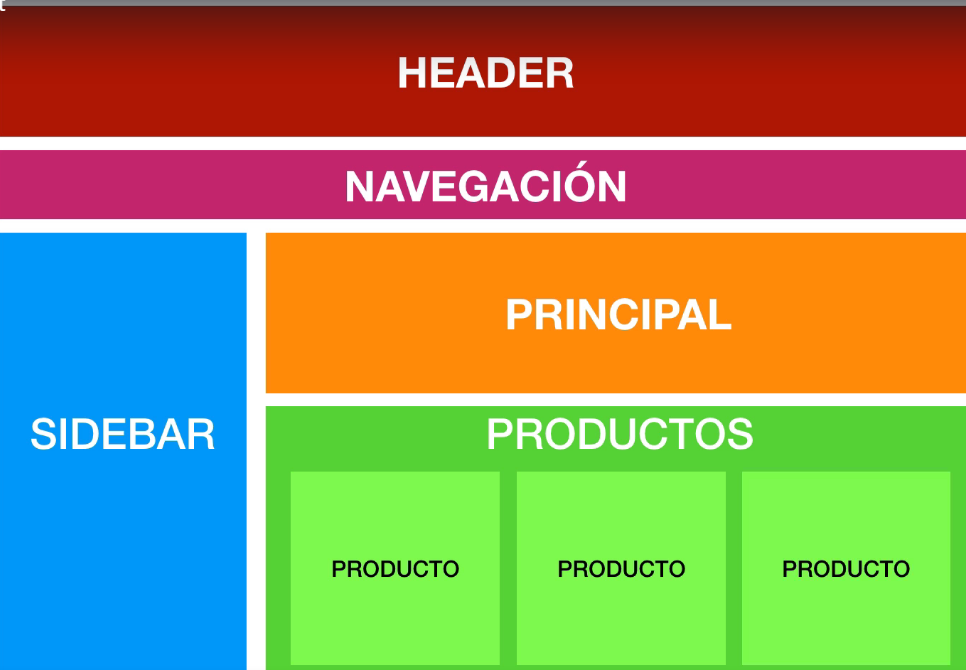
React maneja todo lo que se ve en el navegador, lo desarrollo Facebook. Corre en el cliente. Permite ordenar el código por componentes.

Componentes:



**Instalar:**

Se necesita node y npm

En la carpeta del proyecto se hace **npx créate-react-app prestamos** , donde prestamos seria el nombre del proyecto

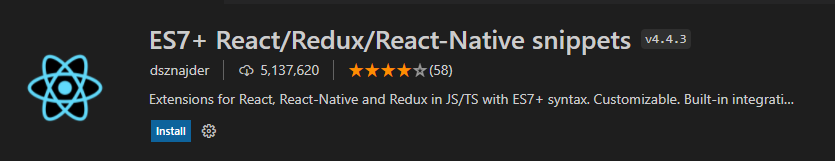
Instalar la extensión de Windows **react developer tools** de Facebook

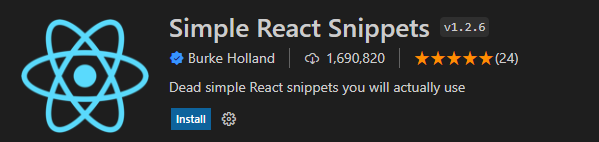
Cuando se termina de instalar todo le das a **npm start**

Con esa ip de on your network podes acceder desde cualquier dispositivo conectado a la misma red



Plugins:





De las carpetas que se crearon nosotros escribimos el código en src

En public solo se modifica para cargar algunas hojas de estilo de forma global en todo el proyecto

En <https://cdnjs.com/> buscamos skeleton y copiamos el link tag y lo pegamos en el head de index.html dentro de public.

También ahí buscamos normalize y copiamos el link tag y en el index hay que ponerlo ANTES que skeleton

En la carpeta SRC tenemos un index.js, que lo que hace este es importar el archivo app.js, una hoja de estilos y un index.css

App.js es el archivo principal que va a contener todos los demás componentes.

