JS + NodeJS + Meteor + React

Un html importando un js

Primero, hagamos un archivo “prueba.**html**”, que nos traiga un archivo “prueba.**js**”. Normalmente lo de importar el archivo de js va dentro de la sección del head, pero lo pongo más abajo para un ejemplo de más adelante.

```
<head>
   <meta charset="utf-8">
   <title>Pruebilla</title>
</head>
<body>
  <script src="prueba.js"></script>
</body>
```

Crear un objeto

Vamos a ver que JS tiene muuucho en común con wollok. Por ejemplo, podemos hacer esto en nuestro archivo prueba.js:

```
const pepita = {
   energía: 100,
   volá(){this.energía -= 10},
   comé(){this.energía +=15},
}
```

Una vez hecho esto, vamos a abrir nuestro archivo “prueba.html” en el explorador. Abrimos chrome, y ponemos esto en donde van las direcciones:

file:///C:/Users/… la ruta en donde hayas guardado el archivo .../**prueba.html**

Eso nos abre una página “en blanco”. ¿Por qué? Porque en el html no creamos ni un botón, ni nada. Solo importamos un JS y nada más.

Apretemos “F12” (o vayamos a opciones, más herramientas, herramientas de desarrolladores). Luego, vayamos a “Console”.

Esto permite interactuar con el JS que hayamos cargado. Por ejemplo:

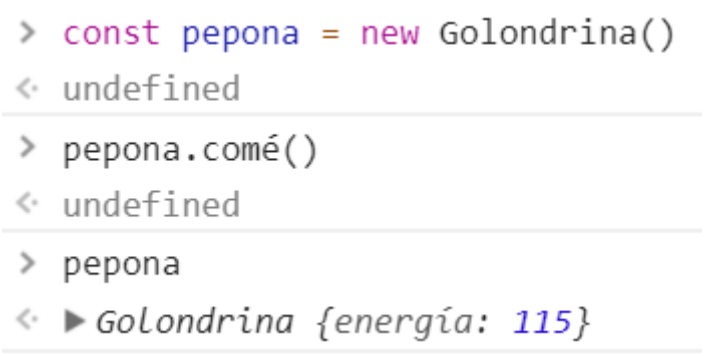
```
> pepita.volá()
< undefined
> pepita.energía
< 90
> pepita
< ▶ {energía: 90, volá: f, comé: f}
```

Noten que dice que “volá” es una “f”. Sí, acá los métodos son funciones pero con una particularidad: Tienen “enlazado” un objeto, para que adentro de ellas puedan saber quién es “uno mismo” (en wollok sería “self”, acá se llama “this”).

Crear una clase

```
class Golondrina {
   constructor(){
       this.energía = 100
   }
   volá(){this.energía -= 10}
   comé(){this.energía +=15}
}
```

Claramente, podemos instanciarla e interactuar con los objetos:



¿Qué es "undefined"? Es un valor primitivo que representa a la nada. Toda propiedad sin incializar, tiene asignado "undefined". Y todo método o funcionalidad que no retorna nada, retorna undefined.

Fíjense cómo nos muestra a pepona en la consola. Dice que es una **Golondrina**, y que su energía es **115**. No habla de los métodos que tiene, porque eso se deduce sabiendo que es una Golondrina.

Igual, recordemos que "Golondrina {energía: 115}" es solo una forma cheta de mostrarla en la consola. Eso no es ni el objeto real, ni código que podamos escribir. Es solo una *representación* amigable del objeto.

Algunas particularidades

Enviar un mensaje sin paréntesis

¿Qué pasa si ejecutamos esto?

`pepona.volá`

Humm… En wollok, se rompería, ya que no estamos poniendo paréntesis al mensaje. Pero acá:

Sí, **devuelve** una **función**. Pero ojo, porque recordemos que ahí adentro dice "this", la función solita no sabría quién es.

O sea, podemos decir "pepona.volá()" pero no podríamos tener a la función aislada y pedirle que se ejecute:

```
> const funciónDeVolar = pepona.volá
< undefined
> funciónDeVolar()
▶Uncaught TypeError: Cannot read property 'energía' of undefined
    at volá (prueba.js:12)
    at <anonymous>:1:1
```

Propiedades inventadas

¿Qué pasa si consultamos o asignamos una propiedad nueva?

```
pepona.sarlompa = 17
```

¡Funciona!

Porque acá las propiedades son dinámicas. Podemos crearlas y destruirlas cuanto queramos.

Incluso podemos crear métodos:

```
> pepona.dormíUnaSiesta = function(){this.energía = 1000}
< f (){this.energía = 1000}
> pepona.dormíUnaSiesta()
< undefined
> pepona.energía
< 1000
```

Eso da un gran poder y flexibilidad, pero a la vez, es muy fácil caer en **malas prácticas de programación**, y además, es muy **fácil equivocarse**.

Por eso aparecen TypeScript (un lenguaje que transpila a JS), o herramientas como PropTypes para recibir parámetros de cierto tipo cuando usamos React, etc.

Pero más allá de eso, sigamos encontrando similitudes con wollok.

Colecciones

Podemos crear WKO's, clases, constructores, valores constantes y variables (al menos dentro de una función), heredar, pero además, como si fuese poco, también se parecen mucho las formas de manejar las colecciones.

Acá también se puede hacer un "**new Set()**", y un "**new List()**", y además, podemos escribir las listas de forma bonita "**[]**".

Y toda colección entiende **filter**, **map**, **forEach**, **some** (equivalente a "any"), **every** (adivinen), **reduce** (foldr o foldr1 dependiendo de cómo se use), etc.

