestros estilos y nuestro javascript cambiando el body dependiendo de que ruta usemos

Posteriormente alteramos en la ruta del principal del server app.get para enviar las funciones previamente hechas



Y para terminar, modificamos la respuesta que envía la función renderApp de la siguiente forma



Una vez realizado esto, debemos instalar una dependencia para evitar que el servidor cargue estilos de CSS en el servidor, por ello instalamos la siguiente dependencia y agregamos una linea a index.js dentro del server



Linea a agregar en server/index.js

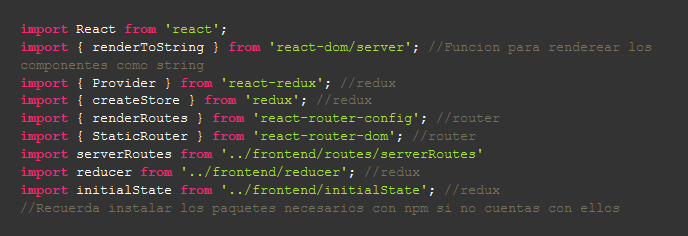


Y listo, podemos correr nuestra app y desactivar javascript en el cliente y todo debería funcionar sin mayor problema pues todo el renderizado está sucediendo en el servidor.

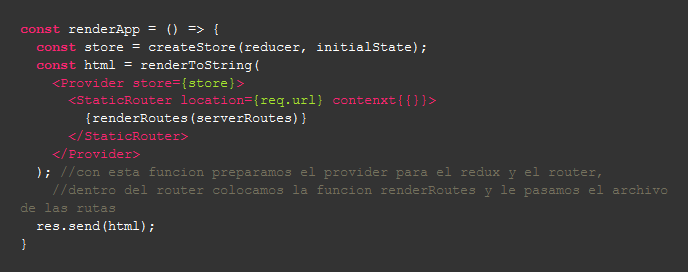
# Definición de la función principal para realizar el renderizado desde el servidor.

Para poder completar el SSR es necesario generar una función el cual va a enviar el html como respuesta para que el cliente sea capaz de renderizarlo

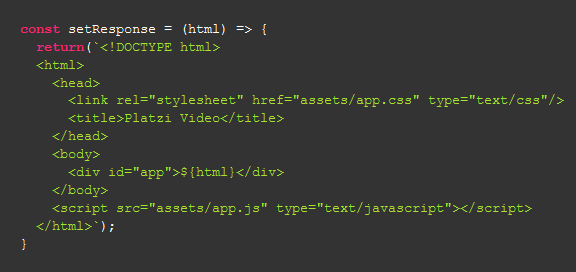
Para comenzar debemos importar las siguientes dependencias en el archivo server.js



Luego, vamos a definir una función dentro del mismo archivo que se encargara de preparar el las rutas y el redux



Después, creamos otra función la cual prepara la respuesta en HTML como string



Con esta función, enviamos un template con nuestros estilos y nuestro javascript cambiando el body dependiendo de que ruta usemos

Posteriormente alteramos el la ruta del principal del server app.get para enviar las funciones previamente hechas



Y para terminar, modificamos la respuesta que envía la función renderApp de la siguiente forma



Una vez realizado esto, debemos instalar una dependencia para evitar que el servidor cargue estilos de CSS en el servidor, por ello instalamos la siguiente dependencia y agregamos una linea a index.js dentro del server

npm install ignore-styles

Linea a agregar en server/index.js

require('ignore-styles');

Y listo, podemos correr nuestra app y desactivar javascript en el cliente y todo debería funcionar sin mayor problema pues todo el renderizado está sucediendo en el servidor.

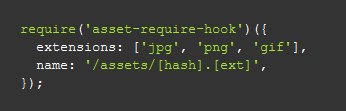
# Assets require hook.