En el app para arrancar habría que borrar lo que hay para empezar a trabajar, quedaría algo así:

import './App.css';

function App() {

  return (

    <div className="App">

      <h1>Hola Mundo</h1>

    </div>

  );

}

export default App;

Creamos una carpeta en src llamada componentes

Adentro creamos nuestro primer Header.js, los componentes tienen que empezar con mayúscula.

Si escribimos imr te esribe la línea de import react

Imr:

import React from 'react';

En el componente hacemos esto, importamos react y app y exportamos Header, tiene que tener ese return

import React from 'react';

import App from '../App';

function Header(){

    return(

        <h1>Hola Mundo</h1>

    )

}

export default Header;

En el App.js importamos el header y lo llamamos <Header/> así

import './App.css';

import Header from './componentes/Header'

function App() {

  return (

    <div className="App">

      <Header/>

    </div>

  );

}

export default App;

Ya se deberia poder ver en el browser

El componente es reutilizable.

Siempre en el return hay que retornar un elemento, un elemento puede contener otros como un div con otras cosas dentro.

En vez de usar un div que envuelva todo podes usar un Fragment, que es como un div para envolver todos los componentes, dentro de ese si se puede llamar a varios:

import './App.css';

import Header from './componentes/Header';

import React, {Fragment} from 'react'; // es como crear un div pero que no se va a mostrar

function App() {

  return (

    <Fragment>

      <Header/>

      <Header/>

      <Header/>

    </Fragment>

  );

}

export default App;

**En React los datos fluyen desde el componente principal hacia los componentes hijos**. El principal si no me equivoco seria App.js

Por ejemplo si queres mandar algo desde el principal le pones en el componente un titulo:

function App() {

  return (

    <Fragment>

      <Header

        titulo= "Cotizador de prestamos" // esto puede ser cualquier tipo de datos

      />

    </Fragment>

  );

}

Y en el Header.js podes poner js entre la primera llave de la función y el return, notar que se le pasa props como parámetro.

Usa JSX que permite usar html y javascript

function Header(props){

    //en esta seccion se puede poner codigo standard de js

    console.log(props);

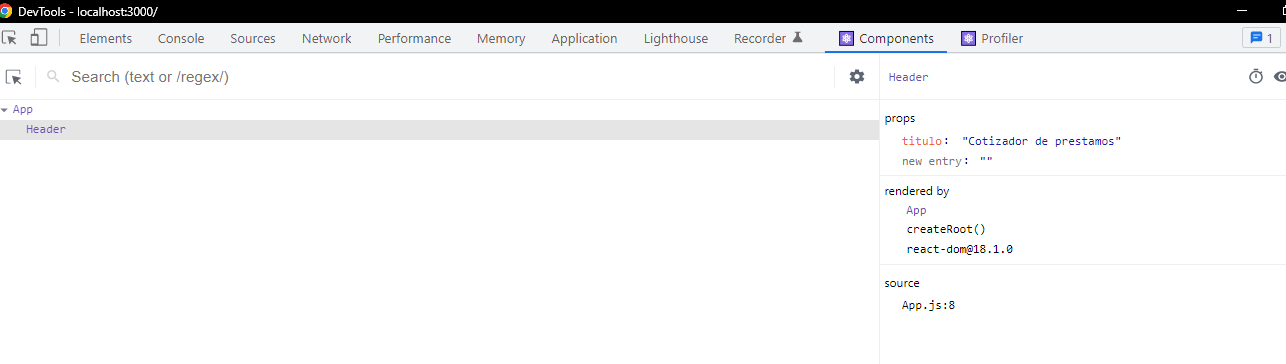
    return(

        <h1>Hola Mundo</h1>

    )

}

Al haber instalado la extensión de react en Chrome en vez de hacer un console.log podemos ver en la devTools la pestaña componentes y ahí va a aparecer el titulo que se envió



En el return del Header ver que se retorna html eso lo permite hacer jsx que permite mezclar html con js.