Pero de esto ya habíamos hablado un poco en el paradigma funcional, así que sigamos avanzando.

Hacer un chat, que sepa renderizarse

Para no perder tiempo, sigamos con los mismos archivos “prueba.html” y “prueba.js”, pero sepamos que vamos a hacer un chat.

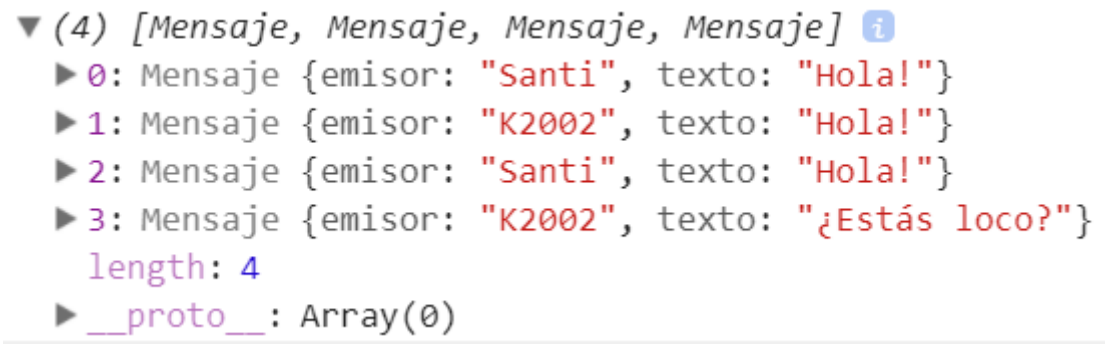
¿Qué chat? Uno que tenga **una sola conversación**. Un chat de este mismo curso.

El chat, tendrá varios mensajes. De cada mensaje, nos interesa el emisor, y el texto del mismo.

Creando un chat

```
class Mensaje {
   constructor(emisor, texto){
       this.emisor = emisor
       this.texto = texto
   }
   // Si lo necesitáramos, podríamos crear métodos específicos del mensaje:
   contenés(ciertaPalabra){return this.texto.includes(ciertaPalabra)}
   longitud(){return this.texto.length}
}
class Chat {
   obtenéMensajes(){
       return [
           new Mensaje("Santi", "Hola!"),
           new Mensaje("K2002", "Hola!"),
           new Mensaje("Santi", "Hola!"),
           new Mensaje("K2002", "¿Estás loco?")
       ]
   }
}
console.log((new Chat().obtenéMensajes()))
```

Al recargar el explorador, nos aparece esto en la consola:



Por si quedaban dudas: Sí, “**console**” es un WKO, que sabe interactuar con la consola. También existen en otros lenguajes.

Y acá, en JavaScript, también tenemos “WK**F**” (funciones bien conocidas), como “alert”, que muestra un mensaje en una ventanita que sobresale a la pantalla.

Responderlo en una página

El usuario difícilmente entrará a la consola para ver los mensajes. Tenemos que mostrarlo en nuestra página, en el html. Así que vamos a hacer que el código de JS lo “enchufe” dentro de un elemento de HTML. Para esto, hagamos esa “sección” en donde lo vamos a enchufar:

```
<head>
   <meta charset="utf-8">
   <title>Pruebilla</title>
</head>
<body>
  <div id='secciónParaRenderizar'></div>
  <script src="prueba.js" charset="UTF-8"></script>
</body>
```

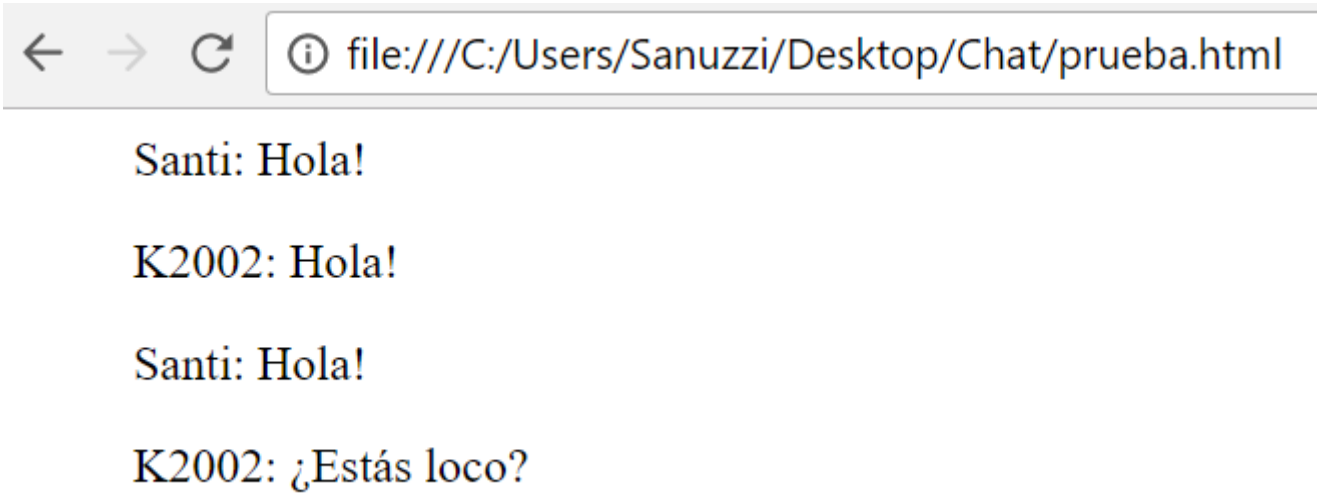
Ahora sí, vamos a “renderizar” (mostrar gráficamente) en la pantalla del explorador los elementos nuevos de HTML para cada mensaje. Así que estaría bueno que tanto un Mensaje, como el Chat entero, sepan “renderizarse”.