Assets require hook es un hook que se encarga de importar todas nuestras imágenes en tiempo real cuando sean requeridas mediante SSR

Con esto, tendremos listo nuestro SSR en su totalidad. Para comenzar instalamos la dependencias



Luego agregamos la siguiente líneas al index.js del servidor



Y listo, tendremos cualquier tipo de imagen disponible en nuestro SSR app

**¿Por qué es una buena práctica agregar hashes a los nombres de nuestros archivos?**

para el cache del navegador, imagínate que haces build de tu app y existe una version 1 por asi decirlo de tu app.js y app.css, después decides hacer una modificación en estilos o en alguna funcion, cuando vuelvas a hacer build vas a volver a generar un app.js y un app.css. El problema viene cuando el navegador los carga, porque el navegador suele verificar si los archivos que pide ya los tiene, entonces puede pasar que el navegador al ver que ya tiene los archivos no los actualiza y no se ven reflejados los cambios. usar Hash en producción te permite obligar al navegador a que baje los archivos cada vez que cambian, porque cada vez que haces build el hash cambia y el navegador identifica que es un archivo diferente.

# Qué es el vendor file?

Antes a la carpeta que contenía todas las dependencias de un sitio web se le llamaba vendor, archivos de JavaScript y CSS de terceros se encontraba dentro de esta carpeta.

Con la llegada de procesadores de archivos como Webpack, en lugar de tener una carpeta vendor con las dependencias del proyecto, se empezó a compilar el código de las dependencias dentro de un archivo comúnmente nombrado vendor.js.

El archivo vendor.js contiene todas nuestras dependencias de un sitio web.

## **Importancia de optimizar**

Webpack trajo consigo una manera fácil y rápida de optimizar nuestros proyectos web. Una práctica que todo desarrollador web debe seguir es realizar **Code Splitting** ya que es raro modificar las dependencias de un sitio de manera frecuente.

Realizando Code Splitting podemos generar dos archivos:

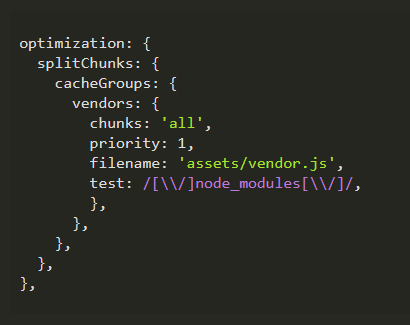
1. app.js: va a contener solamente el código de la aplicación.
2. vendor.js: dentro va a tener todo el código de las dependencias.

Esto trae grandes ventajas en producción pues podemos aprovechar el caché del navegador para alojar nuestro vendor.js y con ello cada vez que un cliente quiera acceder de nuevo a nuestro sitio, solamente tenga que descargar los archivos que hayan cambiado, o sea, el archivo app.js.

[Recuerda que es muy importante tener optimizado tu sitio web ya que más del 50% de los usuarios de internet se conectan a través de un dispositivo móvil y cerca del 53% de los usuarios abandonan un sitio que tarde más de 3 segundos en cargar.](https://platzi.com/blog/optimizar-tu-website/)

## **Webpack 4 al rescate**

Con la última actualización de Webpack se removió el plugin CommonsChunkPlugin y se hizo mucho más sencillo configurar nuestra estrategia de Code Splitting usando optimization.splitChunks.



* chunks: le indica a Webpack que chunks o fragmentos de código va a optimizar, sus valores pueden ser:
  + async: optimiza únicamente los archivos importados dinámicamente usando import().
  + initial: optimiza los archivos que son importados de manera normal usando import.
  + all : optimiza todos los archivos que se importen.
* priority: acepta valores numéricos y le indica a Webpack la prioridad de ejecución, en este caso se ejecuta primero.
* filename: le indica a Webpack el nombre del archivo resultante, en este caso genera el archivo en la carpeta assets con el nombre vendor.js.
* test: indica a Webpack donde buscar los archivos a optimizar, en este caso al ser dependencias las debe buscar en la carpeta node\_modules.

Como puedes ver el vendor file es muy importante para la optimización de tu sitio y gracias a Webpack se ha vuelto muy fácil generar este archivo,

**Tecnologías para SSR.**

Sin embargo existen otras tecnologías que vamos a poder utilizar para realizar el renderizado desde el servidor con Gatsby.js y Next.js.

**About Gatsby.js**

[***https://www.freecodecamp.org/news/why-you-should-use-gatsbyjs-to-build-static-sites-4f90eb6d1a7b/***](https://www.freecodecamp.org/news/why-you-should-use-gatsbyjs-to-build-static-sites-4f90eb6d1a7b/)

**About Next.js**

[**https://www.freecodecamp.org/news/the-next-js-handbook/**](https://www.freecodecamp.org/news/the-next-js-handbook/)

**Important Questions and Answers.**