Para acceder entonces al props que se le envia, o mejor dicho para poder usar javascript dentro del html que se retorna se pone entre llaves:

function Header(props){

    return(

        <h1>{ props.titulo }</h1>

    )

}

Entonces ya con eso vamos a estar pasando al browser el valor de titulo que definimos antes.

Se pueden pasar varios parámetros, o atributos,

App.js

      <Header

        titulo= "Cotizador de prestamos" // esto puede ser cualquier tipo de datos

        descripcion= "Usa el formulario y obten una cotizacion"

      />

Header.js

En este caso vamos a tener que agregar el Fragment en el import, ya que vamos a retornar dos elementos

function Header(props){

    return(

        <Fragment>

            <h1>{ props.titulo }</h1>

            <p>{ props.descripcion}</p>

        </Fragment>

    )

}

Para no tener que pasar props, se puede hacer un destructuring en el parámetro y pasar titulo solo, quedaría así:

function Header({titulo}){

    return(

        <Fragment>

            <h1>{ titulo }</h1>

        </Fragment>

    )

}

**Otra forma de escribir componentes funcionales:**

Si escribis sfc y apretas tab ya te pone la estructura.

Esta seria la sintaxis y funcionaria igual que poniendole como function

const Header = ({titulo}) => {

    return (

        <Fragment>

        <h1>{ titulo }</h1>

        </Fragment>

     );

}

En este ultimo caso se puede dar por implícito el return ósea lo podemos borrar, entonces quedaría así:

const Header = ({titulo}) => (

        <Fragment>

        <h1>{ titulo }</h1>

        </Fragment>

);

Si es necesario cuando queres poner js entre el return y la llave x ej:

const Header = ({titulo}) => {

    const total = 1 + 1;

    return(

        <Fragment>

        <h1>{ total }</h1>

        </Fragment>

    )

};

En vez de class para definir una clase se tiene que usar className xq estas mezclando js con html y class es palabra reservada de js.



**Leer valores de un formaulario, useState:**

import React, {useState} from 'react';

Ese import lo hace en el componente, igual después se va a hacer en el padre, en el app

Todo lo que sea interactivo va a tener un state

Definimos uno para la cantidad, cada vez que se usa useState devuelve dos valores, le hacemos destructuring para sacar los dos valores , el primero es una variable que contiene el valor, y el otro una función que va a estar siendo usada para guardar el state que estamos creando. Podes dar un valor inicial en este caso le ponemos 0

Esto va entre el return y la llave de la función.

const [cantidad, guardarCantidad] = useState(0);

Si imprimis cantidad ahora se va a imprimir cero

{cantidad}

Analizemos esto:

const Formulario = () => {

    //Definir un state

    const [cantidad, guardarCantidad] = useState(0);

    const leerCantidad = (e)=>{

        guardarCantidad( parseInt(e.target.value));

    }

    return (

        <form>

        {cantidad}

          <div className="row">

              <div>

                  <label>Cantidad Prestamo</label>

                  <input

                      className="u-full-width"

                      type="number"

                      placeholder="Ejemplo: 3000"

                      onChange={ leerCantidad }

                  />

              </div>

En el input ponemos un onchange que va a tomar cada vez que se haga un cambio de valor en el input y va a ejecutar una función.

La función va a ejecutar guardar cantidad, que lo que va a ir haciendo es ir cambiando el valor de cantidad.

En el browser en los componentes vas a poder ver el state dentro de hooks del componente.

Se podría simplificar haciendo todo en una sola línea

onChange={ e => guardarCantidad(parseInt(e.target.value))}

Hay que recordar que en react los valores fluyen del padre al hijo pero nunca del hijo al padre.