Y por último, vamos a hacer un objetito para agarrar esos componentes, y enchufarlos efectivamente en nuestra ventana del explorador.

```
class Mensaje {
   constructor(emisor, receptor, texto){
       this.emisor = emisor;
       this.receptor = receptor;
       this.texto = texto;
   }
   contenés(ciertaPalabra){return this.texto.includes(ciertaPalabra);}
   longitud(){return this.texto.length;}
   renderizate(){return "<p>" + this.emisor + ": " + this.texto + "<p>";}
}
class Chat {
   obtenéMensajes(){
       return [
           new Mensaje("Santi", "K2002", "Hola!"),
           new Mensaje("K2002", "Santi", "Hola!"),
           new Mensaje("Santi", "K2002", "Hola!"),
           new Mensaje("K2002", "Santi", "¿Estás loco?"),
       ];
   }
   renderizate(){
       return "<ul>" + this.obtenéMensajes().map((mensaje => mensaje.renderizate())).join("")
+ "</ul>";
   }
```

```
}
const objetoMágicoRenderizador = {
   renderizá(componente){
       const secciónDeNuestraPantalla = document.getElementById('secciónParaRenderizar');
       secciónDeNuestraPantalla.innerHTML = componente.renderizate();
   }
}
objetoMágicoRenderizador.renderizá(new Chat());
```

Recargamos la página y…



¡Funciona! ¡Es “visible”!

Peeeero esto sigue apestando. Solo estamos levantándolo del disco directamente. No es un sistema web real en donde un servidor nos devuelve cosas, a menos que...

¡Hagamos un servidor!

Esto lo vamos a hacer en NodeJS. Es muy fácil instalarlo. Y recién acaba de salir del horno la versión que viene con npm5 por defecto.

¿Qué es npm? Es un gestionador de paquetes para las aplicaciones que hagamos con node. Y npm5 vuela, es mucho más rápido que sus antecesores.

Y pará… ¿Qué es node? Citando a la página oficial:

> Node.js® es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome. Node.js usa un modelo de operaciones E/S sin bloqueo y orientado a eventos, que lo hace liviano y eficiente. El ecosistema de paquetes de Node.js, npm, es el ecosistema mas grande de librerías de código abierto en el mundo.

Deaj, “librerías”... Malditos traductores. Bueno, volviendo: Sirve para poder ejecutar JS por fuera de un explorador, y eso te permite crear bocha de cosas. Seguimos con esto al final del apunte.

Una aclaración: Hay dos posibilidades para instalar NodeJS. La estable y mantenida por más tiempo, y la que tiene las últimas características. Por ahora, recomiendo la **estable** (**8.9.0**). Si la de las últimas características se estabiliza más y tiene funcionalidades extra muy necesarias, hagan switch, pero por ahora no conviene.

¡Sigamos! Abramos una terminal[1], y vayamos a la carpeta donde estén los archivos de antes. Luego ponemos:

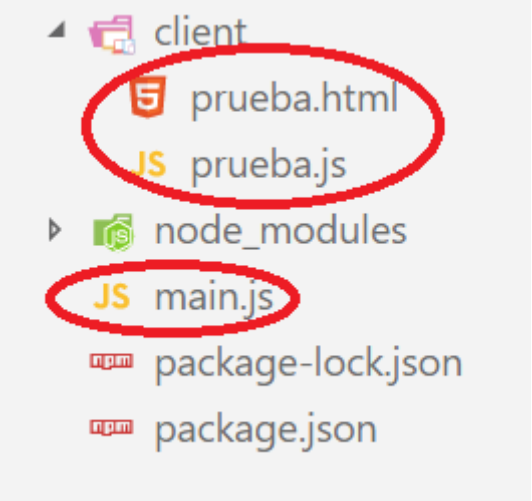
```
npm init
```

Y todo "enter" para hacer esta prueba más rápida. Luego, agreguemos una biblioteca muy útil para sistemas web:

```
npm install express
```

Ahora, vamos a crear una carpeta llamada "client". Y vamos a mover los archivos "prueba.html" y "prueba.js" ahí adentro. Así todo lo "público" accesible para cualquier usuario está metido en una sola carpeta.

Luego, afuera, hagamos un "main.js"[2] para ejecutar el código principal del servidor para que pueda encenderse. Debería quedarnos así:



Luego, en main.js, vamos a codificar lo siguiente:

```
const express = require("express")
const app = express()

app.get("/", (pedido, respuesta) => respuesta.sendFile(__dirname  + "../client/prueba.html"))

app.use(express.static('client'))

app.listen(3000, () => console.log("Corriendo!!"))
```

¿Qué hace ese cacho de código?

- Importa la biblioteca "express". Sirve para iniciar aplicaciones web, y para configurar las rutas (URLs) a las cuales podemos acceder, y qué se responde en cada caso.
- En la segunda línea, notamos que la biblioteca es una **función**. Al ejecutarla, nos devuelve un objeto con el cual configuramos rutas e iniciamos el servidor.

---

[1] Amíguense con la terminal (por más que usen el cmd de windows y sea horrible). Tienen que saber abrirla y moverse entre carpetas **ya**

[2] Podrían llamarlo como quisieran siempre que termine en "js". Podrían ponerle main.js, index.js, server.js, cualquier cosa.

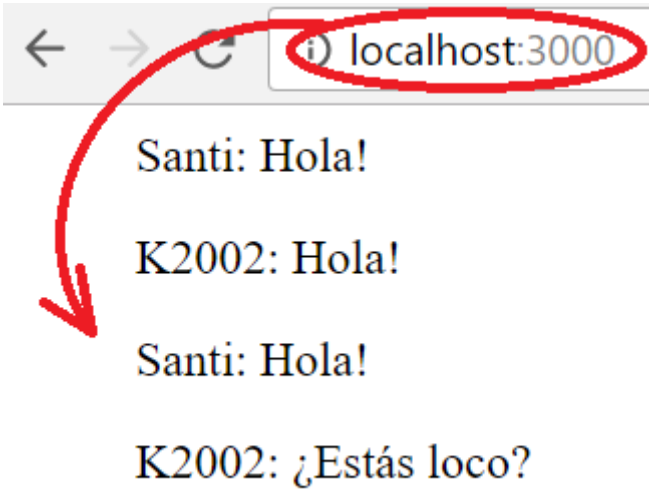
- Decimos que al entrar al “localhost:3000**/**” (es decir, la ruta “raíz”), se devuelva lo que está en el archivo prueba.html.
- Indicamos que todo lo que está en la carpeta “client” es accesible desde afuera. Gracias a esto, el usuario podrá traerse los archivos “prueba.html” y luego “prueba.js”.
- Iniciamos el servidor en el puerto 3000. Y cuando inicie, se ejecutará una función que imprima “Corriendo!!” en la consola.

Luego, ejecutamos lo siguiente en la terminal:

node **`main.js`**

En la consola, debe decir “Corriendo!!”.

Y si vamos al explorador y ponemos esta url, la aplicación nos responde con el HTML:



¡El server está vivo!

Direcciones web y DNSs

Vamos de cero. Cuando escribimos “http://google.com”, en realidad, le pedimos a un servidor llamado “DNS” que nos diga el nombre real de google.

Y él nos dice: “google” en realidad se llama “172.217.30.142”. ¿No me creen? Pongan esos números en su explorador.

Bueno, ese número es la “Dirección IP” (es decir, la dirección usada para “Internet Protocol”, que es una forma de comunicación entre computadoras).

Y ese número no alcanza para conectarnos con alguien. También tenemos que indicar cierto “puerto”. Si no lo escribimos, se asume que es el 80. Si queremos otro específico, lo ponemos a mano.

Nosotros queríamos levantar un servidor en nuestra propia PC. Así que queríamos comunicarnos con una computadora que sea… Nosotros mismos.

Por convención, “localhost” o “127.0.0.1” significan exactamente eso. Así que escribimos eso, indicamos el puerto 3000, y listo. Hablamos con nuestro servidor :D

SPA + Meteor + React

SPA: Single Page Application. Pueden googlear estas cosas. Básicamente, se trata de que todo lo que suceda en la página vaya alterando ciertas partes de la misma, en vez de recargar toda la página a cada rato. Hace que la interacción sea más rápida y fluida, y por ende, da una mejor experiencia al usuario.

Meteor: Sirve para hacer aplicaciones web y mobile a las chapas, y tiene herramientas locas que te ayudan a sincronizar los datos que ve el cliente con la base de datos real.

React: Es una biblioteca para crear interfaces de usuario de forma rápida y mantenible, orientado justamente a las SPA. Lo hizo y usa Facebook, y tiene cada vez más popularidad.

Nuestra solución tiende a una SPA, pero es muy mala porque tuvimos que hacer un objeto renderizador que vaya enchufando cosas a la página. No escala tener que hacer esto siempre. Además, seguro hasta nos va a traer problemas de vulnerabilidades.

Y además, aún no interactuamos con ninguna base de datos en donde guardar los mensajes.

Así que borremos todo el directorio y empecemos de cero.

Bajen meteor de meteor.com (tengan paciencia, puede tardar).

Crear una app de cero con meteor

Crear la aplicación

Abramos una consola. Vamos a la carpeta en donde queramos crear nuestra app y otros proyectos, y ponemos:

```
meteor create PdeParlar
```

¿Por qué PdeParlar? Porque estamos en PdeP, y vamos a parlar :D

Ese comandito nos va a crear una carpeta “PdeParlar”, con varias cosas dentro:

- Carpeta “.meteor”: Ahí hay información que va autogenerando nuestra app, así que jamás vamos a tocarlo a mano.
- Carpeta “node_modules”: Acá se van guardando las bibliotecas que vayamos importando que descarguemos de internet. Tampoco se toca a mano.
- Archivo “.gitignore”: Se escriben los archivos y carpetas que **no** hace falta subir al repositorio en cuanto usemos uno.
- Archivo “package.json”: Se escribe información de la app, y las dependencias (bibliotecas) que utiliza.
- Carpeta “client”: Todo lo que ejecutará el cliente de nuestra app. Cualquiera puede acceder y ver los archivos que están acá.
- Carpeta “server”: Todo lo que se ejecuta del lado del servidor. El cliente no tiene acceso a esta parte.