Por eso sería mejor dejar en el padre:

import './App.css';

import Header from './componentes/Header';

import Formulario from './componentes/Formulario';

import React, {Fragment, useState} from 'react'; // es como crear un div pero que no se va a mostrar

function App() {

  const [cantidad, guardarCantidad] = useState(0);

  return (

    <Fragment>

      <Header

        titulo= "Cotizador de prestamos" // esto puede ser cualquier tipo de datos

      />

      <div className="container">

          <Formulario

            cantidad = {cantidad}

            guardarCantidad = {guardarCantidad}

          />

      </div>

    </Fragment>

  );

}

export default App;

y en el hijo formulario

import React from 'react';

const Formulario = ({cantidad, guardarCantidad}) => {

    return (

        <form>

        {cantidad}

          <div className="row">

              <div>

                  <label>Cantidad Prestamo</label>

                  <input

                      className="u-full-width"

                      type="number"

                      placeholder="Ejemplo: 3000"

                      onChange={ e => guardarCantidad(parseInt(e.target.value))}

                  />

              </div>

Así entonces desde el componente padre vos lo podes mandar al dato a distintos componentes.

**Leer los datos de un select:**

Se hace el onChange en el select, se usa el useState para los datos necesarios.

**Leer un submit y validar un formulario:**

En el form le ponemos un onSubmit esto es en el componente.

<form onSubmit={calcularPrestamo}>

Y creamos la función que se va a llamar con el onsubmit

    const calcularPrestamo = (e) => {

        e.preventDefault();

        //Validamos

        if(cantidad === 0 || plazo === ''){

            console.log('Hay un error');

        }

        //realizamos cotizacion

    }

Retorna también un p con un mensaje de error



Tiene que ir elform y el p dentro de un fragment

Se puede poner la condición con javascript

{ (error) ? <p className="error"> Todos los campos son obligatorios </p> : ''}

Y la función queda así

    //Definir state

    const [error, guardarError] = useState(false);

    //Cuando el usuario hace submit

    const calcularPrestamo = (e) => {

        e.preventDefault();

        //Validamos

        if(cantidad === 0 || plazo === ''){

            guardarError(true);

            return;

        }

        //Eliminar error previo

        guardarError(false);

        //realizamos cotizacion

    }

**Helpers: exportar funciones de otro archivo**

Se recomienda hacer un archivo de helpers con funciones que se puedan usar y reutilizar:

Crea el helpers.js en la carpeta src esto se hace para no cargar un componente tanto

En el helper le hizo esta función

export function calcularTotal(cantidad, plazo){

    let totalCantidad;

    if(cantidad <= 1000){

        totalCantidad = cantidad \* .25;

    }else if(cantidad > 1000 && cantidad <= 5000){

        totalCantidad = cantidad \* .20;

    }else if(cantidad > 5000 && cantidad <= 10000){

        totalCantidad = cantidad \* .15;

    }else{

        totalCantidad = cantidad \* .10;

    }

    console.log(totalCantidad);

}

En el componente que lo queres llamar se pone el import

import { calcularTotal } from '../helpers';

y lo ejecutamos pasandole los parámetros

calcularTotal(cantidad, plazo);

En el helper se podría hacer un return con el totalCantidad y después asignarlo a una variable en el componente.

**Carga condicional de componentes:**

En el app.js

  let componente;

  if(total === 0){

     componente = <Mensaje/>

  }else{

    componente = <Resultado/>;

  }

Después para imprimirlo:

          <div className='mensajes'>

            {componente}

          </div>

Este toFixed hace que solo se muestren 2 decimales

<p>Su pago mensual es de : $ { (total / plazo).toFixed(2)}</p>

En el index.js queda algo así, igual podemos separar los get que usamos, el get se ejecuta cuando se accede a la sección desde la url

import  express  from 'express';

const app = express(); // funcion para ejecutar express

//Definir puerto

const port = process.env.PORT || 4000; //la primera variable en local no va a existir, va a tomar el segundo

//una vez que se hace el deployment esa variable de arriba si va a existir

//express soporte, get, post, delete etc.