Borrar y crear cosas iniciales

Dentro de las carpetas client y server, nos creó algunos archivos para arrancar con algo. Por defecto, meteor cree que vamos a usar su motor (**Blaze**) para renderizar cosas, pero vamos a usar otro (**React**, el de facebook), así que el **contenido** de algunos archivos los vamos a borrar:

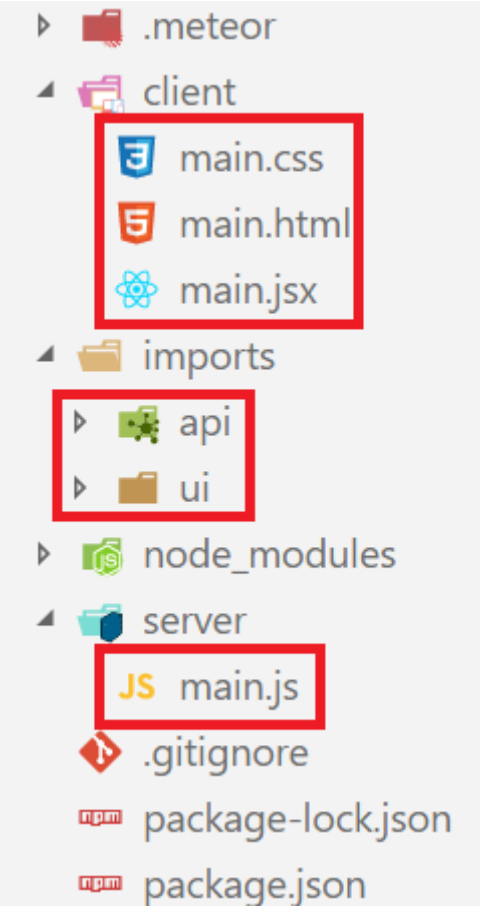
- client/main.html
- client/main.js → Además de borrar su contenido, lo vamos a renombrar por “main.js**x**”.

¿Qué es eso de “js**x**”? Es la extensión de archivo que le gusta a React, y que convierte cosas de forma automágica para hacernos trabajar menos. Es como un “js”, pero con agregados para renderizar.

También vamos a crear algunas carpetas para usarlas dentro de un rato:

- **/imports/ui/** (ui = user interface)
- **/imports/api/** (api = application programming interface)

Debería quedarnos algo así:



Dependencias

Luego, vamos a borrar Blaze y configurar algunas dependencias para usar React:

```
meteor remove blaze-html-templates
meteor npm install --save react react-dom react-addons-pure-render-mixin prop-types
meteor add react-meteor-data static-html
```

Al igual que hicimos antes, vamos a tener componentes. Y ellos sabrán renderizarse. Pero de forma más o menos cheta, como propone React.

Código inicial del cliente

Del lado del cliente, vamos al “main.html”. ¿Qué vamos a hacer acá?

El HTML básico y mínimo de nuestra página. Primero un título en los metadatos, y luego, un “cuerpo”:

```
<head>
 <title>PdeParlar</title>
</head>
<body>
 <div id="secciónParaRenderizar"></div>
</body>
```

¿Qué hará nuestro código js del cliente? Reemplazar esa sección con lo que queramos.

Así que en el “main.jsx” del cliente ponemos:

```
import React from 'react';
import { Meteor } from 'meteor/meteor';
import { render } from 'react-dom';
import PdeParlar from '../imports/ui/PdeParlar.jsx';
Meteor.startup(() => {
 render(<PdeParlar />, document.getElementById('secciónParaRenderizar'));
});
```

¿Qué hace eso? Bueno, primero, importamos algunas bibliotecas.

Luego, importamos “PdeParlar”, que es el componente que sabe renderizarse y que representa a nuestra página completa. Aún no lo creamos, ¡De a poco!

Por último, decimos “Cuando termines de inicializarte (startup), ejecutá esta función”. ¿Qué función? Renderizar nuestro componente PdeParlar en el div que pusimos en el .html.

Pero pará, ¿Qué es “`<PdeParlar />`”? Es un texto que sabe interpretar React. Lo que hará, es crear una instancia de una **clase** “PdeParlar”, y a partir de esa instancia, crear un componente con algunos agregados para poder manejarlo en la pantalla.

PdeParlar

Vamos a crear PdeParlar.jsx, adentro de /imports/ui

Ahí, vamos a crear un componente, que herede de… Component! Y vamos a hacer que sepa renderizarse:

```
import React, { Component } from 'react';

export default class PdeParlar extends Component {
 render() {
   return ( ??? );
 }
}
```

Ok, listo. Ya existe. Y… ¿Qué es lo que debería tener toda app? Un encabezado, y algo debajo.

Para hacerla sencilla, digamos que lo de abajo son directamente los mensajes. Los “listados” en HTML se escriben con el tag “ul” (unordered list). Así que podría ser algo así:

```
import React, { Component } from 'react';
```

```
export default class PdeParlar extends Component {
 render() {
       <header>
         <h1>PdeParlar</h1>
       </header>
       <ul>
         ????
       </ul>
 }
}
```

Bueno, a eso, lo vamos a englobar por un “div” únicamente para que después lo podamos acomodar mejor en la pantalla. Pero no es estrictamente necesario:

```
export default class PdeParlar extends Component {
 render() {
     <div className="container">
       <header>
         <h1>PdeParlar</h1>
       </header>
       <ul>
         ????
       </ul>
     </div>
 }
}
```

Ahora… ¿Qué renderizamos? Ta difícil la cosa. ¿Podríamos delegarlo, no? Un objeto PdeParlar debería saber de dónde sacar y mostrar un listado de mensajes.

Así que lo que vamos a poner en “????” va a ser un cacho de código que React va a saber ejecutar y enchufar ahí. Algo como:

```
export default class PdeParlar extends Component {
   return (
     <div className="container">
       <header>
         <h1>PdeParlar</h1>
       </header>
       <ul>
         {this.renderMensajes()}
       </ul>
     </div>
   );
 }
}
```

Y bueno, ¡Hace falta que sepa hacer eso! ¿Qué debería devolver? Debería ser una **lista de Mensajes**.