// la barra diagonal simboliza entiendo la pagina principal

// en el callback va el request (lo que yo envio) response es lo que express te envia

// el get se ejecuta cuando visitas una url

app.get('/', (req, res)=>{

    res.send('Inicio');

})

app.get('/nosotros', (req, res)=>{

    res.send('Nosotros');

})

app.get('/contacto', (req, res)=>{

    res.send('Contacto');

})

app.listen(port, ()=>{ // arranca el servidor

    console.log(`El servidor esta funcionando en el puerto ${port}`);

})

Para poner las rutas en un archivo aparte, creamos una carpeta en el root llamada routes, y dentro un archivo llamado index.js

No se puede crear otro app() ahí, así que tenemos que hacer una instancia de la aplicación

SEND imprime lo que se le pone ahí en pantalla

RENDER nos va a imprimir una vista

El archivo de las rutas quedaría así:

import  express  from 'express';

const router = express.Router();

//express soporte, get, post, delete etc.

// la barra diagonal simboliza entiendo la pagina principal

// en el callback va el request (lo que yo envio) response es lo que express te envia

// el get se ejecuta cuando visitas una url

router.get('/', (req, res)=>{

    res.send('Inicio');

})

router.get('/nosotros', (req, res)=>{

    res.send('Nosotros');

})

router.get('/contacto', (req, res)=>{

    res.send('Contacto');

})

export default router;

y al exportarlo ya Podemos usarlo en el index.js que esta en el raíz

lo importamos y agregamos el app.use

import  express  from 'express';

import router from './routes/index.js';

const app = express(); // funcion para ejecutar express

//Definir puerto

const port = process.env.PORT || 4000; //la primera variable en local no va a existir, va a tomar el segundo

//una vez que se hace el deployment esa variable de arriba si va a existir

//agregar router

app.use('/', router) // soporta get, post, put, patch y delete, lo que hace es que desde la pagina principal, agrega router y las rutas que estan dentro

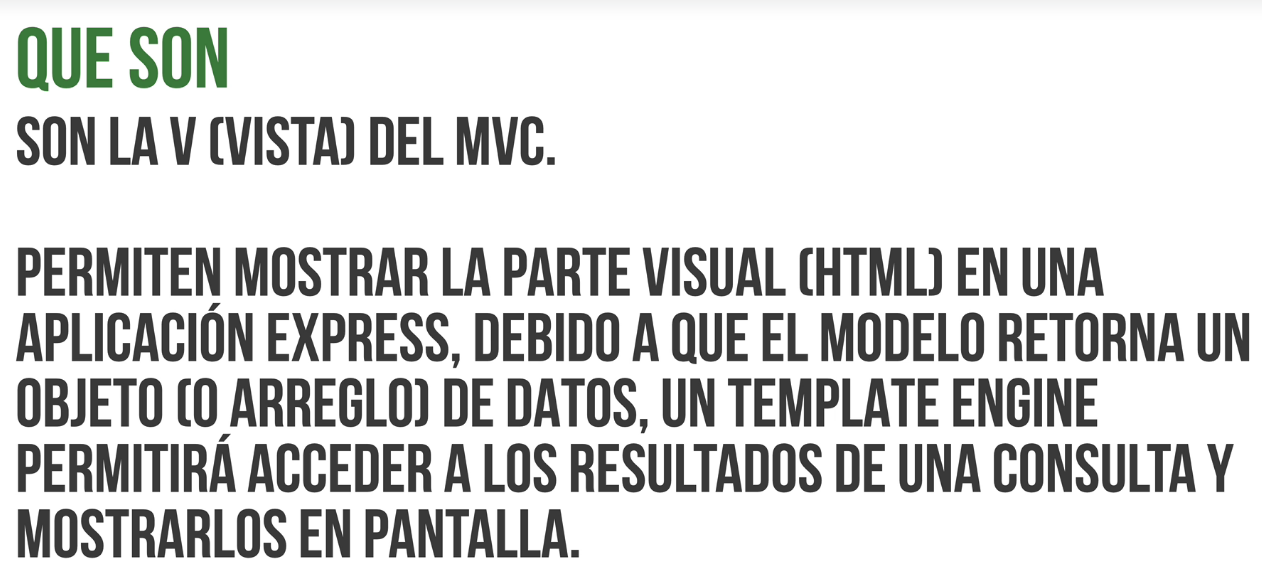
app.listen(port, ()=>{ // arranca el servidor

    console.log(`El servidor esta funcionando en el puerto ${port}`);