¿Y qué es un mensaje? ¡Un componente! ¡Él sabrá renderizarse!

Así que de la misma manera en la que creamos a PdeParlar, deberíamos crear un componente de tipo Mensaje, y crear muchos componentes a partir de la información cruda que tengamos de los mensajes.

Así que importamos esa clase (que aún no existe), y renderizamos un conjunto de mensajes con información hardcodeada horrible (por ahora).

```
import React, { Component } from 'react';
import Mensaje from './Mensaje.jsx';
export default class PdeParlar extends Component {
 renderMensajes() {
   return [
       <Mensaje key={1} emisor={"Santi"} texto={"Hola!"} />,
       <Mensaje key={2} emisor={"K2002"} texto={"Hola!"} />,
       <Mensaje key={3} emisor={"Santi"} texto={"Hola!"} />,
       <Mensaje key={4} emisor={"K002"} texto={"Y éste qué se fumó?"} />,
   ];
 }
 render() {
   return (
     <div className="container">
       <header>
         <h1>PdeParlar</h1>
       </header>
       <ul>
         {this.renderMensajes()}
       </ul>
     </div>
   );
 }
}
```

El problema es que no está bueno clavar la información directamente en cada uno de los mensajes.

Más adelante vamos a obtener la información cruda de una base de datos. Estaría bueno que a partir de eso, sin importar la cantidad de elementos, se arme una colección de mensajes.

¿Usando lo qué? ¡**Map**! Podría quedarnos así:

```
import React, { Component } from 'react';
import Mensaje from './Mensaje.jsx';
export default class PdeParlar extends Component {
 getMensajes() {
   // La idea es que cuando usemos una base de datos, solo haya que cambiar este método
   return [
     { _id: 1, emisor: "Santi", texto: 'Hola!' },
     { _id: 2, emisor: "K2002", texto: 'Hola!' },
```

```
      { _id: 3, emisor: "Santi", texto: 'Hola!' },
      { _id: 4, emisor: "K2002", texto: 'Y éste qué se fumó?' },
    ];
  }
  renderMensajes() {
    return this.getMensajes().map( (mensaje) =>
        <Mensaje key={mensaje._id} emisor={mensaje.emisor} texto={mensaje.texto} />
    );
  }
  render() {
    return (
      <div className="container">
        <header>
          <h1>PdeParlar</h1>
        </header>
        <ul>
          {this.renderMensajes()}
        </ul>
      </div>
    );
  }
}
```

Ah, ¿Y qué es esto del **export default**? Desde afuera, van a querer hacer un "import" de este archivo. Y nosotros estamos diciendo que esta clase será visible desde afuera para poder importarse, y será lo que se importe por defecto de todas las cosas que se puedan importar de este archivo.

Mensaje

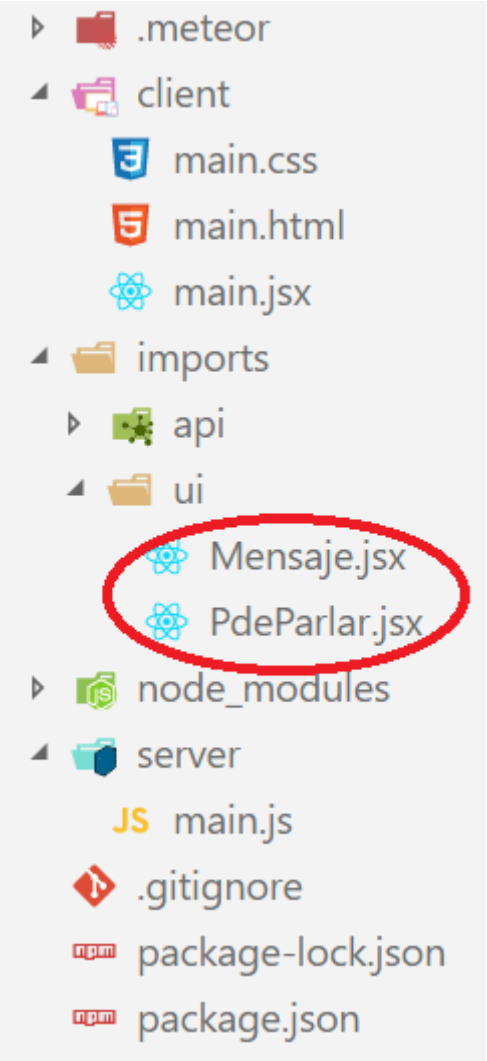
Ahora sí, finalmente, hagamos el componente Mensaje en el archivo **Mensaje.jsx**:

```
import React, { Component } from 'react';
export default class Mensaje extends Component {
  render() {
    return (
      <li>{this.props.emisor + ": " + this.props.texto}</li>
    );
  }
}
```

Eso, al renderizarse, muestra un "ítem de la lista" (li, list item), cuyo texto es el nombre del emisor, seguido de un dos puntos, espacio, y qué dijo.

El día de mañana podría tener partes locas para mostrar la foto del emisor, algún iconito de mensaje nuevo vs. mensaje ya leído, y lo que queramos.

La estructura de archivos debería estar así:



Un poco de estilos

Estos estilos son una porquería pero es mejor que nada (con más tiempo para preparar esta clase, habríamos hecho algo en bootstrap y escribiendo mucho menos). En **main.css**, poner esto:

```
body {
    font-family: sans-serif;
    background-color: #315481;
    background-image: linear-gradient(to bottom, #315481, #918e82 100%);
    background-attachment: fixed;

    position: absolute;
    top: 0;
    bottom: 0;
    left: 0;
    right: 0;

    padding: 0;
    margin: 0;

    font-size: 14px;
```

```
}

.container {
  max-width: 600px;
  margin: 0 auto;
  min-height: 100%;
  background: white;
}

header {
  background: #d2edf4;
  background-image: linear-gradient(to bottom, #d0edf5, #e1e5f0 100%);
  padding: 20px 15px 15px 15px;
  position: relative;
}

#login-buttons {
  display: block;
}

h1 {
  font-size: 1.5em;
  margin: 0;
  margin-bottom: 10px;
  display: inline-block;
  margin-right: 1em;
}

form {
  margin-top: 10px;
  margin-bottom: -10px;
  position: relative;
}

.new-task input {
  box-sizing: border-box;
  padding: 10px 0;
  background: transparent;
  border: none;
  width: 100%;
  padding-right: 80px;
  font-size: 1em;
}

.new-task input:focus{
```

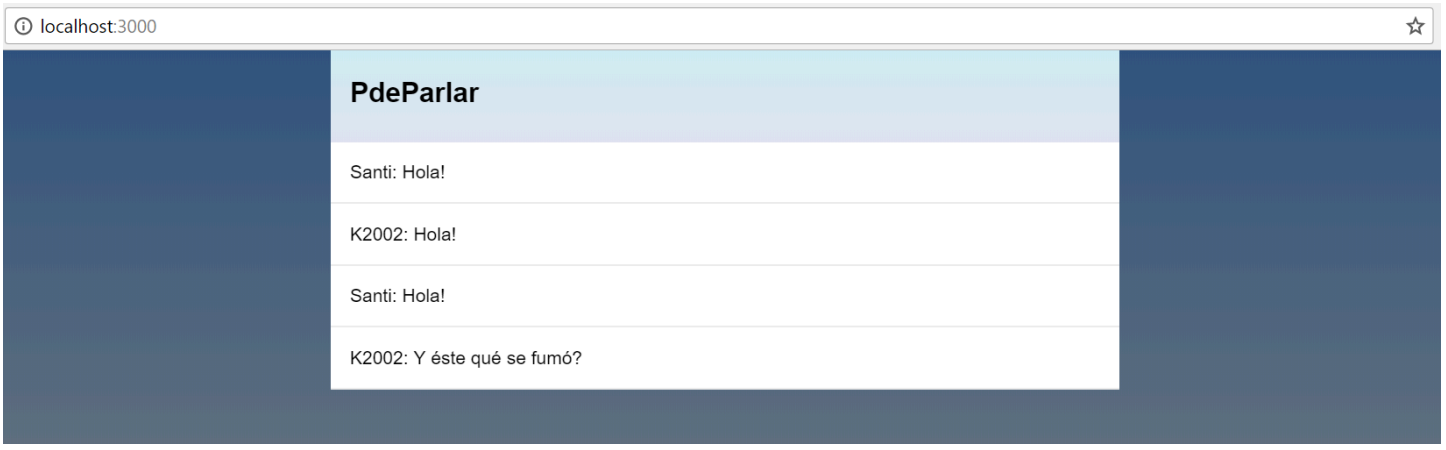
```
  outline: 0;
}

ul {
  margin: 0;
  padding: 0;
  background: white;
}

.delete {
  float: right;
  font-weight: bold;
  background: none;
  font-size: 1em;
  border: none;
  position: relative;
}

li {
  position: relative;
  list-style: none;
  padding: 15px;
  border-bottom: #eee solid 1px;
}

li .text {
  margin-left: 10px;
}

li.checked {
  color: #888;
}

li.checked .text {
  text-decoration: line-through;
}

li.private {
  background: #eee;
  border-color: #ddd;
}

header .hide-completed {
  float: right;
}
```

```
.toggle-private {
  margin-left: 5px;
}

@media (max-width: 600px) {
  li {
    padding: 12px 15px;
  }

  .search {
    width: 150px;
    clear: both;
  }

  .new-task input {
    padding-bottom: 5px;
  }
}
```

Lo ejecutamos en la terminal poniendo:

meteor

Si todo sale bien, deberíamos entrar a localhost:3000 y ver esto:



Usando una base de datos

Mensajes reales de una base de datos

Vamos a querer leer y guardar mensajes en una base de datos. Así que vamos a modificar lo siguiente.

Primero, hagamos un archivo “colecciónDeMensajes.js” en la carpeta imports/api, que tenga lo siguiente:

```
import { Mongo } from 'meteor/mongo';
export const colecciónDeMensajes = new Mongo.Collection('mensajes');
```

Esto crea un objeto colecciónDeMensajes, a partir de conectarse a una base de datos “Mongo”, para interactuar con una “tabla” (ponele) llamada “mensajes”.

Aclaración: Desde Mac no me funcionaban los nombres de archivos con tildes. Si a ustedes les aparece algún error poco entendible, pongan “coleccionDeMensajes.js” en el nombre del archivo y en los lugares en donde lo importemos.

Luego, en el server/main.js, agregaremos este import:

```
import '../imports/api/colecciónDeMensajes.js';
```

Por último, en el PdeParlar.jsx tenemos que leer de ahí esos datos.

Pero hay una particularidad. Cada vez que se agreguen mensajes a la base, vamos a querer enterarnos de ese cambio y mostrarlo en pantalla, **sin** tener que refrescarla a cada rato.

La forma de hacer esto, es crear un nuevo componente a partir de PdeParlar, que actualice su información a partir de una función que traiga los mensajes y otra magia de Meteor.