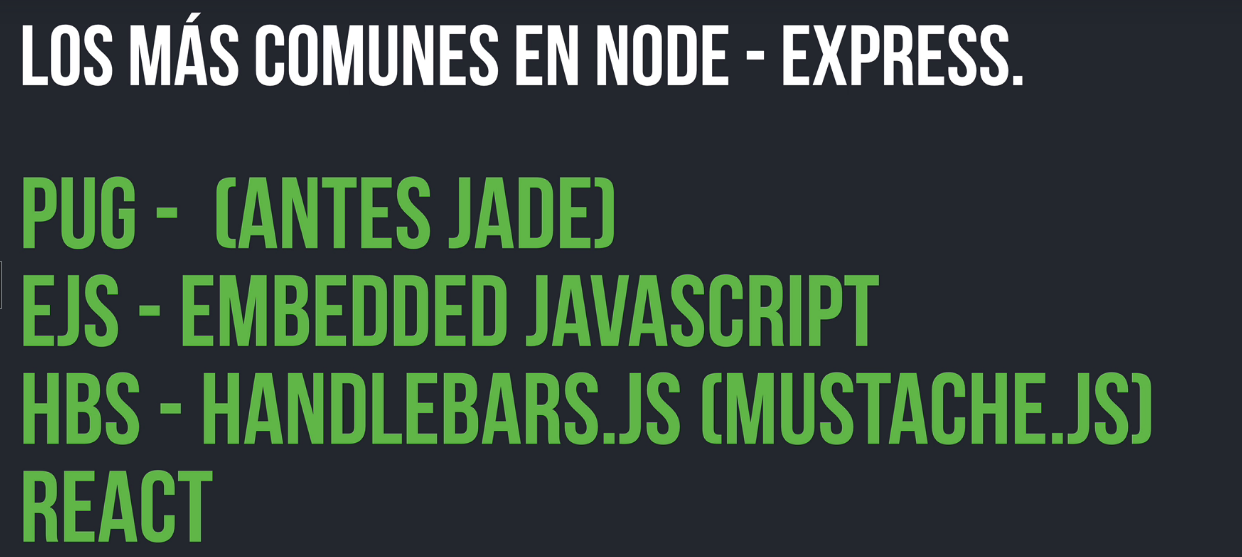
})

**Template Engine o motores de plantilla:**

Son la V del MVC(Model view controller) son las vistas

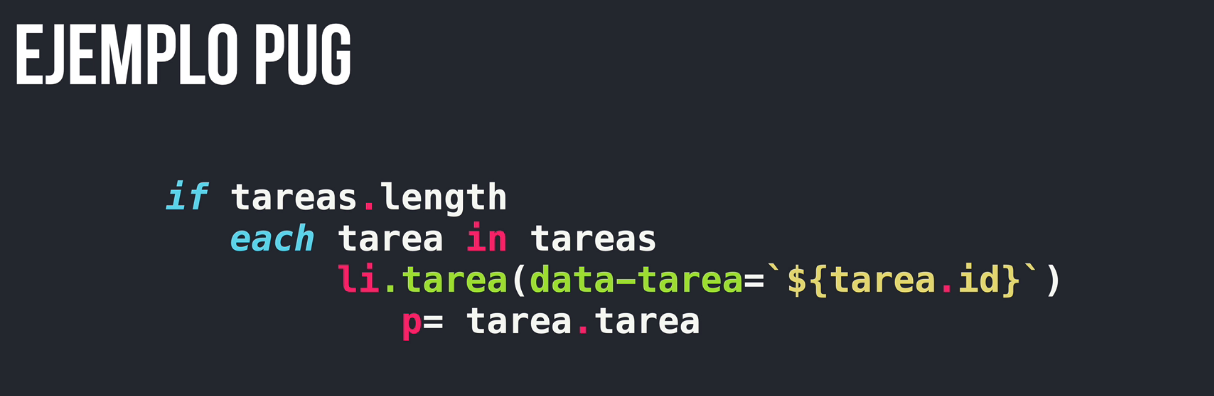


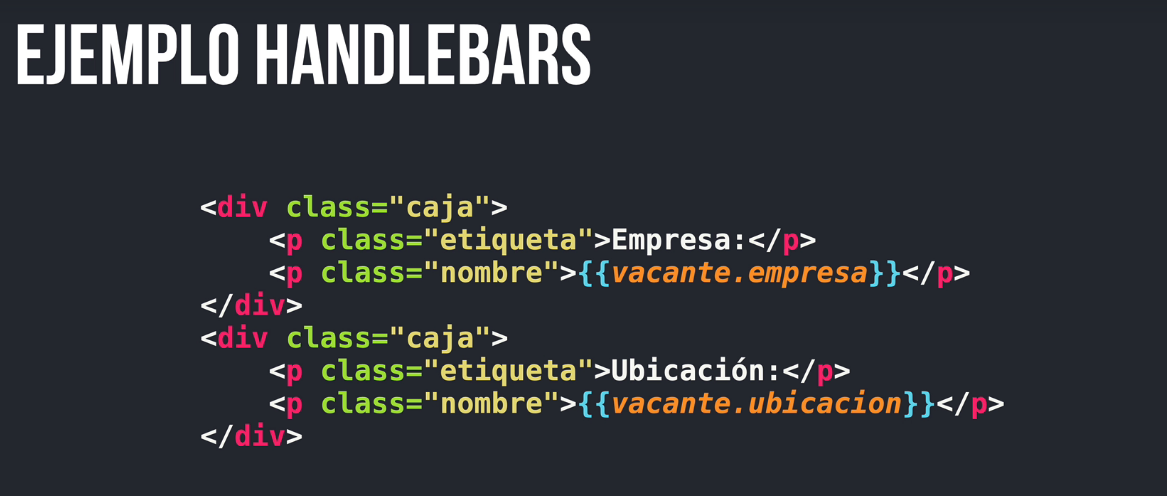
****

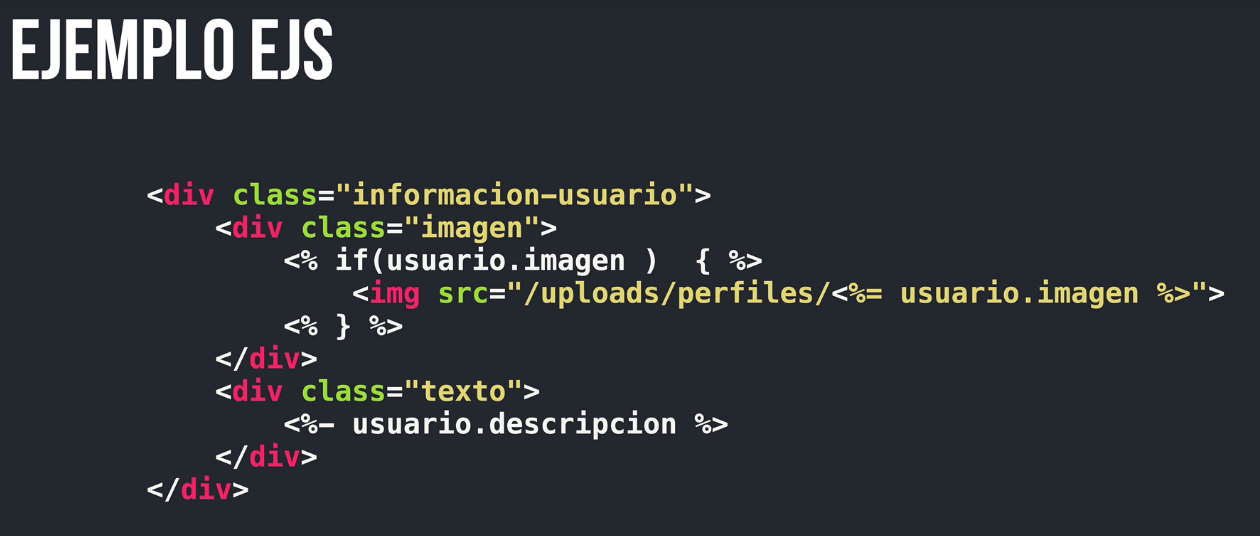
****

**Pug**

****

****

****

****

Hacemos en la terminal

Npm install pug

Agregamos en el index del raíz, después del puerto

//habilitar pug

app.set('view engine', 'pug');

Creamos una carpeta llamada views en el root y adentro un archivo llamado nosotros.pug

En pug esto es lo mismo

<h1>Hola</h1>

//Esto es lo mismo

h1 Hola

Ahora en el archivo de rutas cambiamos el send, por el render y le ponemos el nombre del archivo pug que creamos

router.get('/nosotros', (req, res)=>{

    res.render('nosotros');

})

Entonces si vas a la url y pones al lado del localhost /nosotros te tiene que traer lo que escribimos en el archivo pug

**Pasar valores hacia las vistas:**

Podemos crear una variable viajes y se le pasa después como objeto en el render

router.get('/nosotros', (req, res)=>{

    const viajes = 'Viaje a Alemania';

    res.render('nosotros', {

        viajes

    });

})

En el pug si a la etiqueta le pones un = ya espera una variable de javascript:

h1 Hola

p= viajes

También se puede hacer con numeral y llaves

h1 Hola

p #{viajes}

**Crear un layout principal o pagina maestra:**

Esto se usa porque todas las paginas que visitas tienen el mismo header o el mismo footer.

Dentro de la carpeta de vistas creamos una nueva carpeta llamada layout y dentro un archivo index.pug

En ese index ponemos la estructura de html, SON IMPORTANTES LAS INDENTACIONES, cuando indentas indicas que esta dentro de lo de arriba el elemento.