El archivo nos queda así:

```
import React, { Component } from 'react';
import { withTracker } from 'meteor/react-meteor-data';

import Mensaje from './Mensaje.jsx';

import { colecciónDeMensajes } from '../api/colecciónDeMensajes.js';

class PdeParlar extends Component {
 getMensajes() {
   return this.props.mensajes; // Nos lo van a asignar a una propiedad desde afuera cada vez que cambie, y automágicamente va a renderizarse todo de nuevo :D
 }

 renderMensajes() {
   return this.getMensajes().map( (mensaje) =>
       <Mensaje key={mensaje._id} emisor={mensaje.emisor} texto={mensaje.texto} />
   );
 }
 render() {
   return (
     <div className="container">
       <header>
         <h1>PdeParlar</h1>
       </header>
       <ul>
         {this.renderMensajes()}
       </ul>
     </div>
   );
```

```
 }
}

export default withTracker(() => {
 return {
   mensajes: colecciónDeMensajes.find({}).fetch(),
 };
})(PdeParlar);
```

Miramos el explorador y… ¡Todo desapareció!

Pfff, ¡Por supuesto! No creamos nada aún en la base de datos.

Creando un mensaje

Primero, vamos a crear a mano un mensaje directamente en la base.

Para esto, ejecutamos lo siguiente en otra terminal, en la carpeta del proyecto:

```
meteor mongo
```

Y grabamos lo siguiente:

```
db.mensajes.insert({ emisor: "Santi", texto: "Hola!", createdAt: new Date() });
```

Vemos la pantalla y… ¡Cambió! ¡Todos los clientes de nuestra app que hayan estado viendo esa pantalla, van a ver el cambio instantáneamente, sin recargar nada!

Crear mensajes desde la interfaz web

El chiste de chatear, es poder mandarnos mensajes y que todos los veamos.

Así que vamos a crear un formulario, que lea el emisor y el mensaje (sí, es súper rústico, con más tiempo hacíamos algo mejor).

¿En dónde lo mostramos? Para simplificar, elegimos mostrarlo en el encabezado.

Y este formulario sería… ¡Obvio, otro componente!

Así que agregamos un import, y cambiamos el render, en PdeParlar.jsx:

```
...
import CrearMensaje from './CrearMensaje.jsx';
...
        <header>
          <h1>PdeParlar</h1>
          <CrearMensaje />
        </header>
...
```

Y en el archivo CrearMensaje.jsx:

```
import React, { Component } from 'react';
```

```
import { colecciónDeMensajes } from '../api/colecciónDeMensajes.js';

export default class CrearMensaje extends Component {
 constructor(props) {
   super(props);
   this.state = {emisor: '', texto: ''};
 }
 handleSubmit(event) {
   event.preventDefault();
   console.log(this.state);
   const emisor = this.state.emisor;
   const texto = this.state.texto;
   colecciónDeMensajes.insert({
     emisor,
     texto,
     createdAt: new Date(), // current time
   });
   this.setState({texto: ''});
 }
 handleChange(event) {
   this.setState({[event.target.name]: event.target.value});
 }
 render(){
   return (
     <form className="new-task" onSubmit={(evento) => this.handleSubmit(evento)} >
       <input
           type="text"
           name="emisor"
           placeholder="¿Quién so' vo'?"
           value={this.state.emisor}
           onChange={(evento) => this.handleChange(evento)}
       />
       <input
         type="text"
         name="texto"
         placeholder="¡Escribí tu mensaje, ídolo!"
         value={this.state.texto}
         onChange={(evento) => this.handleChange(evento)}
       />
       <input type="submit" style={{display:"none"}}/>
     </form>
   )
 }
}
```

¡Y listo! Ahora todos podemos mandarnos mensajecitos :3

¿Por qué...

...JS?

Primero, para que veamos rápidamente que los conceptos de objetos, clases, herencia, delegación, etc., sí se usan en otros lenguajes y en todos los trabajos.

Además, se eligió este lenguaje en particular porque todos se van a encontrar con algo para hacer en JS en sus laburos tarde o temprano. Saber que se puede programar mejor, ayuda.

...NodeJS?

Porque se usa cada vez más. Y no solo para páginas web:

- Aplicaciones de escritorio (vean **Electron**).
- Aplicaciones mobile nativas (vean **Ionic** , o incluso **Meteor**, que usan "**Cordova**" para convertir el código).
- Aplicaciones para plaquetas (vean **Johnny-Five**).
- Y cada vez más.

...Meteor?

Por defecto viene con un transpilador para poder escribir en ES6 (en Node se tiene que poner "require" por ejemplo en vez de "import" hasta ahora), y detenernos en eso no aportaba a la clase.

Permitía avanzar más rápido en la clase sin detenernos en cosas relativamente complejas, como sockets.

Fácilmente pueden crear apps para celulares, con la sincronización de bases de datos bastante resuelta.

...React?

Porque el paradigma de "componentes" está avanzando cada vez más, y es muy común que se prefiera hacer que un componente sepa renderizarse.

De paso, vemos responsabilidades de objetos y polimorfismo :D

...MongoDB?

Porque es una base de datos relativamente fácil de usar, suele elegirse para combinarse con NodeJS, y parece ser buena escalando horizontalmente.

...todo esto?

Porque ustedes **pueden** hacer un sistema cliente servidor sin tanto esfuerzo. No llegamos a mostrar cómo subirlo a la nube, pero ni siquiera eso es tan complicado.

Así que **anímense**, e investiguen por su cuenta. Ya sea por hobby, para tenerla más clara en Proyecto Final y otras materias, para capacitarse por sus trabajos, o para hacer un sistema y "pegarla" llenándose de dinero y/o popularidar.