Creamos un block y le asignamos un nombre

doctype html

html(lang="en")

    head

        meta(charset="UTF-8")

        meta(http-equiv="X-UA-Compatible", content="IE=edge")

        meta(name="viewport", content="width=device-width, initial-scale=1.0")

        title Document

    body

        h1 Agencia de viajes

        block contenido

Y después en el de inicio le asignamos ese block y le hacemos un extend a el index

extends ./layout/index

block contenido

    p Inicio

En nosotros

extends ./layout/index

block contenido

    p Nosotros

el parrafo esta dentro del block por la indentacion

Con eso ya se debería estar mostrando

Si se lo asignamos a varias vistas al block contenido el h1 va a quedar fijo y va a ir variando lo que le asignemos en el block contenido

**Agregar hojas de estilo y archivos estáticos**

Creamos una carpeta llamada public, dentro de esa le metemos la carpeta css y dentro iran los css

En el index del app del raíz ponemos

//definir la carpeta publica

app.use(express.static('public'));

la referencia en pug se hace así a las hojas de estilo

        link(rel="stylesheet", href="/css/bootstrap.css")

        link(rel="stylesheet", href="/css/style.css")

desde las otras vistas podes llamar así a la imagen x ej

block contenido

    p Nosotros

    img(src="img/cupon.jpg")

**Header y navegación principal:**

Podemos crear un archivo header .php en la carpeta de layout, ahí le ponemos un h1 por ejemplo

h1 Agencia de viajes

y lo llamamos del index.pug con include y el nombre del archivo

    body

        include header

        block contenido